34th International Conference on
Automated Planning and Scheduling

June 1–6, 2024, Banff, Alberta, Canada

HPlan 2024

Proceedings of the 7th ICAPS Workshop on

Hierarchical Planning

Program Committee

Ron Alford	The MITRE Corporation
Gregor Behnke	University of Amsterdam
Pascal Bercher	The Australian National University
Maurice Dekker	University of Amsterdam
Humbert Fiorino	Université Grenoble Alpes
Christopher Geib	SIFT LLC
Daniel Höller	Saarland University, Saarland Informatics Campus
Prakash Jamakatel	University of Bundeswehr München
Jane Jean Kiam	Universität der Bundeswehr München
Pascal Lauer	Saarland University
Songtuan Lin	The Australian National University
Simona Ondrčková	Charles University
Kristýna Pantucková	Charles University
Mark Roberts	The US Naval Research Laboratory
Enrico Scala	University of Brescia
Dominik Schreiber	Karlsruhe Institute of Technology
Mohammad Yousefi	The Australian National University

Organizing Committee

Pascal Bercher	The Australian National University
Dominik Schreiber	Karlsruhe Institute of Technology
Simona Ondrčková	Charles University
Ron Alford	The MITRE Corporation

Preface

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

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

HPlan was founded in 2018 and is since then part of the annual International Conference on Automated Planning and Scheduling (ICAPS). This year's iteration of the HPlan workshop took place as a part of the 34th ICAPS in Alberta, Canada. The organizing committee consisted of a mix of senior researchers (Pascal Bercher, Ron Alford) and early career researchers (Simona Ondrčková, Dominik Schreiber). The organizers have received 10 paper submissions (9 research papers and 1 challenge paper) in total. Following a review process with 17 PC members and three reviews per paper (i.e., 1-2 papers per PC member), 9 papers were accepted (of which one was a challenge paper). The considered topics of submissions range from theoretical investigations over various aspects of solving hierarchical problems efficiently models to more application-driven research such as repairing plans.

The workshop took place on June 3, 2024 with 26 participants. An invited talk was given by Daniel Höller, awardee of the 2024 ICAPS Best Dissertation Award and one of the main developers of the state-of-the-art hierarchical planning framework PANDA, which has dominated the HTN tracks of the recent International Planning Competition 2023. The accepted papers were presented in the form of teaser talks and subsequent poster sessions, which gave plenty of room for discussion and exchange. In addition, two works already accepted or published at other venues were accepted for a presentation at HPlan. All accepted HPlan papers, except for one simultaneously accepted at another venue, are featured in the proceedings at hand. Presentations are made available on the YouTube channel of the workshop https://www.youtube.com/@hplan.

Overall, we hope that HPlan 2024 has achieved its goal of bringing together researchers to exchange new ideas and foster future research in hierarchical planning.

Pascal, Dominik, Simona, and Ron
HPlan Workshop Organizers,
June 2024

Invited Talk

This year, our invited speaker was *Daniel Höller*. We invited him due to his large large success in the IPC 2023, where almost all first and second placed systems were based on the progression based PANDA$_\pi$ planner, of which Daniel is one of the main developers.

HTN Planning as Heuristic Progression Search in the PANDA Framework

PANDA is a framework to solve different tasks around hierarchical planning. It comes with components for pre-processing, solving planning problems, and techniques for related tasks like plan and goal recognition, plan repair, and plan verification. It includes several solvers for hierarchical planning problems, namely heuristic plan space search, compilations to propositional logic, BDDs, and classical (i.e., non-hierarchical) planning, and heuristic progression (i.e., forward) search. In this talk, I will first give an overview of the different parts of PANDA. Then, we will have a detailed look at the forward progression search system: its pre-processing, search algorithm, and heuristics.

Bio

Daniel Höller, Post-doctoral researcher at the Universität des Saarlandes
Computer Science Department, Foundations of Artificial Intelligence (FAI) Group

Daniel Höller has been a post-doctoral researcher in Jörg Hoffmann's Foundations of AI Group at Saarland University since 2020. Before that, he was at the Institute of Artificial Intelligence at Ulm University, where he did this PhD on hierarchical planning (mainly HTN planning), supervised by Susanne Biundo. His PhD thesis on hierarchical planning won the ICAPS Best Dissertation Award in 2024. Besides HTN planning, his work is concerned with lifted planning, and with the combination of machine learning and planning.

He is interested in many aspects of HTN planning like the expressivity of different formalisms, translations of related problems like plan and goal recognition and plan verification, and especially solving techniques. He has worked on grounding, heuristic plan space and progression search, translations to classical planning, and to propositional logic. He is a main developer of the planning systems PANDA, TOAD, and LiSAT. At the 2023 International Planning Competition, the winners of all 6 tracks on HTN planning have been based on PANDA, as well as 5 out of 6 runner-ups.

Table of Contents

Scientific Papers

A Comparative Analysis of Plan Repair in HTN Planning

Robert P. Goldman and Paul Zaidins and Ugur Kuter and Dana Nau ..... 1 – 9

An ILP Heuristic for Total-Order HTN Planning

Conny Olz and Alexander Lodemann and Pascal Bercher ..... 10 – 18

Barely Decidable Fragments of HTN Planning

Maurice Dekker and Gregor Behnke ..... 19 – 26

Correcting Totally-Ordered Hierarchical Plans by Action Deletion and Action Insertion

Kristýna Pantůčková and Roman Barták ..... 27 – 35

Laying the Foundations for Solving FOND HTN problems: Grounding, Search, Heuristics (and Benchmark Problems)

Accepted at the 33rd International Joint Conference on Artificial Intelligence (IJCAI 2024)

Mohammad Yousefi and Pascal Bercher ..... URL-not-yet-available

Redundant Decompositions in PO HTN Domains: Goto Considered Harmful

Roland Godet and Arthur Bit-Monnot and Charles Lesire-Cabaniols ..... 36 – 44

Towards Search Node-Specific Special-Case Heuristics for HTN Planning – An Empirical Analysis of Search Space Properties under Progression

Lijia Yuan and Pascal Bercher ..... 45 – 53

Weighted Randomized Anytime Planning in Pyhop

Gabriel J. Ferrer ..... 54 – 58

Challenge Papers

Toward Planning with Hierarchical Decompositions and Time-frames

Mica Gardone and Rogelio E. Cardona-Rivera ..... 59 – 63

A Comparative Analysis of Plan Repair in HTN Planning

Robert P. Goldman[1], Paul Zaidins[2], Ugur Kuter[1], Dana Nau[2]

[1]SIFT, LLC
[2]University of Maryland, College Park

Abstract

This paper reports an analysis of three recent hierarchical plan repair algorithms: SHOPFIXER, IPYHOPPER, and REWRITE. We compare these algorithms qualitatively, and evaluate their performance, quantitatively, in a series of benchmark planning problems, informed by our qualitative analysis. A critical part of the qualitative comparison is that REWRITE, a problem-rewriting technique, has a substantially different and more restrictive *definition* of plan repair than the other two systems. Understanding this distinction will be important when choosing a repair method for any given application. Our results explain the runtime repair performance of these systems as well as the coverage of the repair problems solved, based on algorithmic properties such as chronological backtracking vs. backjumping over plan trees.

1 Introduction

Plan repair has been shown to provide advantages over generating new plans from scratch both in terms of planning runtime and plan stability – the amount of plan content that is retained between the original and repaired plans (Fox et al. 2006). Fox, *et al.* showed that plan repair could provide new plans faster, and with fewer revisions, than replanning *ab initio* in the face of disruptions. They used the term "stability" to refer to the new plan's similarity to the old one, by analogy to the term from control theory. The term "minimal perturbation" has been used synonymously (Cushing and Kambhampati 2005). To be precise, "stability" is actually a *relation*: a solver is stable if the size of the change in the output is proportional to the size of the change in its input: at least in theory, a minimal perturbation solver could actually be unstable. Plan stability is particularly important for human interaction, as users are confused by radical changes to plans introduced in response to trivial upsets.

The concept of stable plan repair has been generalized from classical planning to Hierarchical Task Network (HTN) planning. Early work on hierarchical plan repair introduced validation graphs in the context of hierarchical and partial-order causal link planning, where plan repair used validation graphs to identify disruptions and patches to the partial-order plans (Kambhampati and Hendler 1992). Extending classical plan repair on sequences of actions, hierarchical repair algorithms provide localization of errors and failures and problem refinement methods that take advantage of such localizations to provide better stability (Robert P. Goldman, Ugur Kuter, and Richard G. Freedman 2020).

Over the years, there have been great strides in HTN plan repair in which a variety of repair algorithms have been proposed. The three that we consider are SHOPFIXER (Robert P. Goldman, Ugur Kuter, and Richard G. Freedman 2020), IPYHOPPER (Zaidins, Roberts, and Nau 2023), and an unnamed algorithm that we will call REWRITE (Höller et al. 2020b). These build on several previous methods (Ayan et al. 2007; Kuter 2012; Bansod et al. 2022; Bercher et al. 2014).

We compare SHOPFIXER, IPYHOPPER, and REWRITE qualitatively, and evaluate their quantitative performance in a series of benchmark planning problems in the light of our qualitative analysis. Our results demonstrate the following:

- Because of differences in their notion of what repairs are permissible and how to go about doing them, there are differences in which repair problems REWRITE can solve as opposed to IPYHOPPER and SHOPFIXER, which share a definition. The three algorithms also differ in what kinds of repairs they make.
- The REWRITE repair method, which must replicate already-executed actions, involves extensive amount of re-derivation of plans, as can be seen in its worse runtimes for all of the domains.
- Chronological backtracking during hierarchical repair involves blindly trying a large number of subtrees of the original plan tree, most of which do not contribute to repairing the plan. In more complex problems, semantic (*i.e.*, causal) backjumping yields better performance, as can be seen in the Openstacks domain and the more difficult Rovers problems.
- Less-expensive simulation lookahead for repair provides a better payoff than extensive work in building data structures (*e.g.*, explicit causal links) to speed backtracking and backjumping in problems of modest scale, such as the Satellite domain.

An additional contribution of our work is to provide the first publicly-available implementation of the REWRITE repair algorithm. We also extended it to work in lifted domains, which is critical for completeness, since practical grounding methods typically include problem-specific pruning. Such pruning may compromise the completeness of the

plan repair method, since disturbances may render new parts of the state space reachable.

2 Hierarchical Plan Repair Strategies

The three algorithms analyzed in this paper have important qualitative differences that color the experimental results and are critical to their interpretation. First, REWRITE's definition of plan repair is more stringent than the others; we have found benchmark planning problems IPyHOPPER and/or SHOPFIXER solve but REWRITE does not. The second difference is between SHOPFIXER and IPyHOPPER. SHOPFIXER attempts to detect when a plan *will be* invalid, before any actions actually fail; it invests in data structures and computation in order to detect problems as soon as possible. IPyHOPPER's projections are not model-based: it relies on an external simulation to do projection for it, instead of having an internal action model as most planners do. These differences lead to different plan repair behaviors.

These differences are not simply a matter of one repair method being "better" than another: instead, different repair methods are better in different situations. The more stringent definition offered by REWRITE is better when an HTN method library captures important considerations about what sequences of actions are and are not correct plans; the more relaxed definition better when the precise trajectory is less important. SHOPFIXER's plan repair approach is better if the costs of wrong actions are higher than the costs of computation, *e.g.* when actions are particularly expensive, or when deferring repair could leave the agent trapped in a dead end. When a situation is more forgiving and when it changes frequently, SHOPFIXER's aggressive repair strategy will not be worthwhile. In the following, we give simple examples that illustrate these differences.

The REWRITE paper (Höller et al. 2020b) defines a repaired plan as one that, among other considerations, has a plan (decomposition) tree that is a refinement of the plan tree of the initial plan. This definition can exclude some repairs that seem intuitively plausible.

Consider an HTN plan domain for inter-city travel that has two alternative methods: rail and air travel. For rail travel, we take the bus to the station, embark, travel, and then debark. Similarly, to fly we take the bus to the airport, get on the plane, fly, and then deplane. Each method has the precondition that the train station (resp., airport) be open. Starting from home, Panda (Höller et al. 2021) and SHOP3 (Goldman and Kuter 2019) both can find plans for air and rail travel.

Consider the plan repair problem that occurs when we take the bus to the train station and discover that the train station is closed. SHOPFIXER and IPyHOPPER will detect the problem, identify that the original `rail-travel` method cannot be fixed, and switch to air travel. However, REWRITE cannot repair this plan, because the resulting plan – bus to train station, bus to airport, embark, fly, deplane – is ill-formed: no expansion of the top level goal contains *both* "bus to train station" *and* "bus to airport."

This highlights a tradeoff between flexibility and efficiency that the authors of HTN domain descriptions typically face. If the HTNs use strong search-control strategies and knowledge to make planning efficient by quickly finding a good and acceptable solution, then those HTNs are usually not flexible enough to allow exploring alternative plans for in hierarchical plan repair. As the above example shows, the REWRITE algorithm commits to the prefix of the hierarchy for planning reasons, but that excludes possible repairs at the higher levels of the hierarchy when a discrepancy occurs. SHOPFIXER and IPyHOPPER are similar in that limitation for general cases but they provide backtracking and backjumping strategies that can alleviate this limitation in some classes of domains. Section 4 discuss examples of this phenomenon in our experimental results.

The REWRITE paper gives an example that shows the rationale for its more restrictive definition of plan repair. In this example, the agent drives through a city that has congestion pricing, and must pay a toll for each road segment driven in the congestion zone. Solution plans have the form $x^* a^n b^n x^*$: there are actions before and after travel in the congestion zone (the two x^*'s), then n segments traveled in the congestion zone (a^n), followed by n toll payments (b^n). The structure of the HTN methods is the mechanism that enforces the a^n-b^n balance, so if extraneous actions were allowed, incorrect (unbalanced) plans could be derived.

SHOPFIXER and IPyHOPPER share the same model of plan repair, which holds that any HTN method may be redone – *i.e.*, its plan regenerated and then executed from the method's beginning, as long as the method's preconditions hold in the state in which it begins. As we have seen above, that means that the "intercity-travel" task may be restarted after the agent has reached the train station and discovered that it is closed. The difference between the two is how they attempt to detect future action failures.

When building a repairable plan, SHOPFIXER creates a decomposition tree that records task decompositions into subtasks, with cross-links from actions that establish facts to the actions and methods whose preconditions consume those facts. This trades some pre-computation and storage for more rapid detection of possible plan failures – and detection of cases where unexpected effects do not cause plan failures. An example SHOPFIXER tree is shown in Figure 1.

IPyHOPPER, in contrast, does not precompute any data structures for identifying plan failures. Instead, when notified of a plan disturbance, it simulates the existing plan forward in time to find impending failures. When it finds such a failure, it unexpands the simulated action's parent node in the solution tree, and attempts to find a new decomposition for that node that will not fail. If no such decomposition exists, the parent's parent is similarly unexpanded; this continues up the solution tree until either a valid decomposition is found or the root is reached, which means no repair exists.

If a valid decomposition is found, IPyHOPPER restarts the simulation at the leftmost action of the decomposition in postorder traversal. This continues until either the plan completes successfully or another failure is simulated. If another failure is simulated, the repair process is repeated and eventually either the simulation will complete and the plan is repaired or the root will be reached and repair has failed.

We now will briefly summarize the three hierarchical plan repair algorithms that we have evaluated in this work.

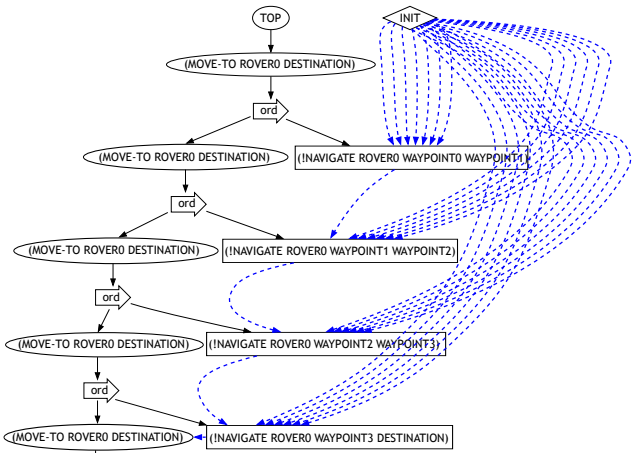


Figure 1: SHOP3 plan tree for a rover plan; decomposition edges are in black; dependency cross-edges in dashed blue. Edges from the diamond node represent dependencies on initial state facts.

SHOPFIXER Plan Repair

SHOPFIXER (Robert P. Goldman, Ugur Kuter, and Richard G. Freedman 2020) is a method for repairing plans generated by the forward-searching HTN planner, SHOP3. It uses a graph of causal links and task decompositions to identify a minimal subset of the plan that must be fixed. SHOPFIXER extends the notion of plan repair stability introduced by (Fox et al. 2006), and further develop their methods and experiments, which showed the advantages of plan repair over replanning.

The basic idea behind SHOPFIXER's plan repair approach is very simple: when a disturbance is introduced into the plan, SHOPFIXER finds the minimal subtree of the plan tree that contains the node whose preconditions are clobbered by that disturbance: the failure node. If there is no such node, then the disturbance does not interfere with the success of the plan. SHOPFIXER will then repair the plan, starting with the minimal subtree. To find the minimal subtree around a failure node, SHOPFIXER finds the first task in the plan that is potentially "clobbered" (rendered unexecutable) by that disturbance, and restarts the planning search from that task's immediate parent in the HTN plan (since that was where that task was chosen for insertion into the plan). This plan repair is done by *backjumping* into the search stack for SHOP3 and reconstructing the compromised subtree without the later tasks. Note that the first clobbered task may be either a primitive task or a complex task. Furthermore, if p is the parent of child c in an HTN plan, then p's preconditions are considered chronologically prior to c's, because it is the satisfaction of p's preconditions that enables c to be introduced into the plan: if both p and c fail, and we repair only c, we will still have a failed plan, because after the disturbance, we are not licensed to insert c or its successor nodes.

SHOPFIXER restarts the planning search by backjumping to the corresponding entry in the SHOP3 search stack, which it retains, and updating the world state at that point with the effects of the disturbance. When restarting the planning search, SHOPFIXER "freezes" the prefix of the plan that has already been executed, as well as the deviation and its effects. It may backjump to decisions prior to the deviation, for example, if the immediate parent of the failed task is the top level task of the problem, but it cannot undo the effects of an action that is already done. SHOPFIXER returns a repaired plan that is made up of the prefix before the disturbance, the disturbance, and the repaired suffix.

IPyHOPPER Plan Repair

IPYHOPPER (Zaidins, Roberts, and Nau 2023) is a progression based HTN planner written in Python. The primary distinction between IPYHOPPER and other plan repair methods is that it does not rely on a projection algorithm. Instead, it uses an external simulator to predict the effects of planned actions.

IPYHOPPER's planning engine is an augmented version of the prior IPyHOP planner (Bansod et al. 2022). For planning, input is in the form of an initial task list, initial state, and domain description. The domain description includes tasks, primitive actions, and method definitions. The initial tasks are repeatedly decomposed into simpler tasks and then finally actions based on the domain description. The decomposition process forms a solution tree by a depth-first traversal and every intermediate state is saved in the tree for backtracking. When a task cannot be successfully decomposed, a new decomposition is attempted of the most recent task expanded. When every task has been decomposed and all actions' preconditions are established, planning has successfully completed and the actions of the tree in preorder constitute the plan. If the planner backtracks to the sentinel root node, which is the parent of all input tasks, planning has failed: no decomposition can achieve the task list in order.

For plan repair, IPYHOPPER restarts the planning process at the parent of the immediate parent of a failed action using the current state in place of the stored state. Initially, IPYHOPPER restricts the process to this subtree and only backtracks further up the tree when all decompositions in the subtree fail. Once it finds a valid decomposition, we simulate the action execution going forward. If our simulation completes, the plan is repaired and the process is terminated. If IPYHOPPER encounters a future simulated failure, it will redo the repair process. This simulation-repair cycle continues until either the plan successfully repaired or root is reached, indicating that no repaired plan is possible.

Plan Repair by Problem Rewriting

The two methods we have discussed above both share the core pattern of resuming the planning process after some appropriate change to the search process. They generate a repaired plan by redoing some portion of planning process. The *rewrite* method of Höller, *et al.* (Höller et al. 2020b) is very different: it operates by generating a new problem and domain definition that is solvable iff the plan can be repaired. These definitions are generated by combining the original problem and domain definitions, the original solution (plan), the position reached in execution, and the disturbance. They

do not provide a method for determining whether a plan repair is actually required, so a repair problem must be solved after every disturbance. A key advantage of their algorithm is that it is not specific to any particular HTN planner: any planner that accepts their input format will work.

The central intuition behind the problem/domain method is to force the planner to build a new plan that has as a prefix the set of actions that were executed before the disturbance. The final action in this prefix is modified so that its effects include the disturbance effects. Any decomposition plan that is consistent with the observed actions and that contains an executable suffix that performs all the initial tasks, and achieves any specified goals.[1] This definition accounts for the distinction between repairs permitted by rewrite and those of SHOPFIXER and IPYHOPPER. The latter systems accept repaired plans that include methods that have been *abandoned* and their tasks achieved through new decompositions not consistent with the original plan: rewrite does not.

Rewrite algorithm implementation No runnable implementation of Höller, *et al.* 's algorithm was available,[2] so we implemented it ourselves; we will share our implementation on GitHub under an open source license. Our implementation follows the original definition in using HDDL for its input and output formats. Our implementation *differs* from the original definition in being able to handle action and method *schemas*, rather than only handling ground actions and methods (*i.e.*, it is a *lifted* implementation). This required extensions to some parts of the original algorithm.

REWRITE generates a new planning domain and problem, so in theory it may be coupled with any HTN planner. In practice, since HTN problem definitions are less standardized than classical ones, there are limits to this flexibility. Our implementation returns a lifted plan repair problem that can be used directly by the lifted HTN planner SHOP3 SHOP3 (Goldman and Kuter 2019), and that can be grounded for use with grounded planners.

We had originally intended to report on experiments that used both SHOP3 and Panda (Höller et al. 2021) as planners for the repair problems. Unfortunately, we found that Panda was unable to handle the benchmark domains we have used in our experiments, namely, Rovers, Satellite, and OpenStacks as reported next section. These domains all use the ADL dialect of PDDL/HDDL, featuring quantified goals and conditional effects. The parsing and grounding methods used in Panda were not able to handle the demands of ADL domains (we confirmed this with Panda developers), so we had to limit ourselves to using only SHOP3.[3] We refer to this combination as Rewrite-SHOP3. However, since SHOP3 and Panda would be solving the same problems, we are still testing the essential features of the rewrite algorithm.

[1]Their HDDL (Höller et al. 2020a) input notation permits problems that have both initial task networks *and* goals.

[2]Daniel Höller, personal communication, 26 September 2023.

[3]None of the satellite problems could be parsed by Panda in 5 minutes, only 7 of the rovers, and 9 of the openstacks. The 7 rovers problems could be grounded, but only 7 of the 9 openstacks problems. Details available upon request.

Summary The REWRITE algorithm is most appropriate for problems where the structure of the plan tree is critical to correctness (e.g., the toll example, where movements and payments must be balanced by the tree), but it may fail where disturbances put the agent into a dead end that it will have to "back out" of. IPYHOPPER and SHOPFIXER will be the opposite: both allow for deviations in the plan tree, so both will easily handle cases that involve backing up to reverse a deviation and then resuming. They differ in that IPYHOPPER will more efficiently handle deviations with immediate impact, and SHOPFIXER can better handle deviations with delayed impact, at the expense of computation that will be wasted on simple cases.

3 Experimental Design

We tested SHOPFIXER, IPYHOPPER, and Rewrite-SHOP3 on a set of identical initial plans and disturbances from three domains: **rover**, **satellite**, and **openstacks**. These are all HTN domains, formalized equivalently in HDDL and in SHOP3's input language. All of these domains were adapted from International Planning Competition (IPC) PDDL domains predating the HTN track, with HTN methods added and PDDL goals translated into tasks. These domains (with slightly different disturbances) were used in a previously published evaluation of the SHOPFIXER plan repair method (Robert P. Goldman, Ugur Kuter, and Richard G. Freedman 2020).

The Satellite and Rover domains each have 20 problems, and the Openstacks domain has 30. For each domain, we ran 50 batches, where each batch was a run of each problem with one injected disturbance, randomly chosen and randomly placed in the original plan. All original plans were generated by SHOP3; they were translated into HDDL for IPYHOPPER and Rewrite-SHOP3. For IPYHOPPER, all inputs were translated into JSON to avoid the need for a new HDDL parser. We wish to emphasize that each repair method started with the same plan and same disturbance in each batch. So for each problem there are 50 disturbance examples, which we ran on all three of the algorithms. Runtimes were measured to the nearest hundredth of a second. All runtimes were wall-clock times, not CPU times: we were concerned that comparing Python CPU times with Common Lisp CPU times might not be valid. In the event, differences between CPU times and wall-clock times were negligible.

Planning Domains

Here we briefly introduce the domains and deviation operators used in our experiments. All three domains used the PDDL/HDDL ADL dialect. Domains were modeled equivalently in both HDDL and SHOP3; we have indicated below where the SHOP3 and HDDL domains diverged. All the *original*, to-be-repaired plans were generated by SHOP3.

Deviations were modeled similarly to actions, with preconditions. Deviation preconditions and effects were defined in ways that aimed to avoid making repair problems unsolvable. Non-trivial plan disturbances were difficult to model without rendering problems unsolvable because the limited expressive power of PDDL (and by extension HDDL)

forced ramifications to be "compiled into" action effects (*e.g.*, counting the number of open stacks in the Openstacks domain), introducing dependencies that often could not be undone (*e.g.* failing the "send" operation in Openstacks had to also restore the relevant order to "waiting").

Rovers The Rovers domain is taken from the third IPC in 2002. Long & Fox say it is "motivated by the 2003 Mars Exploration Rover (MER) missions and the planned 2009 Mars Science Laboratory (MSL) mission. The objective is to use a collection of mobile rovers to traverse between waypoints on the planet, carrying out a variety of data-collection missions and transmitting data back to a lander. The problem includes constraints on the visibility of the lander from various locations and on the ability of individual rovers to traverse between particular pairs of waypoints." (Long and Fox 2003) Rovers problems scale in terms of size of the map, number of goals, and the number of rovers. Disturbances applied include losing collected data; decalibration of cameras; and loss of visibility between points on the map. For the Rovers problem, the SHOP3 domain uses a small set of path-finding axioms to guide navigation between waypoints. To avoid infinite loops in the navigation search space, IPYHOPPER does not use lookahead in the waypoint map, but it does check for and reject cycles in the state space.

Satellite The Satellite problem also premiered in 2002, and is described as "inspired by the problem of scheduling satellite observations. The problems involve satellites collecting and storing data using different instruments to observe a selection of targets." (Long and Fox 2003) Disturbances used were changes in direction of satellites, decalibration, and power loss. Problems scale by number of instruments, satellites and image acquisition goals.

Openstacks Openstacks was introduced, as a translation of a standard optimization problem, in IPC 2006:

> The Openstacks domains are ... based on the "minimum maximum open stacks" combinatorial optimisation problem ... A manufacturer has a number of orders, each for a combination of different products. Only one product can be made at a time, but the total required quantity of that product is made at that time. From the time that the first product requested by an order is made to the time that all products included in the order have been made, the order is said to be "open" and during this time it requires a "stack" .... The problem is to order the making of the different products so that the maximum number of stacks that are in use simultaneously ... is minimised. (Gerevini et al. 2009)

Problems scale by number of orders, number of products, and number of products in a single order. Deviations include removing products that were previously made and causing the shipping operation to fail. Deviations were particularly difficult to add to Openstacks without introducing dead ends into the search space, because there are consistency constraints on the state that are only implicit in the operators. Therefore, to make repair possible, we needed to add a "reset" operation that would reset an order to "waiting" mode from "started." The SHOP3 domain for Openstacks included axioms for a cost heuristic. Note that this is a common difficulty in plan repair: typically domains are not written in such a way that recovery is possible if a plan encounters disturbances because limitations in expressive power means that ramifications must be programmed into the operator definitions. Furthermore, since state constraints (*e.g.*, graph connectivity in the logistics domain) are not and cannot be captured in PDDL, one can inadvertently make dramatic changes to problem structure by introducing disturbances; cf. Hoffmann (2011) on problem topology.

4 Results

Satellite The Satellite domain was the easiest for all three repair methods. Both IPYHOPPER and SHOPFIXER solved all of the repair problems in our data set, and solutions were found quickly. Rewrite-SHOP3 solved the majority of the problems, between 60% and 85% of them (Figure 2). Inspection showed that it correctly solved all the problems that did not need repair (*i.e.*, did not time out rederiving the plan). Figure 3 gives the runtimes for all three methods, using $\log_{10}$ because of the range of values. IPYHOPPER runtimes are slightly better than SHOPFIXER on average. The rewrite times are almost uniformly worse, and scale worse as the problem size grows. The results for Rewrite-SHOP3 are not surprising, since proving unsolvability may take longer. Indeed, plotting the runtimes for success and failure separately, demonstrates that (Figure 5). Interestingly, there are no failures due to timeout: Rewrite-SHOP3 is able to prove unsolvable all of the unrepairable problems). While the IPYHOPPER times are generally the best on average, its times vary more widely: see Figure 4.

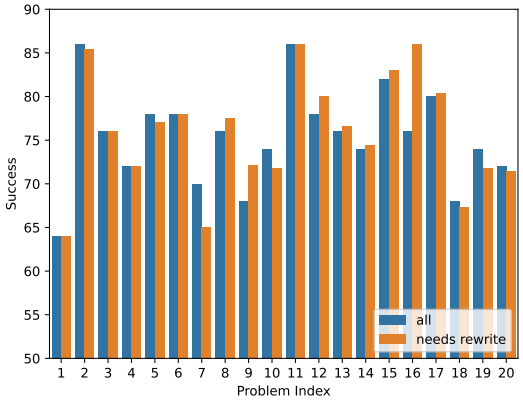


Figure 2: Rewrite-SHOP3 success rates for the satellite repair problems (note the y axis runs only from 50-90%).

Rovers The Rovers repair problems were more difficult than Satellite, and none of the repair methods solved all of them. Figure 6 shows success percentages. Note the outliers for IPYHOPPER and SHOPFIXER in problems 3, and 6 and for IPYHOPPER in problems 10 and 20. Gen-

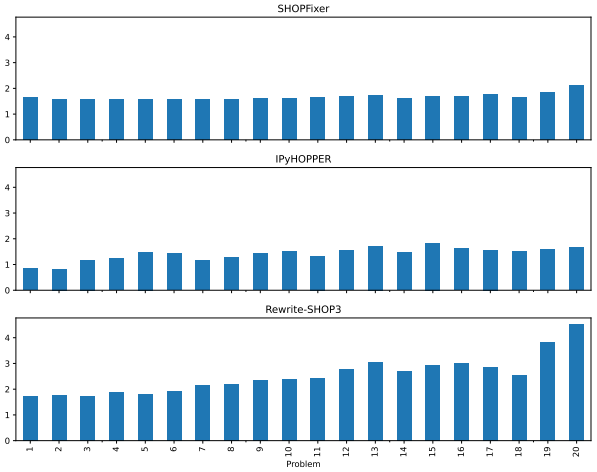


Figure 3: Satellite problem runtimes for all repair algorithms, in msecs (rounded up), plotted in $\log_{10}$.

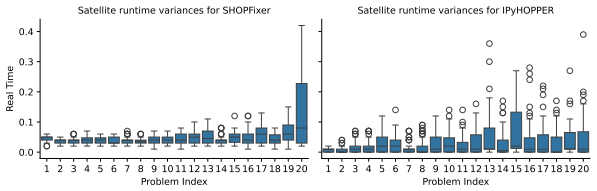


Figure 4: Variance of Satellite problem runtimes.

erally, SHOPFIXER is more successful than IPYHOPPER, as shown in Table 1.

Generally, while Rewrite-SHOP3 was less successful than the other algorithms, it tracks their results except for problems 14 and 15, where the other two are uniformly successful, but Rewrite-SHOP3 is only 84% (42/50) successful.

Runtimes are graphed in Figures 7, and 8. We plot the successful and failed runs separately, because the failed runs include both cases where an algorithm proves that the problem is unsolvable and cases where it simply runs out of time (time limit was set at 300s).

Again, IPYHOPPER is generally faster, but SHOPFIXER scales better with problem difficulty. For IPYHOPPER, problem 3 is an outlier in elapsed time, SHOPFIXER has issues with problem 6, and for Rewrite-SHOP3 the last two problems, and especially problem 20, are notably more difficult. As before, on the successful problems, IPYHOPPER has a higher runtime variance than SHOPFIXER (see Figure 9; variance plotted in seconds, not transformed).

Openstacks For Openstacks, both SHOPFIXER and Rewrite-SHOP3 solved all the repair problems. IPYHOPPER solved *almost* all, but failed for a small number (see Table 2). The runtimes, graphed in Figure 10 clearly show that Rewrite-SHOP3 and IPYHOPPER do not scale well on the more difficult problems, with Rewrite-SHOP3 notably worse. IPYHOPPER runtimes for problems 3, 6, and 12 are outliers: they are much more difficult than similarly-

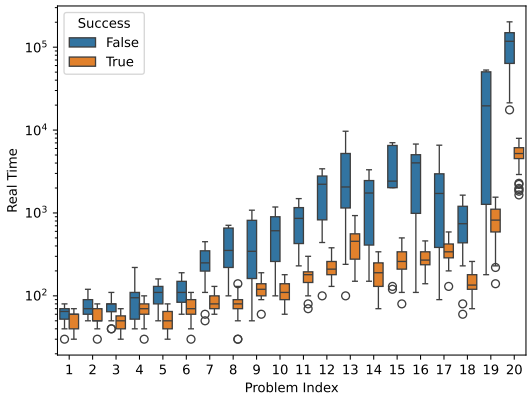


Figure 5: Satellite problem runtimes for Rewrite-SHOP3, comparing successful trials versus failed trials, plotted in $\log_{10}(\text{msec})$.

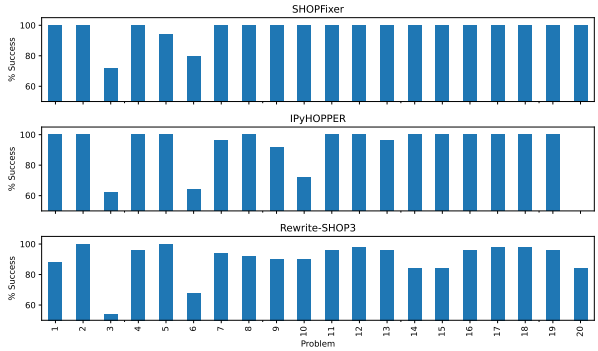


Figure 6: Success rates for the Rovers repair problems for each of the three algorithms.

numbered problems.

5 Discussion

Common Features Across all of the domains, REWRITE is less time-efficient than the other two repair methods. This is due to the fact that it replans *ab initio*, albeit against a new problem that forces the plan to replicate already-executed actions. This involves an extensive amount of rework, as can be seen very clearly in the Openstacks problems, which have the highest runtimes for generating initial plans. Note that this could likely be substantially improved by heuristic guidance that would direct the early part of planning towards methods that replicate actions previously seen and that avoid infeasibly introducing new actions.

Satellite It is unsurprising that REWRITE cannot solve some problems that the other two algorithms solve. Many of the repairs for satellite simply involve immediately restoring a condition deleted by a disturbance—and if it was a condition that the plan established, a re-establishment typically will not work (see Section 2). Perhaps the surprise is that *any*

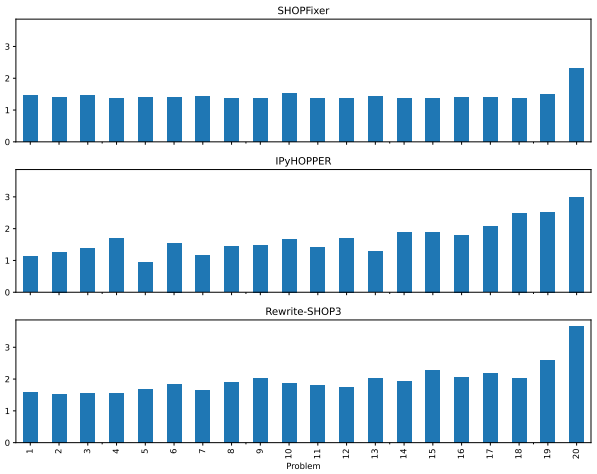


Figure 7: Runtimes for the Rovers repair problems in $\log_{10}(\text{msec})$ for each algorithm. These include only those problems solved successfully by the algorithm in question.

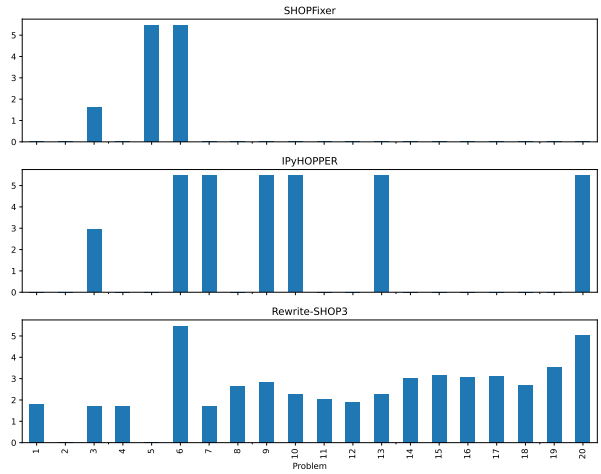


Figure 8: Runtimes for the Rovers repair problems in $\log_{10}(\text{msec})$ for each algorithm. These are only the problems *not* solved successfully by the algorithm in question.

	Solver	
Problem	IPyHOPPER % Success	SHOPFixer % Success
3	62%	**72%**
5	**100%**	94%
6	64%	**80%**
7	96%	**100%**
9	92%	**100%**
10	72%	**100%**
13	96%	**100%**
20	24%	**100%**

Table 1: Success rates for Rover problems where either IPyHOPPER or SHOPFixer did not solve all repairs. Problems not listed were all solved by both repair methods.

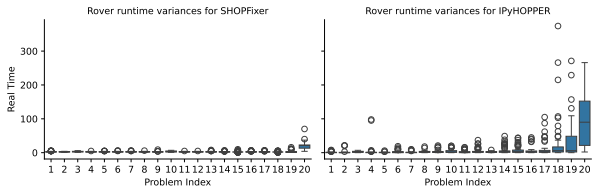


Figure 9: Variance in runtimes for the Rovers repair problems for IPyHOPPER and SHOPFixer. Only successfully solved problems are plotted.

of these scenarios are successfully repaired by Rewrite. We investigated further, and found that Rewrite could handle all of the cases that did not require repair (where the disturbance did not defeat any action preconditions). Removing those cases gives the success rates shown in Figure 2.

Here is an example of how Rewrite's definition of repair makes it unable to solve a problem handled by the other two systems. In this repair problem , `instrument0` becomes decalibrated after it has been calibrated and pointed at its observation target (`phenomenon6`). The way the domain is written, calibration occurs only in a sequence of calibration then observation. Thus there is no plan in which two calibration operations are not separated by an observation, so repair by rewrite is impossible. The other two methods simply treat the preparation task as having failed, and re-execute the calibration, because their repair definition is more permissive.

For this domain, IPyHOPPER is generally faster than SHOPFixer, although it is implemented in interpreted Python rather than compiled Common Lisp. This is probably accounted for by the fact that SHOPFixer invests in building complex plan trees that include dependency information (see Figure 1) in order to more rapidly identify the location of precondition violations and their implications. SHOPFixer also uses this information to *backjump* (Dechter 2003; Gaschnig 1979) to the point of failure, instead of relying on chronological backtracking, as does IPyHOPPER. For these simple problems, SHOPFixer's added effort is generally not worthwhile. We note that the *variance* of IPyHOPPER's runtimes is wider than that of SHOPFixer, and that there are more outliers (Figure 4).

Rovers There were several Rovers problems where even IPyHOPPER and SHOPFixer could not find solutions—but the three algorithms behaved quite differently in these cases. There were 99 Rovers problems that Rewrite-Shop3 could not repair. Of these, only 2 were due to timeouts, both on problem 6, showing that the algorithm usually could prove problems unrepairable. As before, SHOPFixer's more permissive definition of "repairable" meant that it solved more problems: it found only 35 unrepairable, and of these only 3 were due to timeouts, as with Rewrite-Shop3 these were both for problem 6. SHOPFixer's backjumping appeared to serve it well in the Rovers domain: IPyHOPPER had much more difficulty with these problems. It failed to repair 97 cases, of which 78 were due to timeouts. Time-

Problem	Success Rate
10	98%
19	98%
28	90%
29	92%
30	96%

Table 2: Openstacks problems where IPyHOPPER did not solve all of the problems.

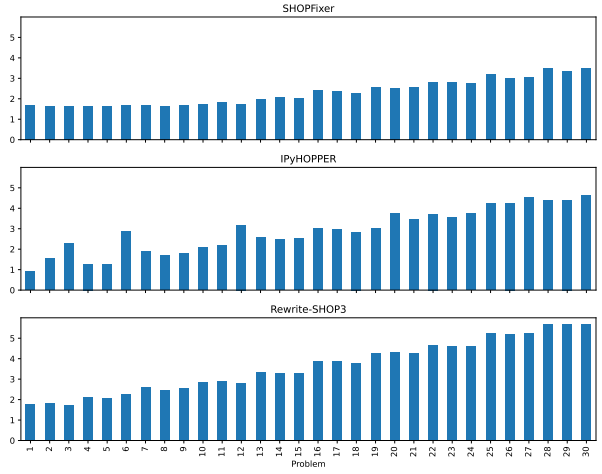


Figure 10: Runtimes for the Openstacks repair problems in $\log_{10}(\text{msec})$ for each algorithm. These include only those problems solved successfully by the algorithm in question.

outs are not a simple matter of scale: the greatest number of timeouts (by a factor of 2) is for problem 20, but the runners-up are 3, 6, and 10, in declining order of number of timeouts. Since the problems are intended to scale from first to last, the outcomes are not due only to raw scale.

Repair difficulties in the Rovers domain are due to the nature of the disturbances in our model. The "obstruct-visibility" disturbance can render the waypoint graph no longer fully connected, in terms of rover reachability. Losing a sample may also give rise to an unrepairable problem.

Openstacks The hardest of the domains, Openstacks shows the benefit of SHOPFixer's more expensive tree representation in runtime. We can see this even more clearly if we plot the two algorithms directly against each other (Figure 11). Indeed, IPyHOPPER's failures in this domain are all due to timeouts. Specific problems are indicated in Table 2. The more difficult search space here heavily penalizes IPyHOPPER's simple chronological backtracking.

In a reversal of the previous patterns, Rewrite solves all of the problems. This is due to a difference in the way the domain was formalized compared to the other two domains. Recall that we had to add a new action to prevent disturbances from making the Openstacks problems unsolvable. That modification had the effect of also helping Rewrite as did the fact that action choice is primarily constrained by preconditions, rather than by method structure, which also

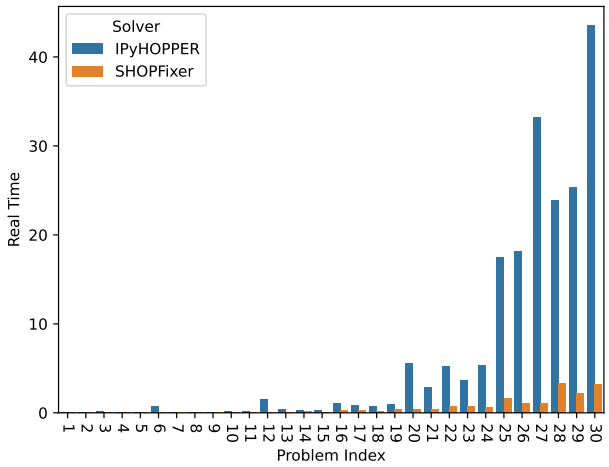


Figure 11: Direct comparison of runtimes for IPyHOPPER and SHOPFixer on Openstacks problems. Note that these are plotted in seconds, and not on a log scale.

avoided issues with this algorithm. Note that this relatively unconstrained planning also made it more difficult to generate the initial plans for this domain.

6 Conclusions and Future Directions

We have presented an analysis of three recent hierarchical repair algorithms from the AI planning literature; namely, SHOPFixer, IPyHOPPER, and Rewrite. Qualitatively, our analyses highlighted significant differences among these methods: First, Rewrite's definition of plan repair is more stringent than the others; we have identified benchmark planning problems that are solvable for IPyHOPPER and/or SHOPFixer that cannot be solved by Rewrite. Secondly, SHOPFixer attempts to detect when a plan will be invalid, before any actions actually fail; i.e., SHOPFixer invests in both data structures and computation in order to detect compromise to a plan as soon as possible. IPyHOPPER, on the other hand, is not a model-based projective planner in the same sense: it relies on an external simulation to do projection for it, instead of having an internal action model as most planners do. This difference in planning approaches leads to different plan repair behaviors.

Our results on the efficiency of the Rewrite algorithm should be taken with a large grain of salt. The original developers of this algorithm point out that their characterization is intended to be conceptually correct and clean, and that they have not yet taken into account the efficiency of the formulation. In addition to tuning the formulation, its efficiency could be improved by improved heuristics for planner when they run against rewrite problems. In particular, a planner searching the decomposition tree top-down should take into account the position of its leftmost child when deciding whether to choose the original methods, or methods whose leaves are taken from the executed prefix of the plan.

Our experiences also highlight unresolved issues in applying Rewrite in grounded planning systems. How best to schedule re-grounding *vis-à-vis* generation of the rewrit-

ten repair problem remains to be determined. While there were some subtleties to resolve in developing our lifted implementation, it did not have this chicken-and-egg problem.

Another interesting research direction is studying how HTN domain engineering affects the tradeoffs between efficiency and flexibility. At present, repair problems are generally created by modifying previously-existing planning problems (including IPC problems), which are not designed for execution, let alone to be repairable. In connection with the general concept of stability, this may yield a new insights in search-control for plan repair and for repairable plans; deriving properties from how preconditions and effects enable planning heuristics and repairability as well as how those preconditions and the task structure enable search control at higher levels of the plan trees. Similar to *refineability* properties (Bacchus and Yang 1992; Yang 1997), this approach can be examined formally, theoretically, and experimentally.

Acknowledgments. This project is sponsored by the Air Force Research Laboratory (AFRL) under contract FA8750-23-C-0515 for the HI-DE-HO STTR Phase 2 program. Distribution Statement A. Approved for public release: distribution is unlimited. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFRL. Thanks to the anonymous reviewers for their helpful feedback.

References

Ayan, F.; Kuter, U.; Yaman, F.; and Goldman, R. P. 2007. HOTRiDE: Hierarchical Ordered Task Replanning in Dynamic Environments. In *ICAPS-07 Workshop on Planning and Plan Execution for Real-World Systems*.

Bacchus, F.; and Yang, Q. 1992. The expected value of hierarchical problem-solving. In *AAAI*, 369–374. Citeseer.

Bansod, Y.; Patra, S.; Nau, D.; and Roberts, M. 2022. HTN Replanning from the Middle. *The International FLAIRS Conference Proceedings*, 35.

Bercher, P.; Biundo, S.; Geier, T.; Hoernle, T.; Nothdurft, F.; Richter, F.; and Schattenberg, B. 2014. Plan, Repair, Execute, Explain — How Planning Helps to Assemble Your Home Theater. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 24, 386–394.

Cushing, W.; and Kambhampati, S. 2005. Replanning: A New Perspective. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 13–16. Monterey, CA USA: AAAI Press.

Dechter, R. 2003. *Constraint Processing*. San Francisco: Morgan Kaufmann Publishers. ISBN 978-1-55860-890-0.

Fox, M.; Gerevini, A.; Long, D.; and Serina, I. 2006. Plan Stability: Replanning versus Plan Repair. In *ICAPS*.

Gaschnig, J. 1979. Performance Measurement and Analysis of Certain Search Algorithms. Technical Report CMU-CS-79-124, Carnegie-Mellon University. Reference for backjumping, cited in Ginsberg's texbook.

Gerevini, A.; Haslum, P.; Long, D.; Saetti, A.; and Dimopoulos, Y. 2009. Deterministic Planning in the Fifth International Planning Competition: PDDL3 and Experimental Evaluation of the Planners. *Artif. Intell.*, 173(5-6): 619–668.

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.

Hoffmann, J. 2011. Analyzing Search Topology without Running Any Search: On the Connection between Causal Graphs and h^+. *Journal of Artificial Intelligence Research*, 41: 155–229.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2021. The PANDA Framework for Hierarchical Planning. *KI - Künstliche Intelligenz*, 35.

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 2020)*. AAAI Press.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2020b. HTN Plan Repair via Model Transformation. In Schmid, U.; Klügl, F.; and Wolter, D., eds., *KI 2020: Advances in Artificial Intelligence*, volume 12325 of *Lecture Notes in Computer Science*, 88–101. Cham: Springer International Publishing. ISBN 978-3-030-58284-5 978-3-030-58285-2.

Kambhampati, S.; and Hendler, J. A. 1992. A Validation-Structure-Based Theory of Plan Modification and Reuse. *AIJ*, 55: 193–258.

Kuter, U. 2012. Dynamics of Behavior and Acting in Dynamic Environments: Forethought, Reaction, and Plan Repair. Technical Report 2012-1, SIFT.

Long, D.; and Fox, M. 2003. The 3rd International Planning Competition: Results and Analysis. *Journal of Artificial Intelligence Research*, 20: 1–59.

Robert P. Goldman; Ugur Kuter; and Richard G. Freedman. 2020. Stable Plan Repair for State-Space HTN Planning. In *HPlan 2020 Working Notes*. Nancy, France.

Yang, Q. 1997. Generating Abstraction Hierarchies. *Intelligent Planning: A Decomposition and Abstraction Based Approach*, 189–206.

Zaidins, P.; Roberts, M.; and Nau, D. 2023. Implicit Dependency Detection for HTN Plan Repair. In *Proceedings of the 6th ICAPS Workshop on Hierarchical Planning (HPlan 2023)*, 10–18.

An ILP Heuristic for Total-Order HTN Planning

Conny Olz[1], Alexander Lodemann[1], Pascal Bercher[2]

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

Abstract

Heuristic Search is still the most successful approach to hierarchical planning, both for finding any and for finding an optimal solution. Yet, there exist only a very small handful heuristics for HTN planning – so there is still huge potential for improvements. It is especially noteworthy that there does not exist a single heuristic that's tailored towards special cases. In this work we propose the very first specialized HTN heuristic, tailored towards totally ordered HTN problems. Our heuristic builds on an existing NP-complete and admissible delete-and-ordering relaxation ILP heuristic but partially incorporates ordering constraints while reducing the number of ILP constraints. It exploits inferred preconditions and effects of compound tasks and is also admissible. Our current heuristic proves to be more efficient than the one we build on, though it still performs worse than other existing (admissible) heuristics.

Introduction

As witnessed by already having the second track on Hierarchical Task Network (HTN) Planning in the International Planning Competition (IPC), solving HTN problems quickly or optimally is a prominent research field. Collectively, ten HTN planners participated at the IPCs (not counting various configurations per planner), and further planners exist as well. Among all the various approaches, heuristic progression search (Höller et al. 2020; Olz and Bercher 2023b) is still the most efficient approach, both for optimal and for suboptimal planning – as witnessed by the most recent IPC (Höller 2023a,b; Olz, Höller, and Bercher 2023).

The success of these search methods is tied directly to the quality of the heuristics deployed. Despite the success of heuristic search, only a very small number of HTN heuristics exist. One of the first was a TDG-based heuristic that estimates the minimal number of tasks that can be obtained by decomposing the compound tasks in the current search node (Bercher et al. 2017), one finds refinements to delete- and ordering-relaxed problems encoded by an Integer Linear Program (ILP) (Höller, Bercher, and Behnke 2020), another bases on landmarks (Höller and Bercher 2021), and the last – but most successful – heuristic is the relaxed composition heuristic (Höller et al. 2018, 2020), which encodes each search node into a classical problem allowing to deploy classical heuristics.

All of these heuristics are designed to cope with the most general case of an arbitrary partial order. However, there are significantly more specialized total-order (TO) planners than planners for general partial order planning, yet no heuristic for this important special case exists. It is however notable that there exists a pruning technique for total-order HTN problems (Olz and Bercher 2023b), which further shows the potential of this special case as its exploitation further improved the total-order planner on top of which the pruning was implemented and won the total-order HTN track of IPC 2023 (Olz, Höller, and Bercher 2023).

In this paper we propose the – to the best of our knowledge – first HTN planning heuristic tailored towards totally ordered problems. It exploits inferred preconditions and effects of compound tasks (Olz, Biundo, and Bercher 2021) (which also serve as the basis for the TO pruning technique (Olz and Bercher 2023b; Olz, Höller, and Bercher 2023)) and deploys them in a simplified variant of the ILP-based (NP-complete) delete- and ordering relaxation heuristic (DOF) by Höller, Bercher, and Behnke (2020). Like the original heuristic, our variant is admissible. Similarly, it can also be computed in polytime (by relaxing the integer variables to real-valued ones), but it naturally loses some of its informedness and hence pruning power if that is done.

We compare our new heuristic against the original ILP heuristic and the currently best-performing RC heuristic (Höller et al. 2020, 2018), using Add (Bonet and Geffner 2001) as the inner classical heuristic in satisficing planning and the RC heuristic with the admissible LM-cut (Helmert and Domshlak 2009) in optimal planning since our proposed heuristic is admissible. Our results show that while the ILP-based heuristics are not competitive with the RC(Add) heuristic[1], our new version generally outperforms DOF and also RC(lmc) in some domains in optimal planning

We hope that future work – further improvements to our technique – could bring our novel heuristic on par or even beat the RC(LM-cut) heuristic as well and discuss ideas for doing so at the end of the paper.

[1]When the DOF heuristic was published, it was on par with RC(Add) but a different (smaller) set of domains was used in the evaluation.

Theoretical Background

We start with providing the necessary definitions for total-order HTN planning and inferred effects of compound tasks.

HTN Planning Formalism

Our heuristic for totally ordered HTN planning is grounded in the formalisms introduced by Geier and Bercher (2011) and Behnke, Höller, and Biundo (2018). A total-order HTN planning domain is defined as a tuple $\mathcal{D} = (F, A, C, M)$, which includes a finite set of facts F, finite sets of *primitive tasks* A and *compound tasks* C (alternatively referred to as *abstract tasks*), and *decomposition methods* $M \subseteq C \times T^*$.[2] The collective set of tasks, both primitive and compound, is denoted by $T = A \cup C$. Actions or primitive tasks $a = (prec, add, del) \in A$ are characterized by their *preconditions* $prec(a) \subseteq F$ and their *effects* $add(a), del(a) \subseteq F$ (namely, the *add and delete effects*). An action $a \in A$ is *applicable* in a state $s \in 2^F$ if $prec(a) \subseteq s$. Upon application, it transitions the state s to a successor state $\delta(s, a) = (s \setminus del(a)) \cup add(a)$. This concept extends to action sequences $\bar{a} = \langle a_0 \dots a_n \rangle$ with each $a_i \in A$, deemed applicable in an initial state s_0 if a_0 is applicable in s_0 and sequentially for each $1 \leq i \leq n$, a_i is applicable in the resultant state $s_i = \delta(s_{i-1}, a_{i-1})$. Compound tasks within HTN planning, denoted by $c \in C$, represent a higher-level abstraction of primitive and/or compound tasks, further specified by decomposition methods $m = (c, \bar{t}) \in M$. These methods *decompose* a compound task c within a given task network $tn_1 = \langle \bar{t}_1\, c\, \bar{t}_2 \rangle \in T^*$ into another task network $tn_2 = \langle \bar{t}_1\, \bar{t}\, \bar{t}_2 \rangle$, denoted as $tn_1 \rightarrow_{c,m} tn_2$, where *task networks* are finite (possibly empty) sequences of tasks. A sequence of methods that transforms tn into tn' is represented as $tn \rightarrow tn'$, with tn' being called a refinement of tn. For a compound task $c \in C$ we also write $c \rightarrow tn'$ if we refer to $\langle c \rangle \rightarrow tn'$. A *TOHTN planning problem* $\Pi = (\mathcal{D}, s_I, c_I, g)$ is defined by the domain $\mathcal{D}$, an *initial state* $s_I \in 2^F$, an *initial task* $c_I \in C$, and a goal description $g \subseteq F$. A solution is a sequence of actions $tn = \langle a_0 \dots a_n \rangle \in A^*$ if $c_I \rightarrow tn$ holds, tn is applicable in s_I, and it leads to a goal state $s \supseteq g$.

Preconditions and Effects of Compound Tasks

In our version of the ILP heuristic we incorporate the concept of inferred negative effects of compound tasks as introduced by Olz, Biundo, and Bercher (2021). Compound tasks, both according to the formalism we base upon (Geier and Bercher 2011; Behnke, Höller, and Biundo 2018), and as described by HDDL (Höller et al. 2020), lack direct effects and serve primarily as placeholders for task networks to which they decompose into during the planning process. A detailed examination of the potential decompositions of compound tasks allows for the inference of state features required prior to any task refinement execution and the state features that result from all possible refinements. Olz, Biundo, and Bercher (2021) categorize such effects into several types, including possible and guaranteed effects, as well as positive and negative ones, in addition to preconditions. Our focus here is specifically on guaranteed negative effects, hence we limit our recap to them.

The set of *executability-enabling states* for a compound task $c \in C$ is defined as $E(c) = \{s \in 2^F \mid \exists\, \bar{a} \in A^* : c \rightarrow \bar{a}$ and $\bar{a}$ is applicable in $s\}$. Moreover, the set of states that could result from executing task c in state $s \in 2^F$ is $R_s(c) = \{s' \in 2^F \mid \exists\, \bar{a} \in A^* : c \rightarrow \bar{a}, \bar{a}$ is applicable in s and leads to $s'\}$.

Now, facts that are deleted after every successful execution of a refinement of a compound task c are called *state-independent negative effects* (cf. Def. 4, (Olz, Biundo, and Bercher 2021)) and are defined as follows:

$$\mathit{eff}^-_*(c) := \bigcap_{s \in E(c)} (F \setminus \bigcap_{s' \in R_s(c)} s')$$

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

Computing these "precise" effects for compound tasks is often too computationally expensive for the exploitation in heuristics, as it essentially requires solving certain planning problems (Olz, Biundo, and Bercher 2021). However, a more computationally feasible approach exists, based on *precondition-relaxation*. The *precondition-relaxed effects*, denoted as $\mathit{eff}^{\emptyset-}_*(c)$, are defined similarly to the original effects but rely on a precondition-relaxed version of the domain $\mathcal{D}' = (F, A', C, M)$, where $A' = \{(\emptyset, add, del) \mid (prec, add, del) \in A\}$. This approach considers only the presence and sequence of primitive tasks in the computation. Procedures for computing these effects in polynomial time have been provided by Olz, Biundo, and Bercher (2021) and are also implemented in the PANDADealer planning system (Olz and Bercher 2023b; Olz, Höller, and Bercher 2023), which we employ for our evaluation.

ILP Encoding for Delete-relaxed TO-HTN Search Nodes

Höller, Bercher, and Behnke (2020) introduced the first HTN planning heuristic based on an ILP. They encode a delete and ordering free (DOF) HTN planning problem, for which the plan existence problem is NP-complete to decide. The encoding can be divided into two parts: Constraints that ensure the successful execution of a sequence of actions (or a relaxed version of it) and constraints to ensure the proper decomposition leading to it. For the first part, Höller, Bercher, and Behnke use the encoding by Imai and Fukunaga (2015) (for classical planning) representing a delete-relaxed planning graph. Here, we introduce a different idea outlined further below. The latter part we take from the work by Höller, Bercher, and Behnke without changes, which we recap next.

Figure 1 outlines the set of ILP variables. The model by Höller, Bercher, and Behnke for the executability of actions needs five types of variables, but we could restrict our version to just one (the first one; the second and third are used to encode the decomposition). The objective function is the same, calculating the goal distance by minimizing the number of applied primitive actions and method applications:

$$min \sum_{a \in A} CA_a + \sum_{m \in M} M_m \qquad \text{(O)}$$

[2] The Kleene star notation T^* represents the set that includes the empty sequence and all finite sequences of tasks from T.

- $\{CA_t \mid t \in T\}$ (int) – value indicating how often a certain primitive or abstract task is in the solution.
- $\{M_m \mid m \in M\}$ (int) – value indicating how often a certain method is in the solution.
- $\{TNI_t \mid t \in T\}$ (bool) – flag indicating whether a certain task is the initial task.

Figure 1: The variable set used in our ILP model. It is a subset of the ones by Höller, Bercher, and Behnke (2020). The first variable is renamed, the last adapted to initial task instead of task network.

In order to simplify our constraints, we encode the current state s and goal description as actions, known from partial-order causal link (POCL) planning (McAllester and Rosenblitt 1991; Bercher 2021). Specifically, we introduce $a_I = (\emptyset, s, \emptyset)$ and $a_g = (g, \emptyset, \emptyset)$. Since they need to be ordered before and after, respectively, all other tasks, we additionally add a new compound task c_I with one method $m_I = (c_I, \langle a_I, tn_I, a_g \rangle)$, which replaces the current task network of the search node.

Task Decomposition Constraints

A solution to an HTN planning problem needs to be a refinement of the initial task. The following two constraints by Höller, Bercher, and Behnke (2020) encode this criterion. If a (primitive) task is contained in the solution, then it is the initial task and/or a method has been applied, which introduced the task into the plan:

Definition 1 (mst)**.** *Let $mst(t)$ be the multiset of methods where the task $t \in T$ is contained as a subtask. A method $m \in M$ is as often in $mst(t)$ as t is a subtask in m.*

$$\forall t \in T : CA_t = TNI_t + \sum_{m \in mst(t)} M_m \qquad \text{(C7)}$$

However, methods can not be applied arbitrarily, there also needs to be a suitable abstract task for every applied method.

Definition 2 ($mdec$)**.** *Let $mdec(c)$ be the set of methods decomposing the abstract task $c \in C$.*

$$\forall c \in C : CA_c = \sum_{m \in mdec(c)} M_m \qquad \text{(C8)}$$

According to Höller, Bercher, and Behnke (2020) the two constraints are sufficient to encode the proper decomposition (more precisely, encoding a so-called decomposition tree) leading to a sequence of tasks for acyclic problems. For cyclic domains further constraints are necessary to handle strongly connected components. This means that the encoding without those additional constraints can also be used for cyclic domains but the solutions can encode (shorter) plans that can not be obtained by a proper sequence of decompositions. To simplify the presentation of this paper, we do not repeat or use them since the evaluation results by Höller, Bercher, and Behnke do not show significant improvements by them.

Achiever Constraints

The ILP model by Höller, Bercher, and Behnke (2020) uses the constraints by Imai and Fukunaga (2015) to simulate a delete-relaxed planning graph. Therefore, for every time point there exist ILP variables for every action and fact, stating whether an action is executed at that time point and the facts being true or false. If an action is applied at some time point, its preconditions need to be true beforehand. If a fact needs to be true at some time point, there must be an action adding it if it is not already true in the initial state. The ILP solver tries to find an order of the actions so that preconditions and goal facts are satisfied (under delete relaxation). The resulting order might not meet the ordering constraints of tasks within methods. Thus, especially in total-order domains a lot of information gets lost.

Our intention was to improve the existing ILP heuristic in terms of accuracy by incorporating (at least some of) the ordering constraints imposed by the methods. We observed that adding additional constraints could improve the heuristic value but the additional time needed to solve the ILP did not pay off (this was done in a pre-evaluation, not reported here). To encode the planning graph, already quite a lot of variables and constraints are necessary. Therefore, in our proposed approach we made the model more simple by calculating necessary information outside of the ILP upfront. We ended up with only one (new) constraint (c_l is a large constant):

$$\forall a \in A, \forall f \in prec(a) : c_l \cdot \sum_{p \in achiever(f,a)} CA_p \geq CA_a \qquad \text{(C1)}$$

This constraint ensures that for every action $a \in A$ in the plan and every of its preconditions there is an action "achieving" the precondition. So, the influence of this constraint heavily depends on the definition of the set $achiever(f, a)$. A naive approach might be the following: Let $a \in A$ be a primitive task and $f \in F$ a precondition, then the set of possible achievers is $achiever(f, a) = \{a' \in A \mid f \in add(a')\}$. However, in this case the solutions of the ILP are not very restricted. Neither are the methods' ordering constraints taken into account nor does it guarantee that there is an executable ordering of the actions (even under delete relaxation). In the next section we present algorithms to restrict the set of achievers for an action a further so that it only contains actions that appear before a according to the method's total order. Moreover, by exploiting the inferred effects, we can even partially consider delete effects. Thus, we do not present further changes to the ILP model, we only discuss several options of how to calculate the set of possible achievers and their impact on the set of ILP solutions.

Determining Achiever Actions

Given a TOHTN planning problem, we can determine for a given action the set of actions that can be ordered before that action in a possible refinement of the initial task. We will see that we can calculate this with different levels of accuracy.

To start we define the set of reachable actions $reachable(c) = \{a \in A \mid \exists\, \bar{t} \in T^* : c \to \bar{t} \text{ and } a \in \bar{t}\}$

Algorithm 1: Calculating Predecessor Actions

Input: A problem $\Pi = (\mathcal{D}, s_I, c_I, g)$
Output: Sets of possible predecessors $pred(a) \, \forall \, a \in A$

1: $pred(a) = \emptyset$ for all $a \in A$
2: **for all** methods $m = (c, \langle t_0, \dots, t_n \rangle) \in M$ **do**
3: **for** $i = 1$ to n **do**
4: **for all** $a \in reachable(t_i)$ **do**
5: **for** $j = i - 1$ to 0 **do**
6: $pred(a) = pred(a) \cup reachable(t_j)$

of a compound tasks $c \in C$, which are the actions reachable via decomposition. For primitive actions $a \in A$, we define $reachable(a) = \{a\}$. The sets can be calculated in polynomial time, e.g., by a depth-first search with a closed list of already visited compound tasks to prevent infinite cycles. The RC heuristic does this in a preprocessing step; it's a one-time computation. Given that set for every compound task, we can compute for every primitive action the set of actions that can possibly appear as predecessor in a refinement of the initial compound task (and are thus candidates for achiever actions) as shown in Algorithm 1.

The algorithm considers every method once. So let $m = (c, \langle t_0, \dots, t_n \rangle) \in M$ be a method. For each task t_i $(i > 1)$ in the method, all of its reachable actions are considered. All reachable actions of preceding tasks $t_j, j < i$ are added to their sets of predecessors.

Proposition 1. *Let $\Pi = (\mathcal{D}, s_I, c_I, g)$ be an total-order HTN planning problem, $a \in A$, and $pred(a)$ be computed by Algorithm 1. Then it holds $a' \in pred(a)$ if and only if there exists a refinement $\bar{a}$ of c_I (not necessarily executable) so that $a, a' \in \bar{a}$ and $a' \prec a$.*

Proof Sketch. We assume that all methods are reachable by decomposition from the initial task, otherwise the unconnected methods should not be considered in the algorithm.

"⇒" Let $a, a' \in A$ and $a' \in pred(a)$. Consider the method $m = (c, \langle t_0, \dots, t_n \rangle) \in M$ in line 2 for which a' was added to $pred(a)$ in line 6. Since there there are $0 \leq j, i \leq n$ with $j < i$, $a \in reachable(t_i)$, and $a' \in reachable(t_j)$ there must be a refinement of $\langle t_0, \dots, t_n \rangle$ in which a' is ordered before a. Moreover, by assumption, there must be a refinement of c_I that contains c which can be decomposed using m, which proves the first direction.

"⇐" Let $\bar{a}$ be a refinement of c_I and $a, a' \in \bar{a}$ two primitive tasks with $a' \prec a$. If we consider the two sequences of decompositions (more specifically, the used methods) leading from c_I to a and from c_I to a' in $\bar{a}$, then the two sequences have the same prefix of methods. The suffix may differ. However, the last common method $m = (c, \langle t_0, \dots, t_n \rangle) \in M$ must have two tasks $t_j \prec t_i$ with $a' \in reachable(t_j)$ and $a \in reachable(t_i)$ so that a' will get added to $pred(a)$ in line 6. □

Algorithm 1 can be extended to restrict the set of possible achievers for the preconditions of an action $achiever(f, a)$ based on the total-order of the method set and inferred negative effects, given in Algorithm 2.

Algorithm 2: Calculating Achiever Actions

Input: A problem $\Pi = (\mathcal{D}, s_I, c_I, g)$
Output: $achiever(f, a)$ for all $a \in A, f \in prec(a)$

1: $achiever(f, a) = \emptyset$ for all $a \in A, f \in prec(a)$
2: **for all** methods $m = (c, \langle t_0, \dots, t_n \rangle) \in M$ **do**
3: **for** $i = 1$ to n **do**
4: **for all** $a \in reachable(t_i)$ **do**
5: **for all** $p \in prec(a)$ **do**
6: **for** $j = i - 1$ to 0 **do**
7: **if** $p \in eff_*^-(t_j)/del(t_j)$ **then**
8: **break**
9: $achiever(a, p) = achiever(a, p) \cup \{a' \in reachable(t_j) \mid p \in add(a')\}$

Again, every method $m = (c, \langle t_0, \dots, t_n \rangle) \in M$ is considered once. Now, for each task t_i $(i > 1)$, all of its reachable actions and their preconditions are considered. Preceding tasks $t_j, j < i$ are checked for reachable actions that can add these preconditions, updating the achiever sets. If some preceding task $t_k, k < i$ deletes a precondition (according to its (inferred) delete effects), we do not consider its reachable actions nor the reachable actions of its predecessors (line 8).

Algorithm 2 runs in polynomial time in $\mathcal{O}(|M| \cdot n^2 \cdot (reach_{max})^2 \cdot prec_{max})$, where n is the size of the largest task network within methods, $reach_{max} = \max_{t \in T} |reachable(t)|$ and $prec_{max} = \max_{a \in A} |prec(a)|$.

Proposition 2. *Let $\Pi = (\mathcal{D}, s_I, c_I, g)$ be a problem, $a \in A$ a primitive task, and $achiever(a, p)$, $pred(a)$ be calculated according to Algorithms 1 and 2. Then it holds*

- $\bigcup_{p \in prec(a)} achiever(a, p) \subseteq pred(a)$ *and*
- *for all refinements $\bar{a}$ of c_I it holds if $a, a' \in \bar{a}, a' \prec a$, $p \in prec(a) \cap add(a')$ and p is not deleted in between then $a' \in achiever(a, p)$.*

Proof Sketch. Since Algorithm 2 collects only actions that add one of the preconditions, the set of achievers is a subset of the predecessors calculated by Algorithm 1. Since for the achievers some of the delete effects are taken into account in line 8 even less actions are added.

For the second point, let us consider a refinement $\bar{a}$ of c_I with $a, a' \in \bar{a}, a' \prec a$, $p \in prec(a) \cap add(a')$, where no other action deletes p in between. According to Proposition 1 we know that $a' \in pred(a)$. Since no action in between deletes p, the condition for line 8 is not satisfied. So, a' is also added to $achiever(a, p)$ since $p \in prec(a)$ and $p \in add(a')$. □

We can now define our first version of our heuristic based on the ILP presented in the last section with the set of possible achiever $achiever(a, p)$ calculated by Algorithm 2, which we denote h^{TOILP}.

Theorem 1. *For every solution of a TOHTN planning problem there exists a valid assignment of the ILP model.*

Proof. Höller, Bercher, and Behnke (2020) already showed for every solution of a DOF HTN planning problem, there

is a valid assignment of their ILP model. Since every solution of a TOHTN planning problem is also a solution under delete and ordering relaxation, we know that there is a valid assignment of our model that satisfies the task decomposition constraints C7 and C8. So we only need to check C1. For all primitive tasks $t \in A$ the variables CA_t are set according to how often the task is in the solution. Consider a primitive task a with $CA_a > 0$ and precondition f. Since the plan is executable there must be an action a' in the plan adding f and no action deletes f in between. According to Proposition 2 it holds $a' \in achiever(a, f)$ and therefore C1 is satisfied since also $CA_{a'} > 0$. □

The next question to ask is whether h^{TOILP} performs some relaxations or whether all valid assignments of the ILP model correspond to some solution of an TOHTN problem. From a theoretical point of view this is "unlikely" (or even impossible, depending on the exact relationship of complexity classes, which are still unknown) since TOHTN planning is EXPTIME-complete in general and still PSPACE-complete for acyclic domains (Alford, Bercher, and Aha 2015). ILPs can only solve problems in NP. Basically two relaxations are performed. Since we only check for achievers of preconditions and ignore most of the delete effects we perform some delete-relaxation. Moreover, the total-order of tasks is partially relaxed. Consider for example a method containing a compound task c twice. Assume that c has two methods $m_1 = (c, \langle a_1 \rangle)$ and $m_2 = (c, \langle a_2 \rangle)$, where a_1 can support a precondition of a_2 and vice versa. Then the two primitive actions are in the achiever sets of each other and in a solution of a corresponding ILP the two actions could support each other. However, actually only one of the preconditions is satisfied because one action is applied before the other, so the first one needs another action adding its precondition.

To overcome this limitation one could duplicate primitive and compound tasks so that every task occurs just once over all methods task networks. So in the example above, we then have two compound tasks c and c', m_1 and m_2 unchanged but two additional methods $m'_1 = (c', \langle a'_1 \rangle)$ and $m'_2 = (c', \langle a'_2 \rangle)$, where a'_1 and a'_2 have the same preconditions and effects as a_1 and a_2, respectively. In the worst case such a transformation introduces exponential many new tasks. If a transformation is possible with polynomial many new tasks, we can actually encode an acyclic, delete-relaxed TOHTN problem. Therefore, we call a TOHTN planning problem $\Pi = (\mathcal{D}, s_I, c_I, g)$ a *unique tasks* problem if for all tasks $t \in A \cup C$ there is at most one method $m = (c, \bar{t}) \in M$ with $t \in \bar{t}$ and additionally t is contained only once in $\bar{t}$.

Theorem 2. *Consider an acyclic, delete-relaxed total-order HTN planning problem with unique tasks. Then, for every valid assignment of the ILP model, there exists a corresponding solution to the underlying TOHTN problem.*

Proof Sketch. Consider an acyclic, delete-relaxed, unique tasks TOHTN problem and a valid assignment of the corresponding ILP model. According to Höller, Bercher, and Behnke (2020) the constraints C7 and C8 ensure that there is a refinement of the initial task that contains exactly the number of primitive tasks as indicated by the variables CA_t. Since the constraints C1 are satisfied, for all actions in the plan and their preconditions there is an action adding it. So we only need to verify that the actions appear in the correct order so that all preconditions are satisfied. Since every task (primitive or compound) appears exactly once in all methods there is only one sequence of decompositions leading to that task. This implies that if for two actions a, a' it holds $a' \in pred(a)$ then $a \notin pred(a')$. This does also hold for the achiever sets since they are subset of the predecessors. So the achievers already encode some total-order over all tasks and the refinement of the initial task is executable under delete-relaxation. □

Since we take some of the (inferred) delete effects into account when we calculate the achievers (line 8, Alg. 2) not every solution of an acyclic, delete-relaxed, unique tasks TOHTN problem has a valid assignment of the ILP model. Some of the non-executable ones (when considering delete effects) are missing, which is beneficial for the heuristic since they are recognizes as not being correct solutions.

If we remove line 8 (what we should not do in practice) the ILP can exactly encode acyclic, delete-relaxed, unique tasks TOHTN problems, which is the first encoding for this class so far. This is also in line with the result by Alford et al. (2014) that acyclic and delete-relaxed (t.o.) HTN planning is NP-complete. The result by Alford et al. actually tells us that there must be an encoding in general without relying on the unique tasks property. The hardness proof by Alford et al. does not rely on unique tasks but we can adapt a reduction by Olz and Bercher (2023a) to show NP-hardness of acyclic and regular, delete- and precondition-relaxed problems to unique tasks so that we can conclude that our studied problem is also already NP-hard (and complete).

For our situation now, it needs to be evaluated empirically whether the transformation to unique tasks pays off but a preliminary evaluation showed no positive effect. The heuristic can be used nevertheless, it just relaxes the problem a bit more in case of non-unique tasks as discussed above.

Admissibility

We saw that for every solution of a TOHTN planning problem there exist a valid assignment of the ILP model, so the heuristic is safe. The objective function of the ILP minimizes the number of primitive actions and methods that need to be applied. So it estimates the distance to the goal and not the length of a minimal plan. However that can easily be changed by setting the objective function to just minimizing the number of primitive tasks. Then the heuristic value is bounded from above by the length of an optimal plan, which makes it admissible. The artificial actions encoding the initial state and goal should be excluded in that function.

Corollary 1. *The heuristic is admissible, goal aware, and safe with the objective function* $min \sum_{a \in A} CA_a$.

Evaluation

We evaluated our proposed heuristic in satisficing planning, where one tries to find a solution as fast as possible within

a given time limit, as well as optimal planning. Therefore, we integrated the heuristic into the progression-based version of the PANDA$_\pi$ system[3][4] (Höller et al. 2020). We used the currently best-performing configuration according to the last IPC in 2023, which is GBFS (and A* for optimal planning, respectively) with loop detection (Höller and Behnke 2021) and dead-end analysis with look-aheads and early refinements (Dealer) (Olz and Bercher 2023b; Olz, Höller, and Bercher 2023). For completeness reasons we also included the results without the latter, though below we focus our report on results with Dealer since those yield overall better results.

We compared our heuristic, called TOILP, against the ILP-based heuristic DOF by Höller, Bercher, and Behnke (2020) and the currently best-performing heuristics. For satisficing planning, this is the Relaxed Composition (RC) heuristic (Höller et al. 2020) in combination with the classical Add heuristic (Bonet and Geffner 2001) as RC(Add), and for optimal planning, RC combined with LM-cut (Helmert and Domshlak 2009) as RC(lmc), which is the only other existing admissible HTN planning heuristic.

We run the evaluation on a machine with a Xeon E5-2660 v3 with 2.60GHz and 40 CPUs. As a benchmark set, we used all problems of the 24 domains in the benchmark set of the IPC 2020[5]. Each planning problem was granted one core, a maximum of 8 GiB RAM, and a time limit of 1800 seconds.

Satisficing Planning

In Table 1 we report results for satisficing planning: The number of solved instances within the time and memory limits (coverage), normalized coverage, where equal significance is assigned to all domains, ensuring that domains with a multitude of instances do not overshadow those with fewer instances, and the IPC score, which is computed by $min\{1, 1 - log(t)/log(1800)\}$, where t is the time required to solve the problem in seconds. It rewards solving problems quickly.

The RC(Add) heuristic outperforms both other heuristics over all in terms of solved instances (744 versus 496 and 415, respectively) and IPC score (15.32 versus 8.21 and 7.02, respectively). When the DOF heuristic was published it was on par with the the RC(Add) heuristic on the benchmark set of that time. It is interesting to see that this picture changed completely with the new domains. There is not a single domain in which the DOF solves more instances than RC(Add). The TOILP heuristic can at least outperform RC(Add) in two domains, namely Depots and Transport, with 2 and 3 more solved problems, respectively.

Our main intention was to improve the DOF heuristic so we compare it with the TOILP next. We observe that TOILP could solve 19.5% more problems and the IPC score was improved by around 17%. Looking at individual domains, we can see that in the Logistics-Learned, Multiarm-Blocksworld, and Towers domains, TOILP solved around twice as many problems as DOF. In the Hiking domain,

[3]http://panda.hierarchical-task.net

[4]https://github.com/ipc2023-htn/PandaDealer

[5]https://ipc2020.hierarchical-task.net/

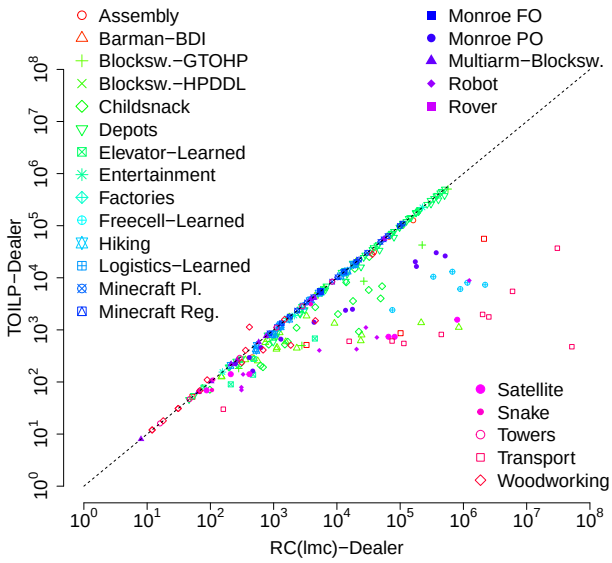


Figure 2: Number of generated search nodes for optimal planning. Be aware of the log scale.

TOILP solved 10 problems where DOF was only able to solve 1 problem. So we can conclude that, indeed, overall the TOILP performs better than the DOF heuristic on total-order domains.

We evaluated the satisficing setting for reasons of completeness. Given that non-admissible heuristics often outperform admissible ones here, it is not too surprising that TOILP does not emerge as the best overall.

Optimal Planning

Given the limited number of admissible HTN planning heuristics available to date the TOILP heuristic has potential in optimal planning. As indicated in Table 2, the performance gap between the RC heuristic and ILP-based heuristics is indeed narrower in optimal planning than in satisficing planning. Although the RC(lmc) heuristic still surpasses the others in terms of coverage, with scores of 360 compared to 302 and 261, and in IPC scores, with 6.93 versus 5.56 and 4.71, respectively, the TOILP heuristic demonstrates superior performance in several domains. Notably, in 13 out of 24 domains, TOILP matched or exceeded the performance of RC(lmc). The Childsnack domain stands out, where the RC heuristic failed to solve any problems, whereas TOILP and DOF solved 5.

The RC(lmc) heuristic runs in polynomial time, while the ILP-based heuristics are capable of encoding and solving NP-hard problems. From a theoretical perspective, this suggests that the ILP-based heuristics are more informed at the cost of increased computation time. While with greedy search such additional runtime often does not pay off since we just want to find any solution, in optimal planning the enhanced information can prove valuable in narrowing down the search space. This is supported by the data in Figure 2, which shows that TOILP generally requires fewer search nodes than RC(lmc) for most problems, with significant dif-

domain		RC(Add)-Dealer		RC(Add)		TOILP-Dealer		TOILP		DOF-Dealer		DOF	
		Coverage	IPC	Coverage	IPC	Coverage	IPC	Coverage	IPC	Coverage	IPC	Coverage	IPC
Assembly	30	**30**	**0.89**	**30**	0.88	29	0.42	25	0.36	24	0.33	20	0.28
Barman-BDI	20	**16**	**0.75**	**16**	0.68	12	0.36	10	0.30	14	0.38	11	0.29
Blocksw.-GTOHP	30	**30**	0.86	**30**	**0.94**	25	0.75	25	0.71	27	0.76	25	0.68
Blocksw.-HPDDL	30	**30**	**0.76**	26	0.68	21	0.39	20	0.28	20	0.33	19	0.26
Childsnack	30	**23**	0.64	**23**	**0.65**	18	0.24	18	0.24	17	0.19	17	0.19
Depots	30	22	**0.73**	22	**0.73**	**24**	0.70	**24**	0.67	22	0.52	21	0.48
Elevator-Learned	147	**147**	**0.75**	**147**	0.60	124	0.43	108	0.35	107	0.35	95	0.30
Entertainment	12	**12**	0.94	**12**	**0.95**	**12**	0.87	**12**	0.86	**12**	0.81	**12**	0.80
Factories	20	**11**	**0.37**	8	0.32	6	0.22	5	0.17	6	0.20	4	0.15
Freecell-Learned	60	16	**0.10**	**18**	0.06	0	0.00	0	0.00	0	0.00	0	0.00
Hiking	30	**25**	0.67	**25**	**0.68**	10	0.16	10	0.22	1	0.01	1	0.01
Logistics-Learned	80	**80**	**0.79**	48	0.45	47	0.33	46	0.26	22	0.24	22	0.20
Minecraft Pl.	20	**4**	**0.08**	**4**	**0.08**	0	0.00	0	0.00	0	0.00	0	0.00
Minecraft Reg.	59	**42**	0.58	**42**	**0.59**	40	0.47	40	0.46	40	0.43	40	0.42
Monroe FO	20	**20**	**0.50**	**20**	**0.50**	0	0.00	0	0.00	0	0.00	0	0.00
Monroe PO	20	**13**	**0.28**	11	0.24	0	0.00	0	0.00	0	0.00	0	0.00
Multiarm-Blocksw.	74	**74**	**0.89**	**74**	0.80	35	0.18	14	0.10	16	0.11	12	0.08
Robot	20	**20**	**0.93**	**20**	0.91	**20**	0.75	11	0.53	**20**	0.76	**20**	0.74
Rover	30	27	0.57	**29**	**0.65**	9	0.24	9	0.24	9	0.23	9	0.23
Satellite	20	**19**	0.66	**19**	**0.68**	10	0.35	10	0.35	10	0.31	10	0.31
Snake	20	**20**	**0.91**	**20**	0.90	4	0.12	3	0.09	3	0.07	3	0.02
Towers	20	**13**	**0.49**	**13**	0.46	5	0.19	6	0.20	3	0.12	3	0.11
Transport	40	22	0.54	25	**0.58**	**28**	**0.58**	**28**	0.57	25	0.45	25	0.42
Woodworking	30	**28**	**0.66**	27	0.64	17	0.45	16	0.39	17	0.40	17	0.39
Overall	892	**744**	**15.32**	709	14.63	496	8.21	440	7.34	415	7.02	386	6.33
Normalized Coverage		**19.50**		18.85		12.23		11.03		10.89		10.24	

Table 1: Coverage and IPC score for satisficing planning

ferences observed in several cases. This indicates that the TOILP heuristic provides more precise heuristic values, although the computation time remains slightly too high, resulting in overall superior performance by RC(lmc).

Despite having considerably fewer variables and constraints than the DOF heuristic, the TOILP heuristic's performance, in terms of calculated search nodes, is comparable to that of DOF, as illustrated in Figure 3, only a few problems need more search nodes. This suggests that TOILP's unique constraint ensuring executability, coupled with the precalculated task ordering, delivers results of similar quality to those of the DOF constraints but faster.

Future Work

We already discussed some future work, which we briefly recap now together with further ideas:

- The constraints for strongly connected components, as proposed by Höller, Bercher, and Behnke (2020) in their ILP model, could be incorporated to better handle cyclic domains.
- We can introduce new tasks so that every task appears just once over all methods, which makes the calculation of achiever actions more precise. However, it also increases the model, which might slow down the ILP solver significantly.

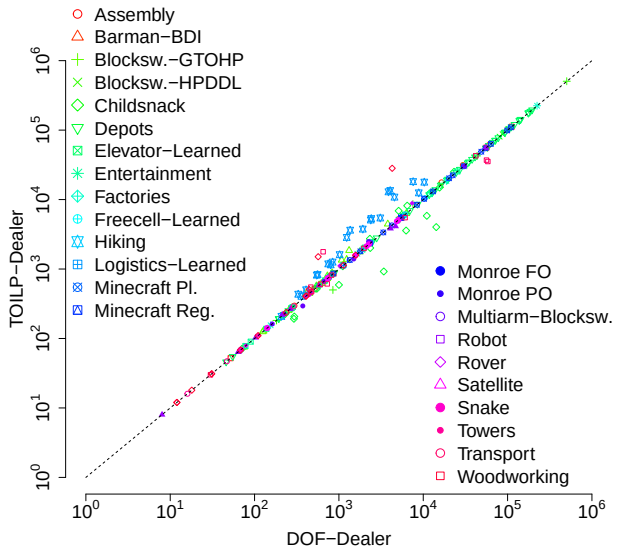


Figure 3: Number of generated search nodes for optimal planning. Be aware of the log scale.

domain		RC(lmc)-Dealer		RC(lmc)		TOILP-Dealer		TOILP		DOF-Dealer		DOF	
		Coverage	IPC	Coverage	IPC	Coverage	IPC	Coverage	IPC	Coverage	IPC	Coverage	IPC
Assembly	30	4	0.11	4	0.10	**5**	**0.12**	4	0.10	**5**	0.10	4	0.09
Barman-BDI	20	**10**	**0.32**	**10**	0.29	6	0.14	5	0.11	5	0.11	5	0.08
Blocksw.-GTOHP	30	**26**	0.68	23	0.61	25	**0.73**	25	0.66	24	0.71	23	0.62
Blocksw.-HPDDL	30	**5**	**0.12**	**5**	0.11	**5**	0.11	4	0.09	**5**	0.10	4	0.09
Childsnack	30	0	0.00	0	0.00	**5**	**0.05**	4	0.03	**5**	0.03	3	0.01
Depots	30	18	**0.55**	18	0.52	**19**	0.51	**19**	0.48	18	0.38	18	0.33
Elevator-Learned	147	92	0.33	**112**	**0.39**	67	0.24	64	0.22	55	0.20	53	0.19
Entertainment	12	5	0.42	5	0.42	**9**	**0.58**	**9**	0.57	8	0.53	8	0.52
Factories	20	**6**	**0.23**	5	0.20	5	0.20	4	0.15	5	0.18	4	0.14
Freecell-Learned	60	**0**	**0.00**	**0**	**0.00**	**0**	**0.00**	**0**	**0.00**	**0**	**0.00**	**0**	**0.00**
Hiking	30	6	0.05	6	0.05	**12**	**0.14**	5	0.04	3	0.03	2	0.02
Logistics-Learned	80	**27**	**0.25**	**27**	**0.25**	22	0.20	22	0.18	22	0.20	22	0.17
Minecraft Pl.	20	**2**	**0.03**	1	0.02	1	0.01	0	0.00	1	0.00	0	0.00
Minecraft Reg.	59	33	0.33	33	0.33	**38**	**0.35**	**38**	**0.35**	33	0.27	33	0.27
Monroe FO	20	**19**	**0.37**	12	0.23	0	0.00	0	0.00	0	0.00	0	0.00
Monroe PO	20	**10**	**0.15**	7	0.14	0	0.00	0	0.00	0	0.00	0	0.00
Multiarm-Blocksw.	74	12	**0.13**	12	0.12	**13**	0.11	8	0.08	9	0.08	8	0.07
Robot	20	**11**	**0.55**	**11**	**0.55**	**11**	0.51	**11**	0.50	**11**	0.51	**11**	0.51
Rover	30	**8**	**0.22**	**8**	0.21	**8**	0.21	**8**	0.21	**8**	0.20	**8**	0.20
Satellite	20	6	0.21	6	0.21	**10**	**0.34**	**10**	0.33	**10**	0.29	**10**	0.28
Snake	20	**20**	**0.76**	**20**	0.68	4	0.13	4	0.10	3	0.07	3	0.02
Towers	20	**13**	**0.48**	12	0.45	6	0.19	6	0.19	3	0.10	3	0.10
Transport	40	10	0.15	9	0.14	**15**	**0.28**	14	0.27	13	0.24	13	0.19
Woodworking	30	**17**	**0.51**	16	0.50	16	0.40	13	0.34	15	0.37	15	0.35
Overall	892	360	**6.93**	**362**	6.51	302	5.56	277	5.00	261	4.71	250	4.25
Normalized Coverage		**10.00**		9.33		7.99		7.30		6.99		6.66	

Table 2: Coverage and IPC score for optimal planning

- Currently we calculate the set of achiever tasks once in the beginning, but one could update the sets according to the reachable tasks of the current search node. Since some methods may not be reachable anymore, the sets can become smaller and more accurate. Whether this additional computation time pays off is an open question.
- The ILP (Integer Linear Program) can be relaxed to a LP, where the variables can be assigned real numbers instead of integers. LPs are solvable in polynomial time but the solution might not correspond to a plan anymore. Since the objective value of an LP is always less or equal to the one of the corresponding ILP the heuristic is still admissible. It needs to be evaluated whether the speedup of calculation time can compensate the loss of information. In the original work by Höller, Bercher, and Behnke (2020) the ILP version was slightly better.

Conclusion

We proposed a novel ILP-based HTN planning heuristic tailored to total-order domains. The ordering information is calculated in advance and integrated into the ILP model so that the number of constraints can be significantly reduced compared to an existing ILP heuristic, which ignores the ordering completely. Empirical results indicate that the new heuristic outperforms the original one clearly, dominating it in terms of solved instances (coverage) and IPC score on every existing total-order domain. When comparing our NP-complete heuristic against the currently best performing admissible HTN heuristic that can be computed in polynomial time, RC(lmc), results are mixed. When looking at the overall sum of solved instances and IPC score, RC(lmc) performs better than the one proposed. However, neither dominates the other. This shows the higher informedness of our proposed heuristic pays out in several domains, but it too costly in several others. It will be interesting to see how the proposed heuristic performs with further progress on the research question on how to compute inferred effects of compound tasks. The informed of the heuristic depends on the amount of effects computed, so the heuristic will *automatically* become more informed when more preconditions and effects can be identified in the preprocessing step that this heuristic and the Dealer technique depend upon.

Acknowledgements

Pascal Bercher is the recipient of an Australian Research Council (ARC) Discovery Early Career Researcher Award (DECRA), project number DE240101245, funded by the Australian Government.

References

Alford, R.; Bercher, P.; and Aha, D. W. 2015. Tight Bounds for HTN Planning. In *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS 2015)*, 7–15. AAAI Press.

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

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

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 31st International Conference on Automated Planning and Scheduling (ICAPS 2021)*, 36–45. AAAI Press.

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

Bonet, B.; and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence*, 129(1-2): 5–33.

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

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 162–169. AAAI Press.

Höller, D. 2023a. The PANDA λ System for HTN Planning in the 2023 IPC. In *Proceedings of the 11th International Planning Competition: Planner Abstracts – Hierarchical Task Network (HTN) Planning Track, IPC 2023*.

Höller, D. 2023b. The PANDA Progression System for HTN Planning in the 2023 IPC. In *Proceedings of the 11th International Planning Competition: Planner Abstracts – Hierarchical Task Network (HTN) Planning Track, IPC 2023*.

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

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

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

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

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2018. A Generic Method to Guide HTN Progression Search with Classical Heuristics. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS 2018)*, 114–122. AAAI Press.

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

Imai, T.; and Fukunaga, A. 2015. On a Practical, Integer-Linear Programming Model for Delete-Free Tasks and its Use as a Heuristic for Cost-Optimal Planning. *Journal of Artificial Intelligence Research (JAIR)*, 54: 631–677.

McAllester, D. A.; and Rosenblitt, D. 1991. Systematic Nonlinear Planning. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI 1991)*, 634–639. AAAI Press.

Olz, C.; and Bercher, P. 2023a. Can They Come Together? A Computational Complexity Analysis of Conjunctive Possible Effects of Compound HTN Planning Tasks. In *Proceedings of the 33rd International Conference on Automated Planning and Scheduling (ICAPS 2023)*, 314–323. AAAI Press.

Olz, C.; and Bercher, P. 2023b. A Look-Ahead Technique for Search-Based HTN Planning: Reducing the Branching Factor by Identifying Inevitable Task Refinements. In *Proceedings of the 16th International Symposium on Combinatorial Search (SoCS 2023)*, 65–73. AAAI Press.

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

Olz, C.; Höller, D.; and Bercher, P. 2023. The PANDADealer System for Totally Ordered HTN Planning in the 2023 IPC. In *Proceedings of the 11th International Planning Competition: Planner Abstracts – Hierarchical Task Network (HTN) Planning Track (IPC)*.

Barely Decidable Fragments of HTN Planning

P. Maurice Dekker,
Gregor Behnke

University of Amsterdam
p.m.dekker@uva.nl, g.behnke@uva.nl

Abstract

Unrestricted HTN planning is undecidable. However some fragments of HTN planning – such as totally-ordered or tail-recursive HTNs – are decidable. Studying these restricted fragments has lead to valuable insights, which ultimately gave rise to the development of new efficient HTN planning algorithms.

We identify new decidable fragments of HTN planning. In a *one-hole-digging* problem, every task network contains at most a single compound task. In an *initial* problem, compound tasks are order-minimal in all task networks, while in a *final* problem, they are order-maximal. We precisely determine the complexity of these fragments – they are Ackermann-complete. This remains true even under the restriction that there is only one compound task and only two methods for it.

Introduction

Hierarchical Task Network (HTN) planning (Sacerdoti 1975; Erol, Hendler, and Nau 1994) is an expressive formalism for planning. It allows for describing the physics of the domain in terms of both the preconditions and effects of actions but also allows for specifying a grammar-like refinement-structure that valid plans must follow. HTN planning has over the past decades been studied both from theoretical and practical views. In several cases, theoretical insights have informed practical planning methods and have lead to new algorithms. This includes for example Erol's insight that totally-ordered HTN planning is decidable (Erol, Hendler, and Nau 1996), which has lead to dedicated total-order HTN planners (Nau et al. 1999; Magnaguagno, Meneguzzi, and de Silva 2021; Schreiber 2021; Behnke, Höller, and Biundo 2018; Alford et al. 2016; Alford, Kuter, and Nau 2009), but also the complexity analysis of tail-recursive HTN problems (Alford, Bercher, and Aha 2015) and the idea of relating HTN planning to formal languages (Höller et al. 2014), which have lead to dedicated planning algorithms (Alford et al. (2016) and Höller (2021) respectively).

These innovations were, at least to some degree, driven by researchers wanting to better understand the complexity-landscape of HTN planning. As HTN planning in general is undecidable (Erol, Hendler, and Nau 1996), this at least includes identifying expressive but still decidable fragments of HTN planning. This had lead to discovering, e.g., totally-ordered and tail-recursive HTN problems.

We present a new class of decidable HTN planning problems: that of *one-hole-digging* problems. This class is orthogonal to the previously investigated classes and thus illuminates a new island of decidability. While being decidable, one-hole-digging problems are only barely decidable: we show that they are complete for the class of $\mathscr{F}_\omega$, i.e., for all problems whose runtime is limited by an Ackermann-function. Our completeness proof also goes through for other fragments of HTN planning, most notably *initial*, *final*, and *clean* problems. See Table 3 at the end of the paper.

Preliminaries

In order to present the new fragments and results, we first formally define the notion of HTN planning. We then introduce Petri nets, which we will use in our reduction, and lastly introduce the required concepts of computational complexity theory.

HTN Planning

In this section, we set up the HTN formalism following Geier and Bercher (2011). This is a simple way of adding hierarchy to the STRIPS formalism, and is hence comfortable for proving complexity results.

A *task network* over a set N of *task names* is a triple $\mathfrak{t} = (T, \prec, \alpha)$ where

- T is a finite set of *tasks*;
- $\prec$ is a strict partial order on T;
- $\alpha : T \to N$ assigns a task name to each task in the network.

Let $\mathfrak{T}_N$ be the set of all task networks over N. If $T' \subseteq T$, we define the *task subnetwork*

$$\mathfrak{t}|T' = \left(T', \prec \cap (T' \times T'), \alpha|T'\right) \in \mathfrak{T}_N.$$

For $t \in T$ we write $T \setminus t = T \setminus \{t\}$. We write $\mathfrak{t} \setminus t$ as a shorthand for

$$\mathfrak{t}|(T \setminus t).$$

An *embedding* $\phi : \mathfrak{t} \hookrightarrow \mathfrak{t}'$ of task networks $\mathfrak{t} = (T, \prec, \alpha)$ and $\mathfrak{t}' = (T', \prec', \alpha')$ is an injection $\phi : T \hookrightarrow T'$ that preserves the partial order both ways and satisfies $\alpha' \circ \phi = \alpha$. An *isomorphism* is a bijective embedding. Let $\mathfrak{o} = (\emptyset, \emptyset, \emptyset)$ be the

empty task network. The symbol $\sqcup$ denotes the union of disjoint sets. A task network $\mathrm{t} = (T, \prec, \alpha)$ is a *disjoint union* of a family $\{\mathrm{t}_i : i \in I\}$ of task networks if there exist embeddings $\phi_i : \mathrm{t}_i \hookrightarrow \mathrm{t}$ such that

$$T = \bigsqcup_{i \in I} \operatorname{Im} \phi_i$$

and $\phi_i(t_i) \not\prec \phi_j(t_j)$ whenever $i, j \in I$ are distinct.

An *HTN problem* is a tuple $\Pi = (F, C, O, M, \delta, \mathrm{t}_0, s_0)$ where

- F is a finite set of *(propositional) variables*;
- C is a finite set of *compound task names*;
- O is a finite set of *primitive task names*;
- $M \subseteq C \times \mathfrak{T}_{C \sqcup O}$ is a finite set of *(decomposition) methods*;
- $\delta : O \to \mathscr{P}(F)^4$ is an *action mapping*;[1]
- $\mathrm{t}_0 \in \mathfrak{T}_{C \sqcup O}$ is an *initial task network*;
- $s_0 \subseteq F$ is an *initial propositional state*.

A *propositional state* of Π is a subset of F. For better readability, we will adopt the following style guide: facts are typewriter `blue`, compound task names in bold and **brown**, primitive task names in sans serif pink and decomposition methods in green. Let $\mathrm{t} = (T, \prec, \alpha)$ be a task network over $C \sqcup O$. It is called a task network *of* Π if either $\mathrm{t} = \mathrm{t}_0$ or $(\mathbf{c}, \mathrm{t}) \in M$ for some $\mathbf{c} \in C$. We call a task $t \in T$ *compound* if $\alpha(t) \in C$ and *primitive* if $\alpha(t) \in O$. We call t *primitive* if all tasks in T are primitive (i.e. t is a task network over O).

Applying a decomposition method changes a non-primitive task network into another task network. Suppose that $\mathrm{t}_1 = (T_1, \prec_1, \alpha_1)$, $\mathrm{t}_2 = (T_2, \prec_2, \alpha_2)$ and $\mathrm{t} = (T, \prec, \alpha)$ are task networks, $t \in T_1$ and $\mu = (\alpha_1(t), \mathrm{t})$ is a method. We write $\mathrm{t}_1 \to_{t/\mu} \mathrm{t}_2$ provided there exists an embedding $\phi : \mathrm{t} \hookrightarrow \mathrm{t}_2$ such that

$$T_2 = (T_1 \setminus t) \sqcup \operatorname{Im} \phi$$

and for all $t_1 \in T_1 \setminus t$ and $t' \in T$ and $* \in \{\prec, \succ\}$ it holds

$$t_1 *_1 t \Leftrightarrow t_1 *_2 \phi(t').$$

Intuitively this means that t got replaced by t. The task network t_2 exists and is unique up to isomorphism.

The *search space* of the problem Π is

$$\Omega_\Pi = \mathfrak{T}_{C \sqcup O} \times \mathscr{P}(F).$$

If pr $\in O$ with $\delta(\mathsf{pr}) = (\pi_+, \pi_-, e_+, e_-)$ and $s \subseteq F$ satisfies $\pi_+ \subseteq s$ and $\pi_- \cap s = \emptyset$, we say pr is *applicable* in s and define $\gamma(s, \mathsf{pr}) = s \cup e_+ \setminus e_-$. Let $(\mathrm{t}, s) \in \Omega_\Pi$ be an HTN state with $\mathrm{t} = (T, \prec, \alpha)$ and let $t \in T$ be a primitive $\prec$-minimal task such that $\alpha(t)$ is applicable in s. Then write

$$(\mathrm{t}, s) \rightsquigarrow_\Pi \Big(\mathrm{t} \setminus t, \gamma\big(s, \alpha(t)\big)\Big) \in \Omega_\Pi.$$

If $\mathrm{t}_1 \to_{t/\mu} \mathrm{t}_2$ for some t and $\mu \in M$, also let

$$(\mathrm{t}_1, s) \rightsquigarrow_\Pi (\mathrm{t}_2, s).$$

[1] We include negative preconditions, but it is well-known that these can be compiled away in linear time; cf. (Gazen and Knoblock 1997, section 2.6).

The *search graph* of Π is $\Phi_\Pi = \big(\Omega_\Pi, \rightsquigarrow_\Pi, (\mathrm{t}_0, s_0)\big)$, where (t_0, s_0) is the initial HTN state. For the relation with other views on HTN search, we refer to (Alford et al. 2012). The problem Π is *solvable* if for some propositional state s_ω, the state $(\mathfrak{o}, s_\omega)$ is reachable in Φ_Π. If it is, it can be reached by first applying only decomposition methods until the task network is primitive, and then applying only actions.

If $\mathscr{Q}$ is a class of HTN problems, let PLANEX($\mathscr{Q}$) be the problem of deciding whether or not a given member of $\mathscr{Q}$ is solvable. This problem has been studied in the literature for various classes $\mathscr{Q}$. The complexities range from polynomial time to undecidable ((Erol, Hendler, and Nau 1994), (Geier and Bercher 2011), (Alford 2013), (Alford, Bercher, and Aha 2015)).

Petri Nets

A *Petri net* is a pair $\mathscr{N} = (P, \Theta)$ where P is a finite set of *places* and Θ is a finite set of *transitions*, which are functions $P \to \mathbb{Z}$. The Petri net $\mathscr{N}$ is called *ordinary* if $\big|\theta(p)\big| \leq 1$ for all $\theta \in \Theta$ and $p \in P$. Ordinary Petri nets have the same modelling power as arbitrary Petri nets (see (Murata 1989, section IV.A)). The *search space* $\Omega_\mathscr{N}$ of $\mathscr{N}$ is the set of all functions $P \to \mathbb{N}$. We work with pointwise addition of functions $P \to \mathbb{Z}$. For $\sigma, \sigma' \in \Omega_\mathscr{N}$, we define $\sigma \rightsquigarrow_\mathscr{N} \sigma'$ iff there exists a transition $\theta \in \Theta$ such that $\sigma + \theta = \sigma'$. This is called a *firing* of θ. Let $\Phi_\mathscr{N} = (\Omega_\mathscr{N}, \rightsquigarrow_\mathscr{N})$ be the *search graph*. A state $\sigma \in \Omega_\mathscr{N}$ is called a *unit state* of $\mathscr{N}$ if $\sigma(p) \leq 1$ for all $p \in P$.

Petri nets are often imagined to include an unlimited pool of *tokens*. State σ encodes that there are $\sigma(p)$ tokens at each place $p \in P$.

PETRI is the problem of deciding given an ordinary Petri net $\mathscr{N}$ and unit states τ_0, τ_1 of $\mathscr{N}$ whether or not τ_1 can be reached from τ_0 in $\Phi_\mathscr{N}$.

PETRIGEN is the same problem but without the "ordinary" and "unit" assumptions. For more information on this and similar problems we refer to (Esparza 1998). It is easy to see that PETRIGEN reduces to PETRI in polynomial time. Moreover, the following is standard:

Lemma 1. *Given a finite set* P *of instances of PETRI we can compute in polynomial time another instance of PETRI that has answer "yes" iff some instance in* P *has answer "yes".*

Complexity Theory

In this paper, we will consider complexity classes that lie beyond the typically considered hierarchy of complexity classes starting with $\mathbb{L}, \mathbb{NL}, \mathbb{P}, \mathbb{NP}, \mathbb{PSPACE}, \mathbb{EXP}, \ldots$. We introduce the notion of Ackermann-completeness following (Schmitz 2016).

We define $F_0(n) = n + 2$ and

$$F_k(n) = F_{k-1}^n(k) = \underbrace{F_{k-1}(\ldots(F_{k-1}}_{n \text{ time}}(k))\ldots).$$

(We pass k as an argument here to ensure that $\lim_{k \to \infty} F_k(0) = \infty$.) As a result, we get that $F_1(n) \geq 2n$, $F_2(n) \geq 2^n$ and

$$F_3(n) \geq \underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{n \text{ high}}.$$

Lastly we define $F_\omega(n) = F_n(n)$, which is the Ackermann function. Let ACKERMANN be the class of all decision problems that can be solved by a deterministic Turing machine with a time bound of $F_\omega(F_k(n))$ for an input length n for some $k \geq 1$. A problem solvable by a program with runtime $F_3(n)$ for an input length n, already need not be in the class ELEMENTARY which contains all problems solvable with run-time limited to some fixed height exponentiation tower. Problems in ACKERMANN can be still much harder.

A problem Π is ACKERMANN-hard if for every problem $\Pi' \in$ ACKERMANN there exists $k \in \mathbb{N}$ such that there exists a program that reduces any instance of Π' of some size n to an instance of Π in time at most $F_k(n)$. The intuition behind this definition is that any function F_k is negligibly small in comparison to the Ackermann function F_ω in the limit. As we can choose $k = 2$, it follows that exponential time reductions are valid for proving ACKERMANN-hardness.

Leroux and Schmitz (2019) proved that PETRI $\in$ ACKERMANN. Recently, Czerwiński and Orlikowski complemented this result with hardness:

Theorem 2. *(Czerwiński and Orlikowski 2022) PETRI is* ACKERMANN*-complete.*

New Fragments

Let $\Pi = (F, C, O, M, \delta, \mathfrak{t}_0, s_0)$ be an HTN problem and $\mathfrak{t}_0 = (T_0, \prec_0, \alpha_0)$.

- Π is *initial* if any compound task in any task network of Π is minimal w.r.t. the task network's order:

$$\forall \mathfrak{t} = (T, \prec, \alpha) : \big(\exists \mathbf{c} : (\mathbf{c}, \mathfrak{t}) \in M\big) \text{ or } \mathfrak{t}_0 = \mathfrak{t}$$
$$\implies \forall t \prec t' : \alpha(t) \in O.$$

- Π is *final* if any compound task in any task network of Π is order-maximal.
- Π is *clean* if it is initial and final.
- Π is *one-hole-digging* if every (initial or method) task network contains at most one compound task:

$$\big|\alpha_0^{-1}[C]\big| \leq 1 \;\&\; \forall \big(\mathbf{c}, (T, \prec, \alpha)\big) \in M : \big|\alpha^{-1}[C]\big| \leq 1.$$

- Π is *bottomless* if every primitive method task network is empty:

$$\forall (\mathbf{c}, \mathfrak{t}) \in M : \mathfrak{t} \in \mathfrak{T}_O \implies \mathfrak{t} = \mathfrak{o}.$$

- Π is *loop-unrolling*, if it only contains one compound task name and two methods:

$$|C| \leq 1 \;\&\; |M| \leq 2.$$

Note that a problem is clean iff compound tasks are isolated.

For a one-hole-digging problem, decomposition is a single sequence of methods that are successively applied to the only present compound task, and the number of compound tasks never exceeds 1 while moving through Φ_Π.

If some loop-unrolling Π with at least one compound task in the initial task network $\mathfrak{t}_0$ is solvable, there can only be one method – call it cont – with a non-primitive task network. The other method – call it stop – serves to end recursion. Consider some subcases:

- If Π is additionally one-hole-digging, the only choice to make for the decomposition of Π is how often to apply cont before applying stop.
- If Π is additionally bottomless, then

$$M = \big\{\text{cont} = (\mathbf{comp}, \mathfrak{t}), \text{stop} = (\mathbf{comp}, \mathfrak{o})\big\} \qquad (1)$$

  for some **comp** and $\mathfrak{t}$. If Π is additionally clean, let $\mathfrak{t}^{\text{pr}}_{(0)}$ be the largest primitive task subnetwork of $\mathfrak{t}_{(0)}$; then the primitive task networks obtainable from $\mathfrak{t}_0$ with the decomposition methods in M are precisely the disjoint unions of $\mathfrak{t}^{\text{pr}}_0$ with any number of copies of $\mathfrak{t}^{\text{pr}}$.

Example 3. *Imagine we want to bury an object. The procedure is to dig a hole in the ground, put the object in it, and then cover it with the dirt that was dug up. The hole can be of any depth. Let* $F = \{\texttt{hole}, \texttt{buried}\}$, $C = \{\textbf{bury}\}$, $O = \{\textit{dig}, \textit{put}, \textit{cover}\}$,

$$M = \big\{\textit{deeper} = (\textbf{bury}, \mathfrak{t}_{\textit{deeper}}), \textit{bottom} = (\textbf{bury}, \mathfrak{t}_{\textit{bottom}})\big\}$$

where $\mathfrak{t}_{\textit{deeper}}$ *is the task network consisting of three tasks* $t^{\textit{dig}}_{\textit{deeper}} \prec t^{\textit{bury}}_{\textit{deeper}} \prec t^{\textit{cover}}_{\textit{deeper}}$ *with names* $\alpha_{\textit{deeper}}(t^{\textit{dig}}_{\textit{deeper}}) = \textit{dig}$, $\alpha_{\textit{deeper}}(t^{\textit{bury}}_{\textit{deeper}}) = \textbf{bury}$ *and* $\alpha_{\textit{deeper}}(t^{\textit{cover}}_{\textit{deeper}}) = \textit{cover}$, $\mathfrak{t}_{\textit{bottom}}$ *is the task network consisting of one task* $t_{\textit{bottom}}$ *with name* $\alpha_{\textit{bottom}}(t_{\textit{bottom}}) = \textit{put}$, $\delta(\textit{dig}) = \langle \emptyset, \emptyset, \{\texttt{hole}\}, \emptyset\rangle$, $\delta(\textit{put}) = \langle \{\texttt{hole}\}, \emptyset, \{\texttt{buried}\}, \emptyset\rangle$, $\delta(\textit{cover}) = \langle \emptyset, \emptyset, \emptyset, \emptyset\rangle$, $\mathfrak{t}_0$ *is the task network consisting of one task* t_0 *with name* $\alpha_0(t_0) = \textbf{bury}$ *and* $s_0 = \emptyset$. *Then* $\Pi = (F, C, O, M, \delta, \mathfrak{t}_0, s_0)$ *is a one-hole-digging loop-unrolling problem. Every path of* Φ_Π *leads to the goal, except when one immediately applies the* bottom *method or when one applies the* deeper *method forever.*

Let $\mathscr{I}$ be the class of initial problems. Let $\mathscr{F}$ be the class of final problems. Let $\mathscr{C}$ be the class of clean problems. Let $\mathscr{H}_1$ be the class of one-hole-digging problems. Let $\mathscr{B}$ be the class of bottomless problems. Let $\mathscr{L}$ be the class of loop-unrolling problems.

All *regular* problems, introduced by (Erol, Hendler, and Nau 1994), are final and one-hole-digging. However, not all final one-hole-digging problems are regular (there can be multiple maximal tasks). Still, if all task networks of some final problem are totally ordered, it is regular.

We call an HTN problem Π *quasi-final* if removing all maximal tasks from all task networks of Π results in an *acyclic* problem (see again (Erol, Hendler, and Nau 1994)). Then the remaining compound tasks can be compiled away in exponential time; thus we can compute a final problem that is solvable iff Π is solvable. Let $\mathscr{F}'$ be the class of quasi-final problems. Then it follows from the results below that PLANEX$(\mathscr{F}') \in$ ACKERMANN. Similar remarks hold for initial problems. Out of the IPC 2023, the domains AssemblyHierarchical, Blocksworld-HPDDL, Multiarm-Blocksworld, Robot, and Tower were quasi-final. However, as the task networks of these examples are totally ordered, they are actually in EXP (Erol, Hendler, and Nau 1996), so much easier than Ackermann.

Results

Intuitively, the class $\mathscr{H}_1$ might be complexity-wise close to other HTN classes or even classical planning as the restrictions to the allowed decompositions are severe. A method application either seems to "move the problem" by replacing the compound task with a new compound task and some primitive tasks, or readily creates a primitive task network. Example 3 is trivial, but this is largely caused by the total order in the method task network $\mathfrak{t}_{\text{deeper}}$. We will prove that the complexity with partial order is high: the problem PLANEX($\mathscr{H}_1$) is ACKERMANN-complete. It is thus significantly more complex than any other known decidable HTN class and only barely easier than undecidable problems. The proof heavily relies on Theorem 2, by showing equivalence of PLANEX($\mathscr{H}_1$) to PETRI. As a bonus, our reductions work for $\mathscr{I}$ and $\mathscr{F}$ as well.

Membership

The idea of the membership proof is to use bi-directional search, a common technique in planning that already exists since (Pohl 1969). If $\Pi = (F, C, O, M, \delta, \mathfrak{t}_0, s_0)$ is an HTN problem, its *bi-directional search space* is

$$\Omega_\Pi^{\text{bi}} = \mathfrak{T}_{C \sqcup O} \times \mathscr{P}(F)^2.$$

The first propositional state in a triple in Ω_Π^{bi} is understood as the propositional state in forward search and the second in backward search. The search graph $\Phi_\Pi^{\text{bi}} = (\Omega_\Pi^{\text{bi}}, \rightsquigarrow_\Pi^{\text{bi}})$ over the bi-directional search space inherits the arrows $\rightsquigarrow_\Pi$ (where the propositional state in backward search does not change) with the addition of regression:

$$\Big(\mathfrak{t}, s, \gamma(s_1, \alpha(t))\Big) \rightsquigarrow_\Pi^{\text{bi}} (\mathfrak{t} \setminus t, s, s_1)$$

whenever $\mathfrak{t} = (T, \prec, \alpha)$ and $t \in T$ is $\prec$-maximal. It is easy to see that Π is solvable iff $(\mathfrak{o}, s, s)$ can be reached from $(\mathfrak{t}_0, s_0, s_\omega)$ in Φ_Π^{bi} for some propositional states s, s_ω.

If $\Pi \in \mathscr{H}_1$, we also inherit the property that the number of compound tasks remains at most 1 while moving through Φ_Π^{bi}.

Proposition 4. *PLANEX($\mathscr{H}_1 \cup \mathscr{I} \cup \mathscr{F}$) reduces to PETRI in exponential time.*

Proof. For each $\mathscr{Q} \in \{\mathscr{H}_1, \mathscr{I}, \mathscr{F}\}$ it is polynomial to decide whether a given HTN problem is in $\mathscr{Q}$. Hence it suffices to show for each such $\mathscr{Q}$ that PLANEX($\mathscr{Q}$) reduces to PETRI in at most exponential time.

In view of Lemma 1, it suffices to reduce the problem of determining whether $(\mathfrak{o}, s_\omega)$ can be reached in Φ_Π given $\Pi \in \mathscr{Q}$ and a propositional state s_ω. This is equivalent to asking whether there exists a propositional state s_1 such that $(\mathfrak{o}, s_1, s_1)$ can be reached from $(\mathfrak{t}_0, s_0, s_\omega)$ in Φ_Π^{bi}. Again by Lemma 1 we can assume s_1 is given and reduce the problem of determining whether

$$(\mathfrak{o}, s_1, s_1) \text{ can be reached from } (\mathfrak{t}_0, s_0, s_\omega) \text{ in } \Phi_\Pi^{\text{bi}}. \qquad (2)$$

Let $\Pi = (F, C, O, M, \delta, \mathfrak{t}_0, s_0)$ be an HTN problem. Let $\mathfrak{X}$ be the set of all nonempty task subnetworks of (initial or method) task networks of Π. For each $\mathfrak{x} \in \mathfrak{X}$, introduce a Petri place $p(\mathfrak{x})$. For each $s \subseteq F$ and $\mathfrak{v} \in \{0, 1\}$, introduce a Petri place $p(s, \mathfrak{v})$. Let P be the set of all places. A state $\sigma : P \to \mathbb{N}$ satisfying

$$\sum_{s \in \mathscr{P}(F)} \sigma\big(p(s, \mathfrak{v})\big) = 1$$

for each $\mathfrak{v} < 2$, will encode an element of the bi-directional search space of Π. Namely, the task network is a disjoint union of $\sigma\big(p(\mathfrak{x})\big)$ copies of $\mathfrak{x}$ for each $\mathfrak{x} \in \mathfrak{X}$; the propositional state in forward search is the unique s such that $\sigma\big(p(s, 0)\big) = 1$ and the propositional state in backward search is the unique s' such that $\sigma\big(p(s', 1)\big) = 1$. Accordingly, let τ_0 be the unit state that encodes $(\mathfrak{t}_0, s_0, s_\omega)$ (naturally with $\tau_0\big(p(\mathfrak{t}_0)\big) = 1$), and τ_1 the unit state that encodes $(\mathfrak{o}, s_1, s_1)$.

Suppose that $s \subseteq F$ and $\mathsf{pr} \in O$ is applicable in s. Also let $\mathfrak{x} = (X, \prec, \alpha) \in \mathfrak{X}$ and $x \in X$ with name $\alpha(x) = \mathsf{pr}$. If $x \in X$ is $\prec$-minimal, we define the transition $\theta_0^{s,\mathfrak{x},x}$ by

$$\theta_0^{s,\mathfrak{x},x}\big(p(\mathfrak{y})\big) = \begin{cases} 1 & (\mathfrak{y} = \mathfrak{x} \setminus x) \\ -1 & (\mathfrak{y} = \mathfrak{x}) \\ 0 & (\text{otherwise}), \end{cases}$$

$$\theta_0^{s,\mathfrak{x},x}\big(p(s', 0)\big) = \begin{cases} 1 & \big(s' = \gamma(s, \mathsf{pr})\big) \\ -1 & \big(s' = s\big) \\ 0 & (\text{otherwise}), \end{cases}$$

$$\theta_0^{s,\mathfrak{x},x}\big(p(s', 1)\big) = 0,$$

This transition corresponds to executing task x in forward search. If $x \in X$ is $\prec$-maximal, define the transition $\theta_1^{s,\mathfrak{x},x}$ by

$$\theta_1^{s,\mathfrak{x},x}\big(p(\mathfrak{y})\big) = \begin{cases} 1 & (\mathfrak{y} = \mathfrak{x} \setminus x) \\ -1 & (\mathfrak{y} = \mathfrak{x}) \\ 0 & (\text{otherwise}), \end{cases}$$

$$\theta_1^{s,\mathfrak{x},x}\big(p(s', 0)\big) = 0,$$

$$\theta_1^{s,\mathfrak{x},x}\big(p(s', 1)\big) = \begin{cases} 1 & \big(s' = s\big) \\ -1 & \big(s' = \gamma(s, \mathsf{pr})\big) \\ 0 & (\text{otherwise}). \end{cases}$$

This transition corresponds to executing task x in backward search.

We shall see that firing transitions $\theta_\mathfrak{v}^{s,\mathfrak{x},x}$ gets rid of primitive tasks until a compound task is *isolated*. For each method $\mu = (\mathbf{c}, \mathfrak{t}) \in M$ and $\mathfrak{x} = (X, \prec, \alpha) \in \mathfrak{X}$ such that there exists an $\prec$-*isolated* $x \in X$ with name $\alpha(x) = \mathbf{c}$, introduce a transition $\theta_{\mathfrak{x},x,\mu}$ with

$$\theta_{\mathfrak{x},x,\mu}\big(p(\mathfrak{y})\big) = \begin{cases} 1 & (\mathfrak{y} = \mathfrak{t}) \\ 1 & (\mathfrak{y} = \mathfrak{x} \setminus x) \\ -1 & (\mathfrak{y} = \mathfrak{x}) \\ 0 & (\text{otherwise}), \end{cases}$$

$$\theta_{\mathfrak{x},x,\mu}\big(p(s, \mathfrak{v})\big) = 0.$$

(We have to include the subscript x because if $\mathscr{Q} \neq \mathscr{H}_1$ the initial task network $\mathfrak{t}_0 \in \mathfrak{X}$ may contain multiple tasks with name $\mathbf{c}$.) Note that x is isolated in the encoded task network

because x is assumed to be isolated in $\mathfrak{x}$ and there is no order between the various $\mathfrak{x}$. Hence it should be clear that firing $\theta_{\mathfrak{x},x,\mu}$ corresponds to an application of the method μ to x in one of the copies of $\mathfrak{x}$ in the encoded task network.

Let Θ be the set of all transitions introduced above and $\mathcal{N} = (P, \Theta)$.

We claim that (2) holds iff τ_1 can be reached from τ_0 in the search space $\Phi_{\mathcal{N}}$.

Since the firings of $\mathcal{N}$ translate into movements of Φ^{bi}_{Π}, the direction "$\Leftarrow$" is clear. This direction of the proof actually works for any HTN problem Π.

Conversely, assume (2). First consider the case $\Pi \in \mathscr{H}_1$. Then any task network reachable from $(\mathfrak{t}_0, s_0, s_\omega)$ in Φ^{bi}_{Π} has at most one compound task. Decomposing a compound task x with a method can always be deferred until x is isolated: if there is a (primitive) predecessor task $t \prec x$, then t can be executed in forward search before decomposing x, since all primitive tasks executed before t have to be already present in the task network because there is no compound task besides x in the task network; similarly, if there is a (primitive) successor task, it can be executed in backward search before decomposing x. Such a solution is exactly what $\mathcal{N}$ captures.

Next suppose that $\Pi \in \mathscr{I}$. Then solving Π with backward search and deferring decompositions for as long as possible implies that again only isolated compound tasks will be decomposed. Thus $\mathcal{N}$ encodes the solution. The argument for $\mathscr{F}$ is analogous using forward search. □

Hardness

Next, we turn from showing membership (and thus decidability) of $\mathscr{I}, \mathscr{H}_1$, and $\mathscr{F}$ to showing hardness. While from Example 3, one might suspect that these problems could be computationally easy, we show that even clean, bottomless, one-hole-digging, loop-unrolling problems are in general much more complex. To be precise, we show that even this highly restricted class of problems is also hard for the class ACKERMANN.

To establish this result, we show that the reachability problem for Petri nets can be encoded in such an HTN planning problem. For transparency, we don't reduce from the general PETRIGEN, but restrict ourselves to ordinary Petri nets with unit states, i.e. PETRI.

Note that parts of the construction – notably the concept of trashing – are only necessary as we want to establish hardness even for loop-unrolling problems. This proof translates to an easier version without the loop-unrolling property and without trashing.

In Figure 1 we provide an example for a plan that simulates a concrete Petri net, which might be helpful to the reader. An explanation of the example can be found at the end of the proof. Further Figure 2 shows the tasks contained in the construction's only recursive method task network while Table 2 shows the preconditions and effects of all actions compactly.

Proposition 5. *PETRI reduces to PLANEX*$(\mathscr{C} \cap \mathscr{H}_1 \cap \mathscr{L} \cap \mathscr{B})$ *in polynomial time.*

Proof. Let $\mathcal{N} = (P, \Theta)$ be an ordinary Petri net and τ_0 and τ_1 unit states of $\mathcal{N}$. We define $\Pi = (F, C, O, M, \delta, \mathfrak{t}_0, s_0) \in \mathscr{C} \cap \mathscr{H}_1 \cap \mathscr{L} \cap \mathscr{B}$. Let $C = \{\textbf{comp}\}$ and M as in (1). τ_1 will be reachable from τ_0 in $\Phi_{\mathcal{N}}$ iff a large number of applications of cont yields a solution to Π.

Let

$$F = \big\{\texttt{searchMode}, \texttt{pTrashMode}, \texttt{rTrashMode}, \texttt{transInProg}, \texttt{pTrashInProg}, \texttt{inc}(p), \texttt{dec}(p) : p \in P\big\}. \quad (3)$$

We shall design our problem in such a way that any solution to Π can be split into three phases, that are characterized by the truth of the variables in (3) and separated by the tasks in O_{main}. searchMode simulates the search in $\Phi_{\mathcal{N}}$, pTrashMode trashes unused place tokens and rTrashMode trashes any remaining tasks.

$$O_{\text{main}} = \{\text{startPTrashMode}, \text{startRTrashMode}\},$$

$$O' = \big\{\text{inc}(p), \text{dec}(p), \text{startPTrash}, \text{endPTrash}, \text{startTrans}, \text{endTrans}, \text{requestInc}(p), \text{checkInc}(p), \text{requestDec}(p), \text{checkDec}(p), \text{fakeInc}(p), \text{fakeDec}(p) : p \in P\big\},$$

$$O = O_{\text{main}} \sqcup O'.$$

The cont network $\mathfrak{t}$ is given by Figure 2 where α removes subscripts. For any $p \in P$, the tasks $\text{inc}(p)$ (to be read "increment p") and $\text{dec}(p)$ (to be read "decrement p") of $\mathfrak{t}$ together encode a single occurrence of a token at place p (at least during searchMode). Specifically, for each $p \in P$, the value of p in a state of $\mathcal{N}$ is given by the number of copies of $\mathfrak{t}$ of which task $\text{inc}(p)$ has been performed but task $\text{dec}(p)$ has not. δ is given by Table 2 and $\mathfrak{t}_0 = (T_0, \prec_0, \alpha_0)$ with

$$T_0 = \{\textbf{comp}\} \sqcup \Big(\big\{\text{requestInc}(p) \prec_0 \text{checkInc}(p) : p \in P \,\&\, \tau_0(p) = 1\big\} \prec_0 \quad (4)$$

$$\big\{\text{requestDec}(p) \prec_0 \text{checkDec}(p) : p \in P \,\&\, \tau_1(p) = 1\big\} \prec_0 \quad (5)$$

$$O_{\text{main}}\Big),$$

$\alpha_0 = \text{id}$ and $s_0 = \{\texttt{searchMode}\}$.

Observe that in searchMode we can only execute "token tasks" ($\text{inc}(p)$, $\text{dec}(p)$), "transition tasks" (right-most columns in Figure 2) and "boundary tasks" ((4)–(5)). The boundary tasks (4) generate τ_0 and the boundary tasks (5) consume τ_1. It is w.l.o.g. that a plan starts with (4) and the searchMode ends with (5) (both accompanied by executions of token tasks). Firing a transition θ corresponds to executing an entire chain of six transition tasks of θ, also accompanied by executions of token tasks. The variable transInProg prevents that we work on two transitions at the same time. I.e. once startTrans_θ is executed, the task endTrans_θ in the same copy of $\mathfrak{t}$ has to be executed before any other $\text{startTrans}_{\theta'}$ can be executed. The intermediate transition tasks ensure that the Petri net state encoded by the HTN state is updated appropriately. All boundary tasks have to be executed before we execute

cont, stop,
$\mathsf{startTrans}_\theta$, $\mathsf{requestInc}(p)_\theta$, $\mathsf{inc}(p)$, $\mathsf{checkInc}(p)_\theta$, $\mathsf{endTrans}_\theta$,
$\mathsf{requestDec}(p)$, $\mathsf{dec}(p)$, $\mathsf{checkDec}(p)$,
$\mathsf{startPTrashMode}$,
$\mathsf{startPTrash}_q$, $\mathsf{requestInc}(q)$, $\mathsf{inc}(q)$, $\mathsf{checkInc}(q)$,
$\mathsf{requestDec}(q)$, $\mathsf{dec}(q)$, $\mathsf{checkDec}(q)$, $\mathsf{endPTrash}_q$,
$\mathsf{startRTrashMode}$,
$\mathsf{startPTrash}_p$, $\mathsf{requestInc}(p)$, $\mathsf{fakeInc}(p)$, $\mathsf{checkInc}(p)$,
$\mathsf{requestDec}(p)$, $\mathsf{fakeDec}(p)$, $\mathsf{checkDec}(p)$, $\mathsf{endPTrash}_p$,
$\mathsf{fakeInc}(q)$, $\mathsf{fakeDec}(q)$,
$\mathsf{startTrans}_\eta$, $\mathsf{requestInc}(q)_\eta$, $\mathsf{fakeInc}(q)_\eta$, $\mathsf{checkInc}(q)_\eta$, $\mathsf{endTrans}_\eta$,
$\mathsf{fakeInc}(p)_\theta$

Figure 1: Example solution to Π (tasks in T_0).

	p	q
θ	1	0
η	0	1
τ_0	0	0
τ_1	1	0

Table 1: Example instance of PETRI.

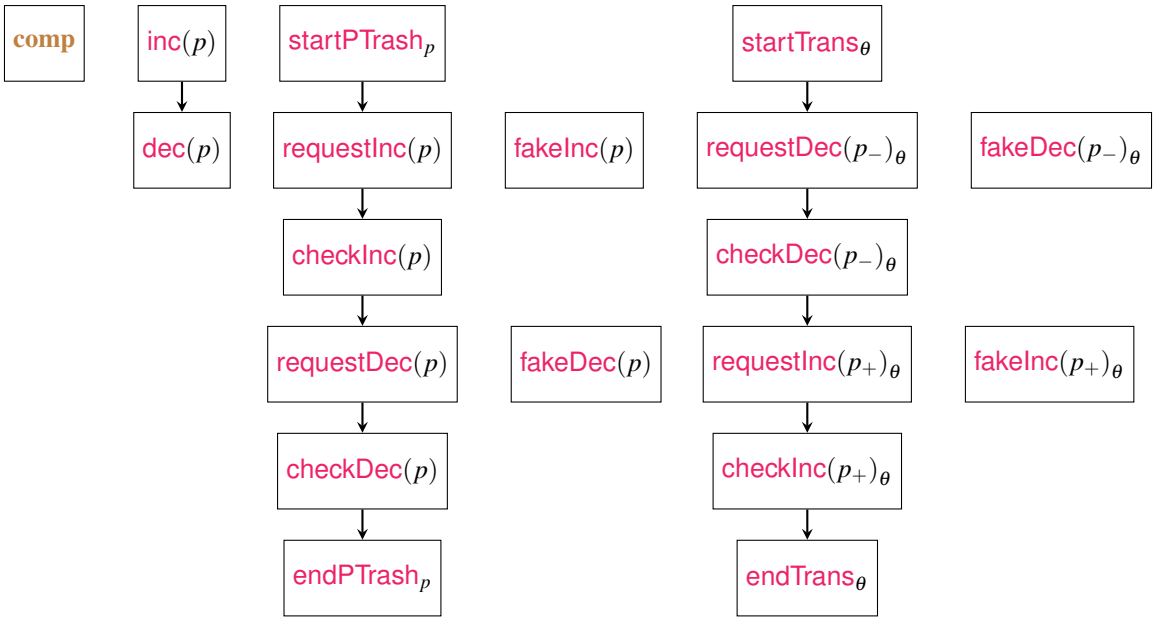


Figure 2: Task network $\mathfrak{t} = (T, \prec, \alpha)$ (include instances for all $p, p_-, p_+ \in P$, $\theta \in \Theta$ with $\theta(p_+) = 1$, $\theta(p_-) = -1$).

pr	π_+	π_-	e_+	e_-
$\mathsf{startPTrashMode}$	searchMode	transInProg	pTrashMode	searchMode
$\mathsf{startRTrashMode}$	pTrashMode	pTrashInProg	rTrashMode	pTrashMode
$\mathsf{startPTrash}$		searchMode pTrashInProg	pTrashInProg	
$\mathsf{endPTrash}$				pTrashInProg
$\mathsf{startTrans}$		pTrashMode transInProg	transInProg	
$\mathsf{endTrans}$				transInProg
$\mathsf{inc}(p)$	inc(p)	rTrashMode		inc(p)
$\mathsf{dec}(p)$	dec(p)	rTrashMode		dec(p)
$\mathsf{requestInc}(p)$		inc(p)	inc(p)	
$\mathsf{checkInc}(p)$		inc(p)		
$\mathsf{requestDec}(p)$		dec(p)	dec(p)	
$\mathsf{checkDec}(p)$		dec(p)		
$\mathsf{fakeInc}(p)$	rTrashMode			inc(p)
$\mathsf{fakeDec}(p)$	rTrashMode			dec(p)

Table 2: Action mapping $\delta(\mathsf{pr}) = (\pi_+, \pi_-, e_+, e_-)$.

startPTrashMode. Hence the crucial claim is that an HTN state without remaining boundary tasks and with propositional state {searchMode} encodes the zero state of $\mathcal{N}$ iff executing startPTrashMode allows one to reach $\mathfrak{o}$ in Φ_Π. But in pTrashMode, only token tasks and tasks in the third column of Figure 2 can be executed. Hence in view of pTrashInProg, for each $p \in P$, equally many copies of inc(p) as dec(p) are executed in this phase, so (since they cannot be executed in the final phase) the HTN state must encode the zero state of $\mathcal{N}$ when entering pTrashMode. Conversely, it is easy to show that after trashing the tokens all remaining tasks can be finished in rTrashMode using the "fake" tasks.

As an example, suppose that $P = \{p, q\}$ and $\Theta = \{\theta, \eta\}$ and τ_0, τ_1 are as in Table 1. Then τ_1 is reachable from τ_0 by firing just θ. Hence to solve Π only one copy of $\mathfrak{t}$ is needed. See Figure 1. Notice that at the start of pTrashMode, both slot tasks for p have been executed while neither slot task for q has been executed; at the start of rTrashMode, all slot tasks have been executed. □

The method task network $\mathfrak{t}$ in Figure 2 has the shape of *parallel sequences*. This type of structure occurs frequently in HTN planning; cf. (Behnke et al. 2022).

Conclusion and Future Work

We introduced several new fragments of HTN planning and proved that multiple combinations of them are complete for the large class ACKERMANN of decidable problems:

Theorem 6. *Let* $\mathscr{C} \cap \mathscr{H}_1 \cap \mathscr{L} \cap \mathscr{B} \subseteq \mathscr{Q} \subseteq \mathscr{H}_1 \cup \mathscr{I} \cup \mathscr{F}$. *Then* PLANEX($\mathscr{Q}$) *is* ACKERMANN*-complete.*

Proof. Theorem 2 and Propositions 4 and 5. □

Table 3 lists some natural fragments of HTN planning. Notice that the new fragments have higher computational complexity than the previously known ones. Our proof relies on the recently established ACKERMANN-completeness of the Petri net reachability problem. In future work we hope to further investigate the relationship between HTN planning and Petri nets, and use it to invent new algorithms for HTN planning. In particular, the construction in the proof of Proposition 4 can be carried out for any HTN problem, and may provide a heuristic.

Moreover, we would like to investigate the class $\mathscr{H}_2$ of *two-hole-digging* problems, that have two compound tasks in the initial task network but only one compound task per method task network. It is known that PLANEX($\mathscr{H}_2$) is undecidable; see (Höller et al. 2023). However, is PLANEX($\mathscr{H}_2 \cap \mathscr{L}$) decidable?

References

Alford, R. 2013. *Search complexities for HTN planning*. Ph.D. thesis.

Alford, R.; Behnke, G.; Höller, D.; Bercher, P.; Biundo, S.; and Aha, D. W. 2016. Bound to Plan: Exploiting Classical Heuristics via Automatic Translations of Tail-Recursive HTN Problems. In *Proceedings of the 26th International Conference on Automated Planning and Scheduling, (ICAPS 2016)*, 20–28. AAAI Press.

Alford, R.; Bercher, P.; and Aha, D. W. 2015. Tight Bounds for HTN Planning. In *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS 2015)*, 7–15. AAAI Press.

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

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

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

Behnke, G.; Pollitt, F.; Höller, D.; Bercher, P.; and Alford, R. 2022. Making Translations to Classical Planning Competitive With Other HTN Planners. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI 2022)*, 9687–9697. AAAI Press.

Czerwiński, W.; and Orlikowski, L. 2022. Reachability in Vector Addition Systems is Ackermann-complete. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, 1229–1240.

Erol, K.; Hendler, J.; and Nau, D. 1994. HTN Planning: Complexity and Expressivity. In *Proceedings of the Association for the Advancement of Artificial Intelligence*.

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

Esparza, J. 1998. Decidability and Complexity of Petri Net Problems – an Introduction. *Lecture Notes in Computer Science*, 1491: 374–428.

Gazen, B.; and Knoblock, C. 1997. Combining the expressivity of UCPOP with the efficiency of graphplan. In *Proceedings of the European Conference on Planning: Recent Advances in AI Planning*, volume 4, 221–233. Springer.

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

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

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language Classification of Hierarchical Planning Problems. In *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI 2014)*, volume 263, 447–452. IOS Press.

Fragment $\mathscr{Q}$	Complexity of PLANEX($\mathscr{Q}$)	Reference
All HTN problems	Undecidable	(Erol, Hendler, and Nau 1994)
$\mathscr{H}_1$ (one-hole-digging)	ACKERMANN	Theorem 6
$\mathscr{I}$ (initial)	ACKERMANN	Theorem 6
$\mathscr{F}$ (final)	ACKERMANN	Theorem 6
$\mathscr{C}$ (clean)	ACKERMANN	Theorem 6
Tail-recursive	EXPSPACE	(Alford, Bercher, and Aha 2015)
Acyclic	NEXP	(Alford, Bercher, and Aha 2015)
Total order	EXP	(Alford, Bercher, and Aha 2015)
Regular	PSPACE	(Erol, Hendler, and Nau 1994)

Table 3: Some fragments of HTN planning and their complexities.

Höller, D.; Lin, S.; Erol, K.; and Bercher, P. 2023. From PCP to HTN Planning Through CFGs. *The 10th International Planning Competition – Planner and Domains Abstracts*.

Leroux, J.; and Schmitz, S. 2019. Reachability in Vector Addition Systems is Primitive-Recursive in Fixed Dimension. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science*, 50, 1–13.

Magnaguagno, M. C.; Meneguzzi, F.; and de Silva, L. 2021. HyperTensioN: A three-stage compiler for planning. In *Proceedings of 10th International Planning Competition: planner and domain abstracts (IPC 2020)*.

Murata, T. 1989. Petri Nets: Properties, Analysis and Applications. In *Proceedings of the IEEE*, volume 77, 541–580.

Nau, D.; Cao, Y.; Lotem, A.; and Munoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI 1999)*, 968–973.

Pohl, I. 1969. Bi-directional and Heuristic Search in Path Problems.

Sacerdoti, E. D. 1975. *A structure for plans and behavior*. Ph.D. thesis, Department of Computer Science, Stanford University.

Schmitz, S. 2016. Complexity Hierarchies beyond Elementary. *ACM Transactions on Computation Theory*, 8(1).

Schreiber, D. 2021. Lifted Logic for Task Networks: TOHTN Planner Lilotane in the IPC 2020. In *Proceedings of 10th International Planning Competition: planner and domain abstracts (IPC 2020)*.

Correcting Totally Ordered Hierarchical Plans by Action Deletion and Insertion

Kristýna Pantůčková, Roman Barták

Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic
{pantuckova, bartak}@ktiml.mff.cuni.cz

Abstract

Hierarchical planning extends classical planning by capturing the hierarchical structure of tasks. Plan correction, an extension of plan verification, involves assessing the validity of a given plan. When an input plan is invalid, the solver searches for a valid alternative plan closest to the original plan. Currently, the only approach for plan correction solely supports action deletion from the original plan. This paper presents the first approach supporting action insertion and deletion within totally ordered hierarchical domain models. Moreover, this new approach is more efficient than the existing technique when only action deletion is allowed.

Introduction

Hierarchical planning (Erol, Hendler, and Nau 1996) extends classical planning by modeling the natural hierarchy of tasks. Complex tasks decompose into simpler subtasks until the simple executable tasks (actions) are reached. The goal of a hierarchical planner is to find such a decomposition for a given root task. Conversely, plan verification aims to determine whether a given sequence of actions is a valid decomposition of any task. Plan correction extends plan verification by providing a valid plan that can be obtained from the invalid plan by a minimum number of corrections.

In real-world scenarios, observations of plans are frequently incomplete or even incorrect. Such observations may encompass actions that have not been executed, or the observer might overlook some actions, leading to an incomplete plan. In this case, a mere statement that the observed plan is invalid may not be the desired answer. Plan correction techniques provide a valid alternative plan that is as close to the observed sequence of actions as possible.

The pioneering work introducing hierarchical plan correction (Barták et al. 2021) defined two possible correction steps: action deletion and action insertion. However, the correction technique presented there supported action deletion only. This paper proposes a novel approach to correct totally ordered hierarchical plans by action deletion and insertion. In totally ordered hierarchical domain models, complex tasks decompose into totally ordered sequences of subtasks. Totally ordered domains can naturally describe many problems. For example, the International Planning Competition (IPC) 2020 used 33 hierarchical domain models, of which 24 were totally ordered.

This paper presents the first approach to correcting totally ordered hierarchical plans supporting both action deletion and insertion. Moreover, if the proposed solver corrects plans solely by action deletion, it is faster than the only existing approach (Barták et al. 2021). HTN plan correction by action deletion and action insertion leads to many possible applications including:

- HTN plan verification (when neither action deletion nor action insertion is enabled), see (Pantůčková, Ondrčková, and Barták 2024);
- correcting an HTN plan for a known goal (when the possible top-level task is given);
- HTN plan recognition with full observability (when action deletion is disabled and action insertion is enabled only after the observed plan prefix), see (Pantůčková and Barták 2023);
- HTN plan recognition with partial observability (when action deletion is disabled and action insertion is enabled anywhere in the observed sequence);
- HTN plan recognition with full or partial observability and with noise (when action deletion is enabled);
- HTN planning (when action insertion is enabled and the input plan is empty).

The paper is organized as follows. We first provide the necessary background on hierarchical plan correction and summarize the related work. Then, we describe the novel plan correction algorithm based on Earley parser, and finally, we present empirical evaluation results. We compare the performance of plan correction by action deletion with the existing technique (Barták et al. 2021). Then, we study the performance of action deletion and action insertion separately, and finally, we assess the performance of our solver when both means of plan correction are allowed.

Background on hierarchical plan correction

Hierarchical planning focuses on planning problems where complex (abstract) tasks decompose into simpler subtasks until a sequence of indecomposable tasks (actions) is reached. Similarly to classical planning, actions are defined by preconditions (propositions that must hold in the state where the action will be executed) and effects (propositions

that will hold after the action is executed). Hierarchical planning is often described by the formalism of hierarchical task networks (HTN).

A domain model can be defined as $\mathcal{D} = (P, T, A, R)$, where P is a set of predicates, T is a set of abstract tasks, A is a set of actions and R is a set of decomposition rules. A rule $T \rightarrow T_1, ..., T_n\ [C]$ decomposes the task T into subtasks $T_1, ..., T_n$. C is a set of rule constraints, which can contain ordering conditions, before-constraints, and between-constraints:

- an ordering constraint $T_i \prec T_j$ enforces the order of actions into which the tasks T_i and T_j will be decomposed; i.e., the last action of the task T_i must be executed before the first action of the task T_j;
- *before*(T', p), indicates that the proposition p must hold in the state in which the first action of the first task in the set T' is executed; and
- *between*(T', T'', p) indicates that p must hold in all states between the execution of the last action of the tasks in T' and the execution of the first action of the tasks in T''.

In a totally ordered domain model, actions into which the subtasks decompose must be executed in the given order, i.e., all actions of T_i are executed before the actions of T_{i+1}.

Given an observed sequence of actions, plan correction aims to find a task that decomposes into a sequence of actions closest to the observed sequence, where the number of corrections measures the distance between plans. We allow corrections of two types: deleting one of the observed actions or inserting a new action into the observed plan (Barták et al. 2021).

Formally, a *plan correction problem* is defined as $\mathcal{P} = (\mathcal{D}, C, I, O)$, where $\mathcal{D}$ is a domain model, C is a set of constants, $O =< o_1, ..., o_n >$ is a sequence of observed actions and I is the initial state (a set of propositions that were valid before the actions were executed). A (optimal) solution to the plan correction problem is an action sequence $\pi =< a_1, ..., a_m >$ such that:

- π is a valid plan with respect to the domain model, that is, π is executable at state I and there exists some task that decomposes to π with respect to domain model $\mathcal{D}$,
- there is a function $f : \{1, ..., n\} \rightarrow \{1, ..., m\} \cup \{nil\}$ such that
  - $\forall i < j : f(i) < f(j) \vee f(i) = nil \vee f(j) = nil$ (actions are not swapping positions),
  - $\forall i : f(i) \neq nil \implies o_i = a_{f(i)}$ (actions preserved in the plan),
  - $del = \{i | f(i) = nil\}$ (actions deleted from O),
  - $add = \{j | \neg\exists i : f(i) = j\}$ (actions added to π),
  - $|del| + |add|$ is minimal among all plans π.

Example 1. *Figure 1 contains an example of a hierarchical task network where the goal is to deliver the package pkg1 to the location loc3. The root task deliver(pkg1, loc3) decomposes into three subtasks: load the package at the location loc1 (pickup(pkg1, loc1)), go to the target location loc3 (get_to(loc3)) and unload the package at the location loc3 (drop(pkg1, loc3)). The middle subtask get_to(loc3) is realized by driving from the location loc1 to loc3 through the location loc2, therefore, it decomposes into two subtasks: drive(loc1, loc2), drive(loc2, loc3). The leaf tasks (pickup(pkg1, loc1), drive(loc1, loc2), drive(loc2, loc3), and drop(pkg1, loc3)) are actions, which can be executed, the internal nodes (deliver(pkg1, loc3) and get_to(loc3)) are abstract tasks.*

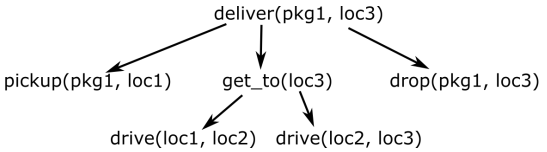


Figure 1: An example of a hierarchical task network.

The decomposition tree includes two decomposition rules: one rule decomposes the top-level task deliver(pkg1, loc3) into three subtasks, and the other rule decomposes the middle subtask get_to(loc3) into two subtasks. Subtasks of both rules are totally ordered. The preconditions of the action pickup(pkg1, loc1) ensure that both the package pkg1 and the truck are at the location loc1 and its effect states that pkg1 is loaded into the truck. Similarly, preconditions of drop(pkg1, loc3) ensure that the truck is at loc3, and after this action is executed, pkg1 will no longer be loaded into the truck. Each action drive must be executed in the first location so the truck will move to the second location.

Related works

The problem of deciding whether a given HTN plan is valid is tackled by plan verification techniques. Existing approaches are based on parsing (Lin et al. 2023), compilation to SAT (Behnke, Höller, and Biundo 2017), or compilation of HTN plan verification problems to planning problems (Höller et al. 2022). Another related problem is HTN plan recognition, where a prefix of an HTN plan is given as part of the input, and the goal is to find an abstract task that can be decomposed into a plan with the given prefix. Similarly to plan verification, recent existing approaches are based either on parsing (Barták, Maillard, and Cardoso 2020) or on compilation to HTN planning (Höller et al. 2018). The Earley parser used in this paper was first proposed for HTN plan recognition by Pantůčková and Barták (2023).

The only other existing approach to HTN plan correction (Barták et al. 2021) corrects plans by action deletion. This parsing-based approach greedily composes from the bottom upward all abstract tasks that can be composed of the available actions using the decomposition rules from the domain model while deleting actions from the input plan that violate preconditions of the actions used in the composition. The algorithm exhaustively performs all such compositions until no new tasks can be composed. In contrast to the approach from this paper, the existing solver (Barták et al. 2021) can be used on partially ordered domain models. On the other hand, our approach supports both action insertion and action deletion; therefore, it can also be used for plans with missing (unobserved) actions in addition to plans with noisy observations. Our solver also demonstrates superior performance

if only action deletion is allowed, as top-down parsing in our solver leads to fewer intermediate solutions than greedy bottom-up parsing.

Another weakly related problem is plan repair (Zaidins, Roberts, and Nau 2023), (Goldman, Kuter, and Freedman 2020), (Höller et al. 2020), which aims to make changes in a not-yet executed suffix of a previously created plan at a point of execution when conditions change and it is not possible to continue with execution of the plan. This problem differs from plan correction, which modifies an action sequence to obtain a valid hierarchical plan.

Aho and Peterson (1972) proposed an approach to find a minimum edit distance string in context-free grammars, which is based on a similar idea as our approach. They extend the given grammar by rewriting rules representing all types of corrections and then they use the Earley parser straightforwardly. However, the search progresses differently. The approach of Aho and Peterson (1972) fills the table of the original Earley parser (i.e., finds all possibilities how the input sequence can be corrected). Then, the decomposition with the fewest corrections is found in the table. In contrast, our search is based on a priority queue, which takes into account the number of corrections required in each intermediate solution. Therefore, the search can terminate earlier – when there are no states which could lead to a better solution than the one that has already been found. Moreover, our approach does not insert new rewriting rules into a grammar. In contrast to the work the work of Aho and Peterson (1972), we generate correction rules one by one when they are needed. While Aho and Peterson (1972) works exclusively with context-free grammars, we include also evaluation of constraints. Furthermore, our approach is lifted (tasks have attributes).

Correction by action deletion and insertion

We present a lifted HTN plan correction approach based on top-down parsing. By omitting constraints of decomposition rules in a totally ordered domain model, we obtain an abstraction to a context-free grammar (CFG). CFG is a formal grammar whose rewriting rules are $A \rightarrow \alpha$, where A is a non-terminal symbol and α is a sequence of symbols (terminal and non-terminal). We propose a totally ordered HTN plan correction algorithm inspired by the Earley parser (Earley 1970), a top-down CFG parser. In contrast to (Barták et al. 2021), where parsing progresses from the bottom (from the available actions) upwards, the Earley parser progresses from the possible top-level tasks downwards to actions. Top-down parsing provides a considerable advantage for action insertion as it allows one to determine which actions must be generated to decompose candidate goals.

Earley parser starts by decomposing all possible top-level tasks into subtasks and then decomposing these abstract tasks until the actions are reached. The parser is based on dynamic programming. The input sequence is parsed from the left to the right as the parser systematically processes states by the index of the last symbol covered in the input sequence. The systematic left-to-right approach is unsuitable for optimal plan correction since we need to process states based on the number of corrections and not the end index. Therefore, we use a priority queue instead of a table. In addition to the priority queue, we also need to remember a set of all states that have been generated. For each state, we define its cost as the minimum number of corrections that will be done in the plan if the rule is used. The priories of enqueued states correspond to their costs, i.e., states from the queue are dequeued based on their cost (states with the lowest cost are dequeued first).

In the plan correction setting, the actions can be selected from the input plan or inserted into the plan. Successful lower-level decompositions can be used to complete higher-level decompositions until a decomposition of a top-level task is found.

The priority queue contains states of the following form:

$$s = (T_1 \rightarrow T_2 ... T_r \bullet T_s ... T_t, i, j).$$

This state represents a decomposition rule, which decomposes task T_1 into subtasks $T_2, ..., T_t$. Symbol $\bullet$ separates subtasks that have been already successfully decomposed (i.e., some lower-level rules decomposing $T_2, ..., T_r$ have been completely decomposed into actions) from subtasks that still remain to be decomposed. Each state is supposed to represent a possible coverage of a continuous subsequence of actions from the input plan; i is the index of the last action covered before this state (i.e., i indicates where in the input plan the decomposition of the first subtask of this rule should start) and j is the index of the last action covered by this state so far (i.e., the last action into which the last task before $\bullet$ decomposes).

The plan correction algorithm starts by enqueueing a state

$$(I \rightarrow \bullet T, 0, 0)$$

for each abstract task T, where I is a dummy starting task. Therefore, we do not need to know the goal task for which the input plan was generated as we take into account all possible goals. All these states have cost equal to zero, corresponding to zero corrections introduced so far.

The Earley parser defines three procedures for processing states of three different types: *completer*, *predictor* and *scanner*. **Scanner** processes states, where the first task, that has not been decomposed yet, is an action. For a state

$$s = (T_1 \rightarrow T_2 ... \bullet T_m ..., i, j),$$

where T_m is an action, we can create multiple new states. The corresponding action can be either taken from the input plan, where some actions from the plan may have to be skipped, or the action is inserted into the plan. If there is action a_{j+1} in the input plan unifiable with with (partially) instantiated primitive task T_m, we create a new state (with $\bullet$ shifted after T_m)

$$s' = (T_1 \rightarrow T_2 ... T_m \bullet ..., i, j+1).$$

The state s' will be enqueued into the priority queue with the priority equal to $priority(s)$.

The possible corrections that can be generated by a scanner state $(T_1 \rightarrow T_2 ... \bullet T_m ..., i, j)$ are the following (k can also be zero):

- delete k actions $(a_{j+1}, ..., a_{j+k})$ from the input sequence and unify action a_{j+k+1} with T_m with $cost = k$;

- delete k actions $(a_{j+1}, ..., a_{j+k})$ from the input sequence and insert a new action T_m after the deleted actions (between a_{j+k} and a_{j+k+1}, if a_{j+k} was not the last action in the input sequence) with $cost = k + 1$.

A scanner state will generate one correction at a time and enqueue itself into the priority queue again.

Let d be the number of deleted actions. If the action for T_m was selected from the plan, we will generate a new state

$$s'' = (T_1 \rightarrow T_2 ... T_m \bullet ..., i, j + d + 1),$$

whose priority will be equal to $priority(s)+d$ (d corrections introduced). If the new action had to be inserted into the plan, the priority of the new state

$$s''' = (T_1 \rightarrow T_2 ... T_m \bullet ..., i, j + d)$$

will be equal to $priority(s) + d + 1$.

Our approach does not require a grounded domain as an input. At the beginning, the priority queue is filled with the artificial initial starting rules for all possible uninstantiated abstract tasks. When scanner selects an action from the input plan, its variables will be propagated upwards into the newly created states. When a new action is inserted into the plan, it will be (partially) instantiated with variables propagated downwards from partially instantiated rules (by predictor).

When a scanner state is dequeued, we perform the next correction (the correction with the lowest priority) and enqueue the scanner state back into the queue with an increased priority and with the information which correction should be performed next.

Predictor is used to process states where the first task that has not been decomposed yet is an abstract task. For a state

$$s = (T_1 \rightarrow T_2 ... \bullet T_m ..., i, j),$$

where T_m is an abstract task, we create a state

$$s' = (T_m \rightarrow \bullet T_o^m ... T_p^m, j, j)$$

for each decomposition rule decomposing the task T_m. As the priority of states in the queue should be equal to the minimum number of corrections, we will enqueue s' with the priority equal to the priority of s as if the state s' is used in a decomposition, the state s must also have been used before. If the same state as s' is generated later by another state s'' with a priority lower than s, the priority of s' will be updated to the priority of s''. If the state s' already exists, we do not enqueue it again; we only update its priority.

Completed states (states, where all subtasks have been decomposed) are processed by **completer**. A state

$$s = (T_1 \rightarrow T_2 ... T_m \bullet, i, j)$$

represents a possible decomposition of the task T_1 covering actions $a_{i+1}, ..., a_j$. This state can be used to decompose the task T_1 whose decomposition should start by a_{i+1}. Therefore, for each state (containing T_1 right after $\bullet$)

$$s' = (T_0 \rightarrow ... \bullet T_1 ..., k, i)$$

we create a new state (with shifted $\bullet$)

$$s'' = (T_0 \rightarrow ... T_1 \bullet ..., k, j).$$

The number of corrections in s'' will be equal to the number of corrections in s' + the number of corrections in s. The priority of s'' will be equal to its number of corrections + the minimum priority of the predictor states that generated the decomposition of T_0. If the state s'' already exists, we will remember the new relation between the completing and completed state (this information will be used later to build a decomposition tree for a candidate solution) and recompute its priority. Instead of adding the number of corrections in s, we will use the minimum number of corrections of all states providing a complete decomposition of T_1.

Each state

$$(I \rightarrow T \bullet, 0, j),$$

covering a prefix of the plan up to the index j represents a candidate top-level task (actions starting with a_j, if any, are supposed to be deleted from the input plan). When the queue does not contain a state with a priority lower than the priority of the best solution found so far, the algorithm stops as no better candidate top-level task can be found. The algorithm may be terminated earlier and return the best candidate top-level task found so far. Hence, the algorithm can be seen as an anytime technique.

During search, completer builds an AND/OR tree for each candidate top-level task, where OR-nodes are abstract tasks, which can be decomposed by multiple decomposition rules, AND-nodes are decomposition rules, which decompose the task into the given subtasks, and leaves are actions. A completer links a decomposition rule to an abstract subtask in another decomposition rule. To extract a solution from an AND/OR tree, it needs to be checked whether we can choose one decomposition rule in each OR-node and ground all ungrounded variables in all nodes of the tree such that preconditions of all actions and all constraints of all decomposition rules are satisfied (with respect to the initial state and to state transitions after each action). We traverse AND-OR trees in a DFS-like manner.

Example 2. *Consider the HTN from Figure 1, let the initial state define all roads as road(loc1, loc2), road(loc2, loc3) and road(loc4, loc3) and let pickup(pkg1, loc1), drive(loc1, loc2), drive(loc4, loc3), drop(pkg1, loc3) be the invalid input plan. At the beginning, our algorithm will enqueue two states into an empty priority queue (assuming that there are only two abstract tasks in the domain):*

$$[(I \rightarrow \bullet deliver(?, ?), 0, 0), priority = 0] \quad (1)$$
$$[(I \rightarrow \bullet get\_to(?), 0, 0), priority = 0].$$

When state (1) is dequeued from the queue, predictor will create one new state:

$$[(deliver(?, ?) \rightarrow \bullet pickup(?, ?), get\_to(?), drop(?, ?), 0, 0), priority = 0]. \quad (2)$$

State (2) will be processed by scanner. As the first action in the input plan can be used as the desired pickup action, a new state can be created without any corrections:

$$[(deliver(pkg1, ?) \rightarrow pickup(pkg1, loc1), \bullet get\_to(?), drop(pkg1, ?), 0, 1), priority = 0]. \quad (3)$$

For state (3), predictor will create a new state based on available decomposition rules. The decomposition of the new state should start by the second action in the input plan, as the first action is already covered:

$$[(get\_to(?) \rightarrow \bullet drive(?,?), drive(?,?), 1, 1), priority = 0]. \quad (4)$$

Again, scanner can use the second action in the plan to cover the first subtask in state (4) and create the next state:

$$[(get\_to(?) \rightarrow drive(loc1, loc2), \bullet drive(loc2, ?), 1, 2), priority = 0]. \quad (5)$$

However, the next action in the plan drive(loc4, loc3) is not unifiable with drive(loc2, ?). The desired action can be inserted into the plan and so the scanner will insert drive(loc2, ?) before drive(loc4, loc3). Since the resulting state required 1 correction, its priority value will be higher than the priority of state (5):

$$[(get\_to(?) \rightarrow drive(loc1, loc2), drive(loc2, ?)\bullet, 1, 2), priority = 1]. \quad (6)$$

State (6) will be processed by completer and used to complete the next subtask of state (3):

$$[(deliver(pkg1, ?) \rightarrow pickup(pkg1, loc1), get\_to(?), \bullet drop(pkg1, ?), 0, 2), priority = 1]. \quad (7)$$

The next action of state (7) is again not unifiable with the next action in the input plan (drive(loc4, loc3)). Scanner will firstly insert the required action drop(pkg1, ?) into the input plan before drive(loc4, loc3), thus increasing the number of corrections, and enqueue the new state:

$$[(deliver(pkg1, ?) \rightarrow pickup(pkg1, loc1), get\_to(?), drop(pkg1, ?)\bullet, 0, 2), priority = 2]. \quad (8)$$

As another possible correction, the scanner can skip the action drive(loc4, loc3) and select the next action from the plan. Therefore, it will enqueue again the state 7 with priority equal to 2:

$$[(deliver(pkg1, ?) \rightarrow pickup(pkg1, loc1), get\_to(?), \bullet drop(pkg1, ?), 0, 3), priority = 2]. \quad (9)$$

Completer can then use the completed state (8) to complete the state 1 and create a candidate top-level rule:

$$[(I \rightarrow deliver(pkg1, ?)\bullet, 0, 2), priority = 2]. \quad (10)$$

We will then attempt to extract a solution from state (10). The missing variable (the location to which the package pkg1 will be delivered) can be chosen arbitrarily from the set of the locations to which the truck can drive from loc2. However, the resulting plan will require two more corrections (four corrections in total) as the last two actions from the input plan will be deleted. As there is still a state with fewer than four corrections (state (9)), the algorithm continues to find a better correction.

Scanner will then process state (9) and create a new state, where the action drive(loc4, loc3) will be deleted from the plan and the last action in the plan will be covered:

$$[(deliver(pkg1, ?) \rightarrow pickup(pkg1, loc1), get\_to(loc3), drop(pkg1, loc3)\bullet, 0, 4), priority = 2]. \quad (11)$$

State (11) will be processed by completer and another candidate top-level rule will be created:

$$[(I \rightarrow deliver(pkg1, loc3)\bullet, 0, 4), priority = 2]. \quad (12)$$

State (12) can be used to extract the desired valid plan, which can be created from the input plan by two corrections: deleting the third action from the plan and inserting a different action into its position.

Theorem 1. *The plan correction algorithm is sound.*

Sketch of proof. The correctness is implied by the soundness of the Earley algorithm. The modified Earley algorithm builds AND/OR trees representing all possible decompositions and then the algorithm selects in the internal nodes the decomposition rules whose constraints are satisfied and in the leaves actions which are executable with respect to their preconditions and effects. □

Theorem 2. *The plan correction algorithm is complete.*

Sketch of proof. We will first prove that the algorithm terminates. Each *scanner* state can only generate a finite number of descendant states (with nondecreasing costs), where the last state is created by the insertion of the requested action after the sequence of observed actions (skipping all preceding actions).

As *completer* and *predictor* procedures work similarly as in the original Earley parser, finiteness of the Earley parser implies finiteness of the HTN plan correction algorithm. Let us note that finiteness is guaranteed even for models with recursive decomposition rules as predictor does not create and enqueue states that already exist. If the requested descendant state ($T \rightarrow \bullet$..., with the suitable starting index) has already been created by another predictor, predictor will not generate the state again, preventing infinite recursion. This is one of the basic principles of the original Earley parser. E.g., if there are two rules $T_1 \rightarrow T_2$ and $T_2 \rightarrow T_1$, the predictor state $s = T_1 \rightarrow \bullet T_2$ will generate a new state $s' = T_2 \rightarrow \bullet T_1$, but when s' is processed, s will not be generated again.

Therefore, if there does not exist any plan that can be generated by the rules available in the domain model, the plan correction algorithm will terminate when the queue is exhausted. If a valid plan exists, the algorithm is guaranteed to find it (which is implied by the completeness of the original Earley parser), in the worst case by deleting all input actions and inserting new ones. In this case, the shortest possible plan will be found as the algorithm terminates when a plan with a lower cost cannot be found.

□

Theorem 3. *The plan correction algorithm is optimal.*

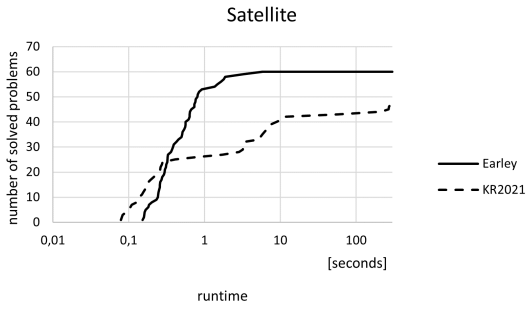


Figure 2: The number of problems solved within given time (logarithmic x-axis) in the domain Satellite. *KR2021* is the solver from (Barták et al. 2021), *Earley* is our solver.

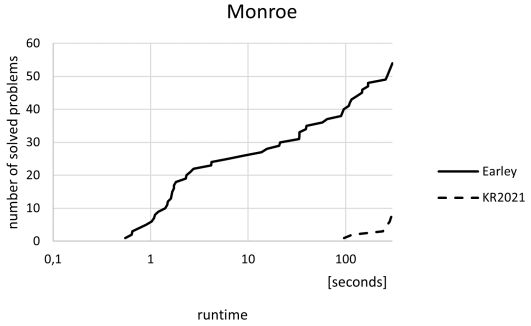


Figure 3: The number of problems solved within given time (logarithmic x-axis) in the domain Monroe. *KR2021* is the solver from (Barták et al. 2021), *Earley* is our solver.

Sketch of proof. For each state, we know the minimum number of flaws of a solution that can be extracted from this state. The algorithm halts and returns a plan only if there remain no untried top-level states which could yield a plan with a lower cost and if there are no states with a lower cost in the queue. As a state with cost c can only contribute to solutions with a cost $c' \geq c$, the solution is optimal. □

Empirical evaluation

The experiments were executed on a computer with the Intel Core i7-8550U CPU @ 1.80GHz processor and 16 GB of RAM. Maximum allowed runtime was set to five minutes for one problem. For the experiments, we assumed that the root task is not known, which is the same setting which was used for the plan correction by action deletion approach (Barták et al. 2021). This setting corresponds to HTN plan recognition with partial observability and noise.

We used domains and plans from the International Planning Competition (IPC) 2020. The valid plans consisted of 9 – 28 actions in the domain Satellite, 18 – 71 in Transport, 15 – 68 in Monroe and 22 – 168 in Blocksworld. For the task of correcting plans by action deletion, we added noise to the plans by inserting extra actions into valid plans. For plan correction by action insertion, we deleted some actions from

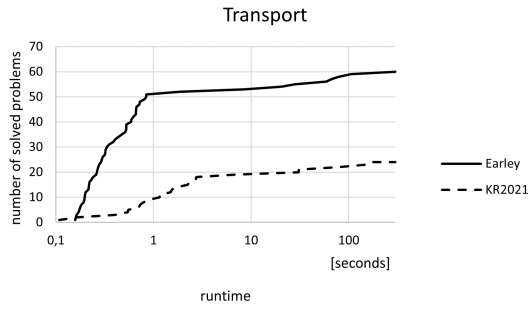


Figure 4: The number of problems solved within given time (logarithmic x-axis) in the domain Transport. *KR2021* is the solver from (Barták et al. 2021), *Earley* is our solver.

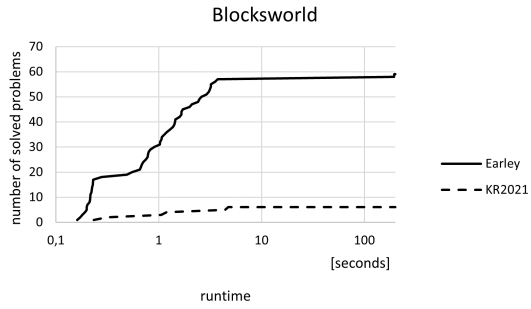


Figure 5: The number of problems solved within given time (logarithmic x-axis) in the domain Blocksworld. *KR2021* is the solver from (Barták et al. 2021), *Earley* is our solver.

valid plans. Let us note that the solution does not have to be the original valid plan as there may be a different valid plan which can be obtained from the modified plan by a lower or equal number of corrections. The resulting plans used for experiments are accessible on-line[1].

Correcting plans by action deletion

We have compared the performance of our approach with the approach of Barták et al. (2021). As this approach supports only action deletion, we have compared it with our approach also restricted to action deletion.

We have run both solvers on valid plans and on invalid plans, which were created from valid plans by inserting at least one and at most five extra actions. Adding more noise to the plans does not seem beneficial as it would lead the solvers to longer valid plans that could be achieved by deleting fewer actions. In the domain Blocksworld, the solvers already found valid plans which were longer than the original plan by one or two actions.

Figures 2, 4, 3 and 5 show how the total number of problems grows with runtime. Each figure shows results from a different domain. Each domain contains 60 plans: 10 different valid plans and 50 plans created by inserting extra ac-

[1] https://github.com/krpant/Plan-correction-benchmarks

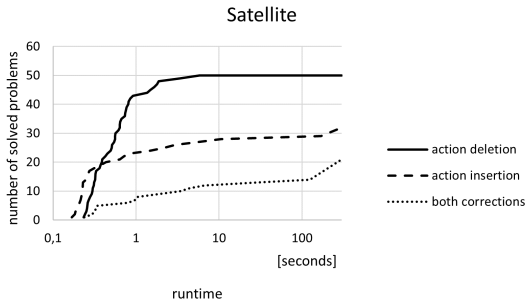


Figure 6: The number of problems solved within given time (logarithmic x-axis) from the domain Satellite using action deletion or insertion or by both means of correction.

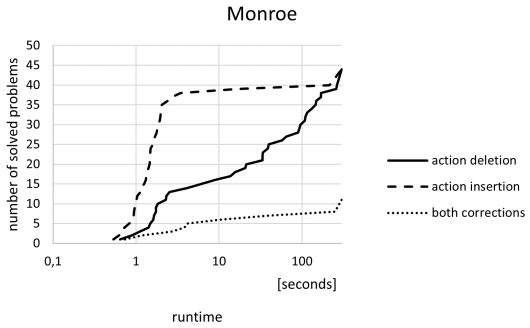


Figure 7: The number of problems solved within given time (logarithmic x-axis) from the domain Monroe using action deletion or insertion or by both means of correction.

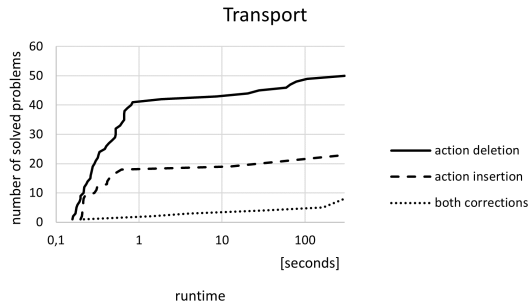


Figure 8: The number of problems solved within given time (logarithmic x-axis) from the domain Transport using action deletion or insertion or by both means of correction.

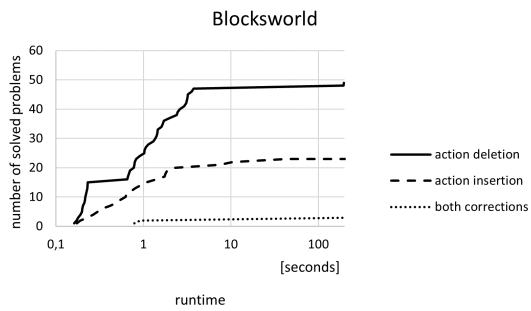


Figure 9: The number of problems solved within given time (logarithmic x-axis) from the domain Blocksworld using action deletion or insertion or by both means of correction.

tions into the valid plans. The graphs consider the runtime after which the computation stopped and provided the correct plan. Solver runs which found the correct solution but did not terminate before the time limit has passed are displayed with runtime equal to the time limit. Our approach outperformed the approach of Barták et al. (2021) on all four domains.

Correcting plans by action deletion and insertion

Action deletion vs action insertion Additional experiments were performed to assess how demanding is action insertion in comparison to action deletion. We have created another 50 invalid plans for each domain by deleting at least one and at most five actions from valid plans and we have measured how fast our solver corrected these plans solely by action insertion.

domain	ins	ins*	del	del*
Satellite	32	19	50	50
Transport	23	19	50	49
Monroe	44	40	44	40
Blocksworld	23	23	47	47
total	122	101	193	188

Table 1: Number of solutions found within 5 minutes by action insertion (*ins*) and deletion (*del*) and number of solutions where the optimality was proven(*ins** and *del**).

Figures 6, 8, 7 and 9 show how fast our solver solves action insertion problems in comparison to action deletion problems and problems of plan correction by action insertion and deletion simultaneously. Table 1 compares the number of problems solved within the given time limit solely by action insertion or deletion, considering separately all solutions with the expected number of corrections which were found within the time limit, and solutions where also the proof of optimality was found (i.e., the solver terminated before the time limit has passed). In general, action insertion seems to be more demanding than action deletion. In the domains Satellite, Transport and Blocksworld, the solver corrected significantly less plans by action insertion than by action deletion within the given time limit.

On the contrary, in the domain Monroe the solver eventually arrived to the same result for both types of problems and most action insertion problems were even solved faster than the action deletion problems. The domain Monroe defines a complex hierarchy of object types and a variety of specialized decomposition rules, while the plans from the other three domains consist of sequences of rather simple tasks

which can be composed using many different combinations of objects. Therefore, we assume that in these three domains it was more difficult to find the missing actions as there were too many possible actions that could be generated in order to complete the rules of the parser. On the other hand, plan correction with only action deletion limits completed rules to those that can be completed by existing actions.

In the domain Monroe, however, the number of possible decompositions of a rule was significantly lower; therefore, the solver was guided by the domain model to insert only the most relevant actions. Action deletion could then require more time simply because the input plan was longer, which resulted in more completed subtrees which could not be used to complete the decomposition of a top-level task.

Figure 10 shows how the number of problems from the domain Transport solved by different numbers of corrections grows with runtime. The figure consider the runtime after which the computation stopped and provided the correct plan. Solver runs, which found the correct solution but did not terminate before the time limit has passed, are displayed with runtime equal to the time limit. As expected, plans requiring more corrections were usually more difficult to correct. The difference was more noticeable on action insertion.

Both corrections simultaneously To assess the performance of the solver when both means of correction are enabled simultaneously, we have created more invalid input plans by deleting one or two actions and inserting one or two actions into valid plans; therefore, we have generated four invalid plans for each valid plan. However, for some plans it was possible to find a valid plan by less corrections than intended. Figures 6, 8, 7 and 9 show how the number of problems solved solely by action deletion or insertion or by both means of correction grows with runtime. The graphs consider the runtime after which the computation stopped and provided the correct plan. Solver runs which found the expected solution but did not terminate before the time limit has passed are displayed with runtime equal to the time limit. Unsurprisingly, the solver runs slower as more rules are created by the scanner. The solver was most successful in the domain Satellite, which contains in general the shortest plans. In this domain, the number of solved problems was close to the number of problems solved solely by action insertion. In the other domains, the solver corrected significantly less plans than by only one mean of correction. Even in the domain Monroe, where the complexity of both means of correction seems to be similar, the performance decreased considerably when both means of correction were enabled.

Solution quality over time Our algorithm incrementally improves the quality of its result in order to be able to provide the best possible result any time it is terminated. Therefore, it attempts to extract a solution from each candidate top-level rule whenever such a rule is found that could provide a solution with fewer corrections than the best solution found so far. Often the optimal solution was found early, though the algorithm required much more time to dismiss candidate partial solutions with less corrections and therefore prove that the solution is optimal (see the difference

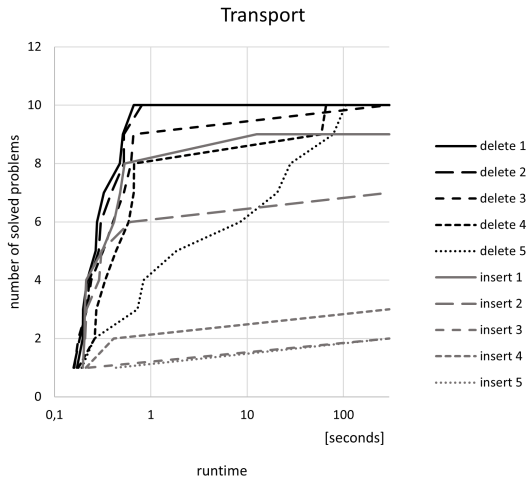


Figure 10: The number of problems solved withing given time (logarithmic x-axis) from the domain Transport using different means and numbers of corrections.

of solved problems with and without proof of optimality in Table 1). When only action insertion was allowed, the algorithm found for all problems in all domains only one complete solution. When action deletion was enabled, a sequence of solutions with improving quality was often found as more possible top-level tasks could be sometimes found simply by covering a prefix of the input plan and deleting the rest of the actions.

Conclusion

We propose a novel approach to correcting totally ordered HTN plans, the first approach facilitating both action deletion and action insertion. Furthermore, our approach outperforms the existing HTN plan correction approach when only action deletion is enabled. Future work may aim to enhance the efficiency of the solver when both means of correction are enabled, as the performance is better when only action deletion or insertion is allowed. Another possible research direction could focus on extending the algorithm to partially ordered HTN plans.

Acknowledgements

Research is supported by the Charles University, project GA UK number 156121, by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215 and by SVV project number 260 698.

References

Aho, A. V.; and Peterson, T. G. 1972. A minimum distance error-correcting parser for context-free languages. *SIAM Journal on Computing*, 1(4): 305–312.

Barták, R.; Maillard, A.; and Cardoso, R. C. 2020. Parsing-based Approaches for Verification and Recognition of Hierarchical Plans. In *The AAAI 2020 Workshop on Plan, Activity, and Intent Recognition*.

Barták, R.; Ondrčková, S.; Behnke, G.; and Bercher, P. 2021. Correcting hierarchical plans by action deletion. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, volume 18, 99–109.

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 International Conference on Automated Planning and Scheduling*, volume 27.

Earley, J. 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2): 94–102.

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

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 2020)*, 27–35.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2018. Plan and goal recognition as HTN planning. In *2018 IEEE 30th International Conference on Tools with Artificial Intelligence*, 466–473.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2020. HTN plan repair via model transformation. In *KI 2020: Advances in Artificial Intelligence: 43rd German Conference on AI, Bamberg, Germany, September 21–25, 2020, Proceedings 43*, 88–101. Springer.

Höller, D.; Wichlacz, J.; Bercher, P.; and Behnke, G. 2022. Compiling HTN plan verification problems into HTN planning problems. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 32, 145–150.

Lin, S.; Behnke, G.; Ondrčková, S.; Barták, R.; and Bercher, P. 2023. On total-order HTN plan verification with method preconditions–an extension of the CYK parsing algorithm. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, 12041–12048.

Pantůčková, K.; and Barták, R. 2023. Using Earley Parser for Recognizing Totally Ordered Hierarchical Plans. In *Proceedings of 26th European Conference on Artificial Intelligence (ECAI) 2023*, 1819–1826. IOS Press.

Pantůčková, K.; Ondrčková, S.; and Barták, R. 2024. Using Earley Parser for Verification of Totally Ordered Hierarchical Plans. *The International FLAIRS Conference Proceedings*, 37(1).

Zaidins, P.; Roberts, M.; and Nau, D. 2023. Implicit Dependency Detection for HTN Plan Repair. *Proceedings of the 6th ICAPS Workshop on Hierarchical Planning (HPlan 2023)*, 10–18.

Redundant Decompositions in PO HTN Domains: Goto Considered Harmful

Roland Godet[1,2], Arthur Bit-Monnot[1], Charles Lesire-Cabaniols[2]

[1]LAAS-CNRS, University of Toulouse, INSA, Toulouse, France
[2]ONERA/DTIS, University of Toulouse, France
roland.godet@laas.fr, arthur.bit-monnot@laas.fr, charles.lesire@onera.fr

Abstract

HTN planning is a widely used approach for solving planning problems by breaking them down into smaller sub-problems. This approach is often motivated by the ability to add constraints between tasks, which can guide the search towards a solution and improve performance by reducing the search space. In this paper, we identify a common pattern in PO HTN planning that can lead to a pathological explosion of the search space, resulting in a significant decrease in computational performance. However, this is not a fatal issue. Alternative HTN models can be used to reduce the search space. We propose two models that maintain the expressiveness of the original problem while reducing the number of possible decompositions. Our results demonstrate improved computational performance on IPC benchmarks.

Introduction

Task planning is a fundamental problem in Artificial Intelligence, with applications in robotics, logistics, and many other domains. The problem consists of finding a sequence of actions that completes a given goal, while respecting a set of constraints.

One of the approaches to planning is Hierarchical Task Network (HTN) planning (Erol, Hendler, and Nau 1994), where actions are hierarchically organized into tasks, which can be refined into subtasks or actions, and so on. This hierarchical structure allows the problem to be described at various levels of abstraction, ranging from highly abstract tasks to directly executable actions.

One common motivation for using HTN planning is the promise of increased performance as the hierarchy is expected to restrict the search space and guide the planner towards a solution. In the case of Partial Order (PO) HTN planning, where several tasks may interact in the achievement of their respective goals, we show that on the contrary, the hierarchy can be extremely detrimental to the search.

After a brief introduction to the HTN formalism, this paper identifies a pattern that is ubiquitous in PO HTN planning benchmarks, and that leads an explosion of the search space of PO HTN planners. While there is no general approach to solve this issue, we propose new models that reduce the number of possible decompositions, and show that they improve the computational performances on the International Planning Competition (IPC) 2020 HTN tracks.

HTN Planning

This section provides a brief introduction to the HTN planning problem as described by Höller et al. (2020).

An HTN planning problem can be notably described using the HDDL language (Höller et al. 2020), an extension of PDDL (McDermott et al. 1998), or the ANML language (Smith, Cushing, and Frank 2008).

Assume that $\mathcal{L} = (P, T, V, C)$ is a quantifier- and function-free first order predicate logic. T is finite set of type symbols. C is a finite set of typed constants. V is a finite set of typed variables. P is a finite set of predicate symbols, each associated to a list of parameter variables from V.

Definition 1 (State)**.** A state is the representation of the world at a given time, defined by a ground (variable-free) conjunction of literals over $\mathcal{L}$. The set of all possible states is denoted by S.

Definition 2 (Primitive Task)**.** A primitive task (or action) is an operation that can be executed directly, defined by the tuple $a = (name, pre, eff)$ where:

- *name* is its unique task name, a first-order atom such as `move(s, d)` consisting of the action name followed by parameters.
- *pre* is its precondition, a conjunction of first-order literals over $\mathcal{L}$.
- *eff* is its effect, a conjunction of first-order literals over $\mathcal{L}$. We split it into positive (eff^+) and negative (eff^-) effects.

Remark. We also refer to *pre* and *eff* as *pre(a)* and *eff(a)* when referring to a specific action a. All variables used in *pre(a)* and *eff(a)* must be parameters of the action. Finally, an action is said *ground* if all its parameters are constants from C.

Definition 3 (Executable Action)**.** Given a state $s \in S$, a ground primitive task a is executable in s if and only if its precondition is satisfied by s:

$$\xi(s, a) = s \models pre(a) \tag{1}$$

Definition 4 (State Transition)**.** Given a state s and an executable action a, the state transition function $\gamma(s, a)$ is the result of executing the action a in s. It is defined by the following formula:

$$\gamma(s, a) = \big(s \setminus \mathit{eff}^-(a)\big) \cup \mathit{eff}^+(a) \tag{2}$$

Remark. The extension of ξ and γ to a sequence of actions are defined recursively by:

$$\begin{cases} \xi(s, \langle a_1, \ldots, a_n\rangle) = \xi(\gamma(s, a_1), \langle a_2, \ldots, a_n\rangle) \\ \gamma(s, \langle a_1, \ldots, a_n\rangle) = \gamma(\gamma(s, a_1), \langle a_2, \ldots, a_n\rangle) \end{cases} \tag{3}$$

Definition 5 (Compound Task)**.** A compound task (or abstract task) is simply a task name, *i.e.*, a first-order atom such as `goto(p)` consisting of the actual task name followed by parameters.

Definition 6 (Task Network)**.** A task network over a set of task names X is a tuple $tn = (I, \prec, \alpha, C)$ where:

- I is a possibly empty set of task identifiers. They are used to distinguishing between tasks that occur multiple times in the task network.
- $\prec$ is a strict partial order over I.
- $\alpha : I \to X$ maps each task identifier to a task name.
- C is a set of constraints over the task parameters.

Remark. For easier comprehension, we will refer to a task network without constraints (*i.e.*, $C = \emptyset$) and with total ordered tasks as the set $tn = \{t_1 \prec \ldots \prec t_n\}$.

Definition 7 (Method)**.** A decomposition method represents a way to achieve a compound task. It is defined by the tuple $m = (c, tn)$ where c is the compound task name and tn is a task network describing the subtasks needed to achieve the compound task.

Definition 8 (Decomposition)**.** Decomposition is the process of replacing a compound task of a task network by another task network. Given a decomposition method $m = (c, (I_m, \prec_m, \alpha_m))$ and a task network $tn_1 = (I_1, \prec_1, \alpha_1)$ such that $I_m \cap I_1 = \emptyset$ (that can be done by renaming), the task network $tn_2 = (I_2, \prec_2, \alpha_2)$ is a decomposition of a task identifier $i \in I_1$ by m if and only if:

$$\begin{cases} \alpha_1(i) = c \\ \quad I_2 = (I_1 \setminus \{i\}) \cup I_m \\ \quad \prec_2 = (\prec_1 \cup \prec_m \\ \qquad \cup \{(i_1, i_2) \in I_1 \times I_m \mid (i_1, i) \in \prec_1\} \\ \qquad \cup \{(i_1, i_2) \in I_m \times I_1 \mid (i, i_2) \in \prec_1\}) \\ \qquad \setminus \{(i', i'') \in I_1 \times I_1 \mid i' = i \text{ or } i'' = i\} \\ \quad \alpha_2 = (\alpha_1 \cup \alpha_m) \setminus \{(i, c)\} \end{cases} \tag{4}$$

Definition 9 (Executable Task Network)**.** Given a state s and a ground task network tn, the task network tn is executable in s if and only if:

- the constraints of C are respected.
- there exists a sequence $\langle i_1, \ldots, i_n\rangle$ of its task identifiers, with $n = |I|$, respecting $\prec$ such that $\langle \alpha(i_1), \ldots, \alpha(i_n)\rangle$ is executable in s.

Definition 10 (Planning Domain)**.** A planning domain is a tuple $\mathcal{D} = (\mathcal{L}, T_P, T_C, M)$ where:

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

Definition 11 (Planning Problem)**.** A planning problem is a tuple $\mathcal{P} = (\mathcal{D}, s_I, tn_I, g)$ where:

- $\mathcal{D}$ is a planning domain.
- $s_I \in S$ is the initial state, a ground conjunction of positive literals over the predicates.
- tn_I is the initial task network.
- g is the goal, a first-order formula over the predicates.

Definition 12 (Solution)**.** Given a planning problem $\mathcal{P} = (\mathcal{D}, s_I, tn_I, g)$, where $\mathcal{D} = (\mathcal{L}, T_P, T_C, M)$, a task network $tn_S = (I_S, \prec_S, \alpha_S)$ is a solution of $\mathcal{P}$ if and only if:

- there is a sequence of decompositions from tn_I to $tn = (I, \prec, \alpha)$, such that $I = I_S$, $\prec \subseteq \prec_S$, and $\alpha = \alpha_S$.
- tn_S is executable in s_I and its execution leads to a state s such that $s \models g$.

Motivating Example

Let us consider a simple navigation problem consisting of a truck that must go to a given position. The aim for the truck is to go from the position `p1` to the position `p5` using the roads defined in Figure 1.

Problem Formalization

Let us first correctly formalize this problem.

Predicate Logic The predicate logic is defined by the tuple $\mathcal{L} = (P, T, V, C)$ where:

- $T = \{\texttt{T}, \texttt{P}\}$, `T` represents a truck and `P` a position.
- C is composed of one truck `t1` and five positions from `p1` to `p5`.
- $P = \{\texttt{at(t, p)}, \texttt{road(s, d)}\}$, with `at` representing that the truck `t` is at the position `p` and `road` the existence of a road between the positions `s` and `d`.
- V is the set of variables appearing in the next defined actions, tasks, and methods.

Primitive Tasks There are two primitive tasks:

- `move(t, s, d)`, that moves the truck `t` from the position `s` to the position `d` if there is a road.
  - *pre(move)* = `at(t, s)` $\wedge$ `road(s, d)`
  - *eff(move)* = `at(t, d)` $\wedge \neg$`at(t, s)`
- `noop(t, d)`, that does nothing and is only applicable if the truck `t` is at the position `d`.
  - *pre(noop)* = `at(t, p)`
  - *eff(noop)* = $\emptyset$.

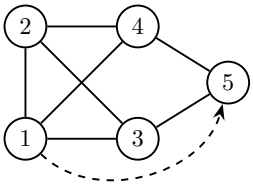


Figure 1: Graph of the navigation problem. The truck can move from one position to another using the roads. It is initially at the position `p1` and must go to the position `p5`.

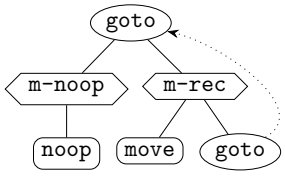


Figure 2: Hierarchical model of the `goto` task. There are two methods to achieve it: (i) do nothing if the truck is already at the given position, (ii) `move` to a nearby position, and then `goto` the given position again.

Compound Tasks & Methods We define a single compound task, `goto(t, d)`, that recursively moves the truck `t` to the position `d`. Two methods can achieve this task as shown in Figure 2:

- do nothing if the truck is already at the given position: `m-noop` = {`noop(t, d)`}.
- move to a nearby position, and then go to the given position again:
  `m-rec` = {`move(t, s, n)` $\prec$ `goto(t, d)`}.

Domain The domain is simply defined by:
$\mathcal{D} = (\mathcal{L}, \{\texttt{move}, \texttt{noop}\}, \{\texttt{goto}\}, \{\texttt{m-rec}, \texttt{m-noop}\})$.

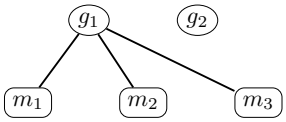
Problem The initial state is defined such that the truck is at the position `p1` and the roads match the graph in Figure 1:
s_I = `at(t1, p1)` $\wedge$ `road(p1, p2)` $\wedge \ldots$
The initial task network is composed of n identical and unordered `goto(t1, p5)` tasks, without constraints. Finally, the problem is defined by $\mathcal{P} = (\mathcal{D}, s_I, tn_I, \emptyset)$.

Pattern Identification

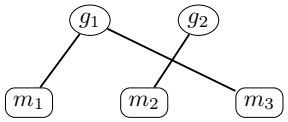
This recursive representation of the `goto` task is common in hierarchical planning community. It is found in many domains, such as *Factory*, *Transport*, or *Minecraft Player* of the HTN track of the IPC.

Consider the problem with three `goto(t1, p5)` in the initial task network with the respective identifiers g_1, g_2, and g_3. In essence this means that three different tasks request the objective of bringing the truck to `p5`.

The shortest solution is composed of two `move` actions m_1 and m_2, for instance going through `p3` with m_1 : `move(t1, p1, p3)` and m_2 : `move(t1, p3, p5)`. Note that we do not explicitly consider `noop` actions in the solution as they are artifact of the hierarchical encoding to break the recursion and could be replaced by method preconditions in the HDDL formalism.



(a) g_1 decomposes all actions.



(b) Decomposition is shared.

Figure 3: Considering two identical `goto` tasks (g_1, g_2), and an optimal solution with three `move` actions. Two possible decompositions of the initial task network that lead to the same solution plan. There are 8 possible decompositions in total.

In order for this $\langle m_1, m_2 \rangle$ action sequence to be considered a solution, it must be decomposable from the initial task network. Intuitively, each of the three (indistinguishable) top level tasks can be decomposed into this sequence. Interestingly, in the absence of ordering constraints, m_1 and m_2 may be decomposed from different top-level tasks. For instance, we may have for instance g_1 moving the truck from `p1` to `p3` and g_3 from `p3` to `p5`.[1] This situation is illustrated in Figure 3.

Let us now consider the number of decomposition paths that lead to this particular optimal solution. The planner will first decompose a `goto` task g_i ($i \in \{1, 2, 3\}$) into the `move` action m_1 and contribute a fresh `goto` task g_4 to the task network. Then, it will decompose a `goto` task g_j ($j \in \{1, 2, 3, 4\} \setminus \{i\}$) into the `move` action m_2 and another `goto` task g_5. Finally, it will decompose all `goto` task g_k ($k \in \{1, 2, 3, 4, 5\} \setminus \{i, j\}$) into the `noop` action.

Because at each step the planner can choose any task to decompose, the number of possible decompositions depends on the number of tasks in the task network. In our example, each `move` action can be decomposed from any of the three `goto` tasks, resulting in $3^2 = 9$ possible decompositions.

This result is trivially generalizable to n `goto` tasks and k `move` actions, resulting in n^k possible decompositions. For instance, in the case of six `goto` tasks and ten `move` actions, the planner must consider $6^{10} = 60466176$ possible decompositions.

HTN planners typically explore the set of possible decompositions of the initial task network until one is found that is both primitive and executable (*i.e.*, a solution). As a result this redundancy of decomposition paths is likely to translate as a redundancy in the search space of HTN planners.

Empirical Tests

To illustrate the impact of this pattern on the computational performance of a planner, we conducted a simple

[1] This only requires the `noop` actions to appear last in the plan, which is not prevented by any ordering constraints.

experiment on the domain of the example.

We used the PandaPi planner (Holler 2023), a state-of-the-art PO HTN planner, on the IPC 2023 HTN track, without timeout on 10 instances. For the i-th instance, we considered i identical `goto(t1, p5)` tasks in the initial task network. For each instance, the shortest plan contains exactly two `move` actions and i `noop` actions.

The results are shown in Figure 4. The five first instances are solved in less than 1 second, while the last instance is solved in 15 minutes.

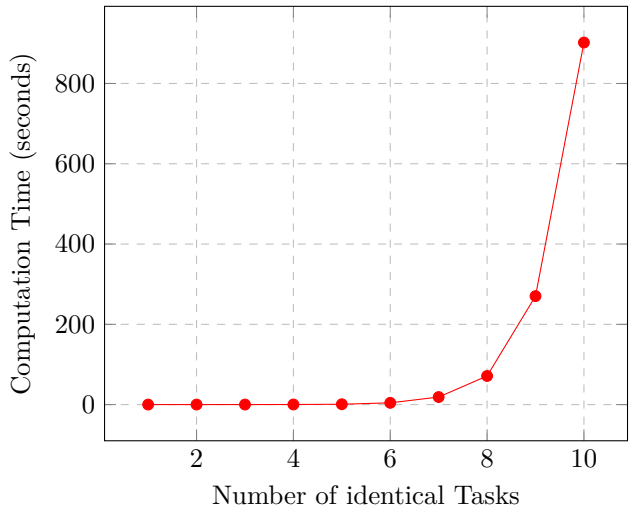


Figure 4: Planning time of PandaPi as a function of the number of identical `goto` tasks in the initial task network. In all cases, the shortest plan requires 2 `move` actions.

Representativity of the Use Case

While this pattern may appear artificial, we argue that is in fact pervasive in PO HTN planning benchmarks. Consider the logistic problem of moving packages from one location to another, as in the *Transport* domain of the IPC 2020 HTN track.

This is generally represented with a `deliver` task whose main method (i) move the truck to the package, (ii) load the package onto the truck, (iii) move the truck to the destination, and (iv) unloads the package at the destination. The first and third steps are typically done with a `goto` task similar to the one we showed.

If we consider the problem of moving three packages initially at location `X` to a location `Y` and assume that the truck has sufficient capacity to hold them all, the optimal plan would involve moving the truck to `X`, loading all packages in `X`, moving the truck to `Y` and unloading package at `Y`. Exactly as in our motivating example, the actions necessary to move the truck from `X` to `Y` could be decomposed from the three `goto` tasks introduced by the step *(iii)*.

This is representative of sharing common steps of the plan among potentially concurrent top level tasks. While the initial task network would likely never involve the same task multiple times, the decomposition of the common steps of the plan would lead to the same issue as the one we identified in our use case.

New Models

As shown in the previous section, the hierarchical model of the `goto` task shown in Figure 2 results in an exponential number of possible decompositions. This negatively impacts the computational performance of the planner, making the problem intractable for large instances.

This section describes two alternative models to the one shown in Figure 2, but with a smaller number of possible decompositions.

Mutex Decomposition

The explosion in the number of possible decompositions is primarily due to potential interference between several tasks for the purpose of producing a sequence of actions, as illustrated in Figure 3b. The idea in this alternative model is to forbid such interference by (i) forcing each `goto` task to lock the truck before producing any `move`, and (ii) only allowing it to release the lock once it has reached its target.

To achieve that, the `goto` task of the previous model is renamed to `goto-exec` and a new `goto(t, d)` compound task is added. The aim of the renamed `goto-exec` task is to force the planner to completely decompose the `goto-exec` task into a sequence of `move` actions that reaches its target `d`. The new `goto` task is used to keep the expressiveness of the domain unchanged.

To ensure that only one `goto-exec` task is decomposed at a time, a mutex predicate `mutex(t)` is added to P. This mutex predicate is manipulated by two new actions, `set-mutex` and `release`:

- `set-mutex(t)` sets the mutex predicate.
  - *pre*(`set-mutex`) = ¬`mutex(t)`
  - *eff*(`set-mutex`) = `mutex(t)`
- `release(t)` releases the mutex predicate.
  - *pre*(`release`) = `mutex(t)`
  - *eff*(`release`) = ¬`mutex(t)`

Finally, the `goto` task can be decomposed by two methods:

- do nothing if the truck is already at the given position: `m-noop` = {`noop(t, d)`}.
- set the mutex predicate, effectively go to the given position, and release the mutex predicate: `m-goto` = {`set-mutex` ≺ `goto-exec` ≺ `release`}.

The final hierarchy of the domain is shown in Figure 5.

The two new actions are used to ensure that the truck will not be moved by another `goto-exec` task while it is already moving. Because a precondition of the `noop` action is for the truck to be at the given position, the

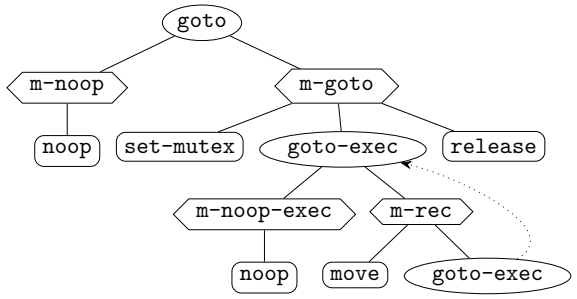
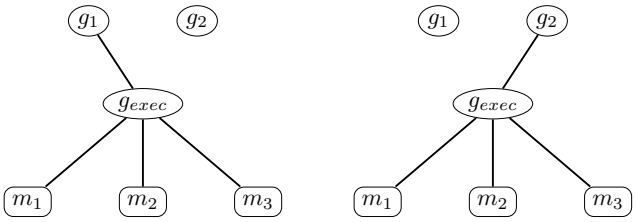


Figure 5: Mutex model of the `goto` task.



(a) g_1 makes the execution. (b) g_2 makes the execution.

Figure 6: Considering two identical `goto` tasks, and an optimal solution with three `move` actions. There are only 2 possible decompositions in total.

mutex will be released only when the truck is effectively at the given position. Therefore, the next `goto` tasks will immediately be decomposed into the `noop` action if it was identical to the previous one.

With this model, the expressiveness of the domain is kept unchanged and, modulo the mutex actions, the set of solutions is the same.

Consider our motivating example of n identical `goto` tasks that should be used to produce a sequence of k `move` actions. Here, the number of possible decompositions is in $\mathcal{O}(n)$: the planner may only choose which of n `goto` tasks to use to produce the full sequence, as illustrated in Figure 6.

(Partial) Task Insertion

As the root of the problem comes from the fact that the `move` actions may be contributed by several concurrent tasks in the initial task network, one solution may be to decouple the introduction of the `move` actions from the top-level objective tasks.

One way to do this would be by adopting the task-insertion variant of HTN planning, where primitive tasks may be introduced independently of any top-level task at arbitrary points of the solution (Alford, Bercher, and Aha 2015). If we were to discard the `goto` task entirely, this would allow the `move` actions to be introduced on-demand to establish the preconditions of actions requiring the truck to be at a given location. Task-insertion however is not without tradeoffs as the freedom of inserting arbitrary tasks in the plan nullifies the ability of HTN models to restrict the set of admissible solutions.

Instead, we propose to mimic the notion of *task-dependency* of FAPE (Bit-Monnot et al. 2020). FAPE distinguishes *task-dependent* actions that may only be introduced through a decomposition and *task-independent* actions that may also be inserted independently of the hierarchy. This enabled FAPE to consider a continuum between generative and HTN planning, where only a subset of the actions are allowed to be the subject of task insertion.

In this model we want to reproduce a similar behavior, where (i) the `move` actions can be inserted arbitrarily, and (ii) a `goto(t, p)` task only imposes a condition that `at(t, p)` holds.

To encode this in a conventional (*i.e.*, without task-insertion or *task-dependency* concepts) HTN model, we introduce a new compound task `free-move(t)` that can be decomposed with two methods:

- `m-move` = {`move(t, s, d)` $\prec$ `free-move(t)`}, that moves the truck between two arbitrary positions `s` and `d` and repeats. Here, `s` and `d` are unconstrained parameters of the method.
- `m-stop-free` = {}, which ends the recursion.

To ensure that the freedom of movement is effective, it is necessary to include the `free-move` task in the initial task network for each truck object. For our example, `free-move(t1)` is added in the initial task network.

To avoid relaxing its post-conditions, the `goto` task is kept in the domain but can only be decomposed by the single method `m-noop`. Because this method contains only the `noop` action, the `goto` task will be decomposed into the `noop` action, effectively doing nothing but checking if the truck is at the required position.

The final hierarchy of the domain is shown in Figure 7.

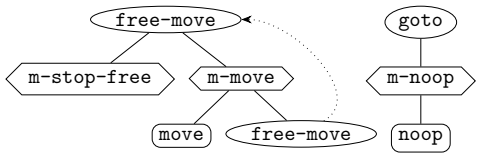


Figure 7: Task insertion model of the `goto` task.

Because each `goto` task is decomposed into a single action `noop`, the number of possible decompositions of n tasks `goto` is constant. Moreover, the number of possible decompositions of the `free-move` task is also constant because it is the only one introducing the sequence of `move` actions needed to achieve the optimal plan and verifying the preconditions of the `goto` tasks. Therefore, the number of possible decompositions for this optimal action sequence is constant and equal to 1.

With this model, the set of solutions differs as the model is strictly more permissive. Any solution of the original model is also a solution of this one, but, because of the freedom of movement, it may be the case that "useless" move actions are inserted in the plan. It should

nevertheless be noted that any optimal (*i.e.*, shortest) plan of the original model is also an optimal plan of this model.

Experiments

To compare the computational performances of the three models, we conducted experiments on our *simple-goto* domain and some domains of the IPC's HTN track. Each domain is duplicated and modified to have four different versions:

- *original*: the original domain.
- *common*: the domain with the common decomposition model, *i.e.*, the one shown in the Motivating Example section.
- *mutex*: the domain with the mutex model.
- *insert*: the domain with the task insertion model.

Domains

We conducted experiments on the following domains, which are available in the IPC's HTN track (except for our example domain). They have been chosen to show the impact of the models on different types of realistic problems.

Goto Simple The goto simple domain is the one described since the beginning of this paper. The *original* and the *common* versions are the same as the one of the motivating example, with an exponential number of possible decompositions. For an instance i ($1 \leq i \leq 30$), the initial task network is composed of $10 * i$ tasks `goto(t1, p5)` and the truck `t1` is initially at `p1`.

Goto Complex The goto complex problem is a more complex version of the goto simple problem. The domain is the same as the goto simple domain, but the initial task network is composed of $10*i$ tasks `goto(t1, p5)`, $10 * i$ tasks `goto(t1, p3)`, for an instance i. It is used to show the impact of nested high-level tasks on the computational performances of the planners.

Factories The *original* version of the factories-simple domain (Sönnichsen and Schreiber 2021) is the one from the IPC's HTN track. It describes the problem of constructing a factory from different resources, each resource needing to be produced from another less-advanced factory. To bring the resources from one factory to another, a truck is used, and its movement is described by the same `goto` task as in the goto simple domain. Therefore, the *common* version is the same as the *original* one (only the *common* version will be displayed in the future results) and the *mutex* and *insert* versions are easily built as described above.

Rovers The *original* version of the rovers domain (Pellier and Fiorino 2021) is the one from the IPC's HTN track. It describes the problem for a set of rovers to navigate and collect data on another planet, before communicating the collected data to the scientists. In this version, the navigation of the rovers is described by a `navigate-abs` task that can be decomposed by three methods: (i) do nothing, (ii) `navigate` to the given position, (iii) `navigate` to an intermediate position, and then `navigate` to the destination. Interestingly, the task is not recursive and the rover can only navigate to a location separated by at most one intermediate position. This is however not an issue because the instances are built such that if the rover needs to go to a location separated by more than one intermediate position, it will need to do another task, *e.g.*, collect data, before going to the final destination. Thus, because the `navigate-abs` is in the task network of several methods, the rovers can accomplish its mission. The *common* version is built by replacing the methods of the original `navigate-abs` task in order to make it recursive. The *mutex* and *insert* versions are then built as described above based on the *common* version.

Transport The *original* version of the transport domain (Behnke, Höller, and Biundo 2018) is the one from the IPC's HTN track. It describes the problem of transporting packages from one location to another using a truck. The truck is capable of carrying multiple packages at once, under a certain limit, and the movement of the truck is described by the `get-to` task that can be decomposed by three methods: (i) do nothing, (ii) `drive` to the given position, (iii) `get-to` an intermediate position, and then `drive` to the destination. In the *common* version, the second method is removed because it is redundant with the third one, and the subtasks order of the third method is changed to `drive` then `get-to` (we use a right recursion instead of a left one). The *mutex* version is build as described above. The *insert* version is obtained by removing all methods of the original `get-to` task except the one that does nothing and adding a `free-drive` as described above.

Planners

We have selected three planners with different resolution strategies to compare the computational performances of the different models. This way we can show the impact of the models on different strategies.

Aries Aries (Bit-Monnot 2023) is a planner transforming *chronicles* (Ghallab, Nau, and Traverso 2004; Godet and Bit-Monnot 2022) into a Constraint Satisfaction Problem (CSP) which is then solved by a specific solver. Because of the recursive nature of the studied domains, the planner needs to instantiate the initial task network until a maximum depth before the generation of the CSP. This depth is set to an initial value and then increased by a fixed step until the planner finds a solution. For the experiments, we are using the version v0.3.3 of the planner[2].

LinearComplex LinearComplex is the winner of the IPC 2023 HTN Partial Order Satisficing track. The main idea of this planner is to first consider the partially

[2]https://github.com/plaans/aries/tree/v0.3.3

ordered task network as a totally ordered one. For the experiments, we are using the winning version of the planner, named `LinearComplex-config-sat-1`.

PandaPi PandaPi (Holler 2023) is a well known planner in the hierarchical planning community. It performs a progression search on the task network, and uses different heuristics to guide the search in the graph. For the experiments, we are using the version for satisficing partial ordered problems of the IPC 2023, which is using the FF heuristic (Hoffmann and Nebel 2001).

Metrics

We are using the following metrics to compare the computational performances of the different models and planners.

Coverage The *Coverage* evaluates the capability of the planner to solve different problem instances in a given domain. It is defined by

$$Cov = \frac{\text{Number of solved instances}}{\text{Total number of instances}} * 100 \qquad (5)$$

Time Score The *Time Score* evaluates the capability of the planner to quickly find a first solution, and matches the one of the IPC agile tracks. The score is computed based on the time t_i in seconds needed to return the first solution of the instance i and the timeout T (120 seconds in our experiments).

$$TS_i = \begin{cases} 1 & \text{if } t_i < 1 \\ 1 - \frac{\log(t_i)}{\log(T)} & \text{otherwise} \end{cases} \qquad (6)$$

Finally, TS is the mean of all TS_i on every instance, multiplied by 100.

Results

The results of the experiments are shown in Table 1.

As expected, we can notice that the *insert* model allows Aries and PandaPi to solve many more instances than the *common* model.

Surprisingly, the *mutex* model is not performing well for the Aries planner, as shown in the Figure 8 for the Rovers domain. This is due to the fact that the planner instantiate the initial task network until a maximum depth before the resolution. In the *common* model, the planner can take one `move` action in each `goto` task, while in the *mutex* model, the planner must take all `move` actions in a single `goto` task. Therefore, Aries needs to go to a deeper depth to find a solution in the *mutex* model than in the *common* one.

For PandaPi, the *mutex* and the *insert* models are performing better than the *common* model. While the improvement of the *mutex* over the *insert* is relatively modest, when the *insert* model exhibits superior performance, the difference is pronounced. This is clearly visible in the Figure 10 for the Transport domain.

LinearComplex The LinearComplex planner behaves differently from the other planners because it first interprets a PO HTN problem as a Total Order (TO) one, therefore prohibiting any interference among `goto` tasks in the *common* model. As a direct consequence, it does not suffer from the redundancies in the search space and the *common* and *mutex* models have similar performance. Moreover, the *insert* model does not perform well for this planner because it may *require* interleaving of tasks to produce a solution, which is not permitted by TO HTN models.

While this choice of interpreting problems as TO HTN ones appears beneficial for coverage, it should be noted that potential high-quality solution are excluded from the resulting search space. For instance in the Transport domain, a truck can carry multiple packages at once. However, the TO HTN projection initially considered by LinearComplex would treat deliveries one after the other, never transporting more than one package at a time.

Left vs Right Recursion A realistic domain which is representative of the pattern we identified is the Transport domain. The results for this domain are shown in the Figure 10 for PandaPi. We can see that the left recursion of the *original* model has a big negative impact on the computational performances of the planner since all instances timed out. Note that this impact is also visible for LinearComplex as shown in Figure 9. Interestingly, it is not the case for Aries which prefers the left recursion of the *original* model. This may be explained by the fact that Aries relies on a plan-space encoding that is naturally backward chaining from the goals towards the initial state. For such planners, left recursion is better suited as it leaves the end of the plan untouched. On the other hand, forward chaining solvers such as PandaPi and LinearComplex perform better with the right recursion of the *common* model.

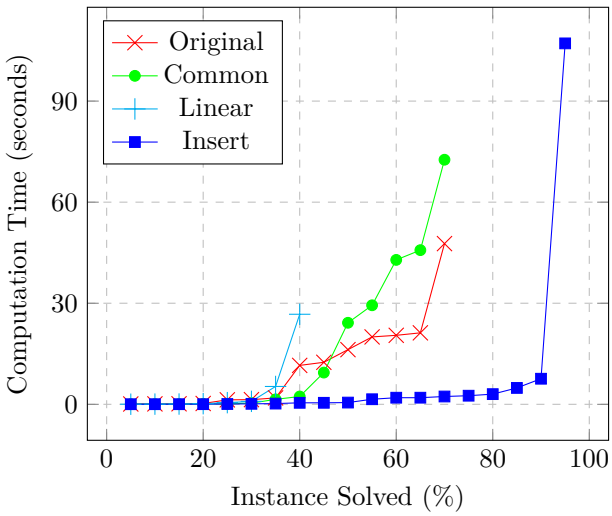


Figure 8: Rovers domain with Aries.

		Aries		PandaPi		LinearComplex	
		Cov	TS	Cov	TS	Cov	TS
Goto Simple	Common	40.00	26.13	0.00	0.00	**100**	**100**
	Mutex	0.00	0.00	**100**	**72.95**	**100**	**100**
	Insert	**100**	**100**	83.33	54.53	76.67	46.55
Goto Complex	Common	26.67	14.07	0.00	0.00	**100**	**98.84**
	Mutex	16.67	8.65	13.33	7.95	**100**	**99.50**
	Insert	**100**	**98.53**	**40.00**	**26.51**	36.67	23.13
Factories	Common	5.00	5.00	**25.00**	**22.68**	**30.00**	**25.09**
	Mutex	5.00	4.22	**25.00**	**22.66**	**30.00**	**24.46**
	Insert	5.00	5.00	25.00	18.98	25.00	19.11
Rovers	Original	70.00	47.09	20.00	10.43	**90.00**	**83.44**
	Common	70.00	47.13	20.00	12.65	80.00	67.52
	Mutex	40.00	34.83	**20.00**	**19.00**	55.00	54.96
	Insert	**95.00**	**81.51**	20.00	17.87	50.00	50.00
Transport	Original	32.50	22.41	0.00	0.00	30.00	24.56
	Common	20.00	14.39	10.00	8.36	**62.50**	**60.25**
	Mutex	17.50	7.60	17.50	11.98	**62.50**	**59.37**
	Insert	**57.50**	**39.24**	**20.00**	**16.28**	55.00	45.52

Table 1: Coverage (Cov) and Time Score (TS) metrics for different planners across different domains and model versions.

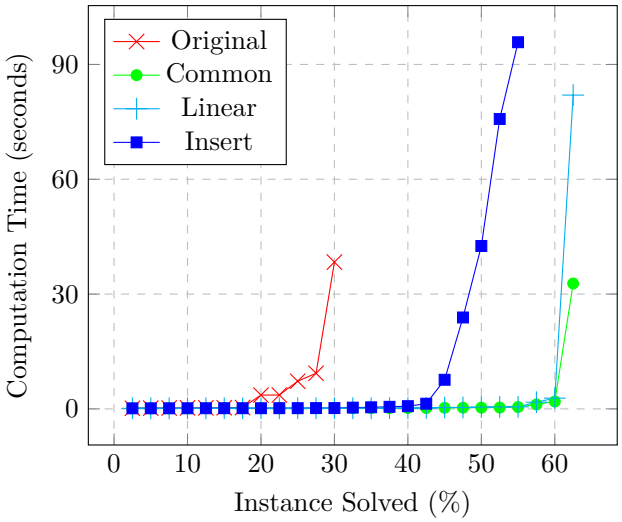


Figure 9: Transport domain with LinearComplex.

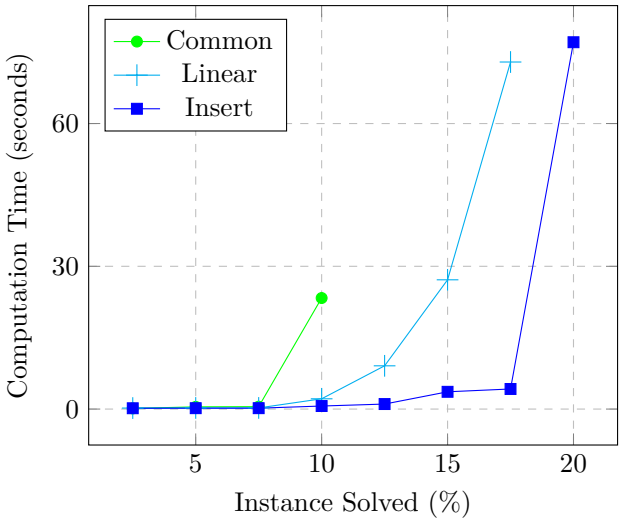
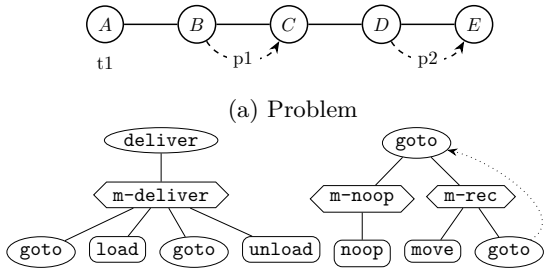


Figure 10: Transport domain with PandaPi. No instance has been solved for the Origin version.

Discussion

The identified pattern is not limited to recursive decomposition of actions as the `move` one, but can also arise from the sharing of common steps of the plan among potentially concurrent top level tasks. This is for instance the case in the Satellite domain of the IPC 2020 HTN track, where the `switch-on` and `calibrate` actions are shared among different tasks. For such cases where a single action may be shared, the impact can be expected to be less dramatic as the number of redundant decomposition would be equal to the number of tasks requiring it.

Finally, let us brought the attention to a false-sharing mechanism that may occur with this pattern. Listing 1 shows a plan with no shared step to sequentially



(a) Problem

(b) Domain Hierarchy

Figure 11: Transport problem with no shared step.

achieve two tasks `deliver(p1, C)` and `deliver(p2, E)`, where `deliver(p, l)` designates the task of delivering the package `p` to the location `l`, as shown in the Figure 11. Even if the plan of the Listing 1 has no shared step, the first and second `move` actions could be attributed to a decomposition of the first `goto` of the second `deliver` task.

Listing 1: Plan with no shared step

```
# Steps 1-4: deliver(p1, C)
move(t1, A, B)
load(p1)
move(t1, B, C)
unload(p1)
# Steps 5-8: deliver(p2, E)
move(t1, C, D)
load(p2)
move(t1, D, E)
unload(p2)
```

Conclusion

In this paper, we have identified a very common pattern in HTN models that can lead to an exponential number of possible decompositions and negatively impact the computational performance of PO HTN planners.

We have proposed two alternative models to the one that leads to this pattern. For native PO HTN planners such as Aries and PandaPi, the model allowing partial task insertion clearly dominates the others and vastly increase the performance on the impacted domains. This suggests that a relaxed HTN models that allow for partial task insertion may be fruitful area for future work.

A notable result is also the drastic performance difference of state-of-the-art planners for different encoding schemes. For instance, unlike PandaPi, Aries favors left-recursive decomposition methods and does not scale up on the *mutex* model. The performance of LinearComplex is highly dependent on the absence of required interleaving between the top-level tasks and thus does not scale on *insert* model. Such issues highlight the fact, at least in PO HTN planning, planner-independent modeling remains a far-fetched goal.

References

Alford, R.; Bercher, P.; and Aha, D. W. 2015. Tight Bounds for HTN Planning with Task Insertion. In *International Joint Conference on Artificial Intelligence*.

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT - Totally-Ordered Hierarchical Planning Through SAT. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32. ISSN: 2374-3468, 2159-5399 Issue: 1 Journal Abbreviation: AAAI.

Bit-Monnot, A. 2023. Experimenting with Lifted Plan-Space Planning as Scheduling: Aries in the 2023 IPC. In *2023 International Planning Competition at the 33rd International Conference on Automated Planning and Scheduling*. Prague, Czech Republic.

Bit-Monnot, A.; Ghallab, M.; Ingrand, F.; and Smith, D. E. 2020. FAPE: a Constraint-based Planner for Generative and Hierarchical Temporal Planning. Technical Report arXiv:2010.13121, arXiv. ArXiv:2010.13121 [cs] type: article.

Erol, K.; Hendler, J.; and Nau, D. 1994. HTN Planning: Complexity and Expressivity. *Proceedings of the National Conference on Artificial Intelligence*, 2: 7.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated Planning: Theroy and Practice*. Morgan Kaufmann Publishers Inc. ISBN 978-1-55860-856-6.

Godet, R.; and Bit-Monnot, A. 2022. Chronicles for Representing Hierarchical Planning Problems with Time. In *ICAPS Hierarchical Planning Workshop (HPlan)*. Singapore, Singapore.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14: 253–302.

Holler, D. 2023. The PANDA Progression System for HTN Planning in the 2023 IPC. In *2023 International Planning Competition at the 33rd International Conference on Automated Planning and Scheduling*. Prague, Czech Republic.

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. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(06): 9883–9891.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C. A.; Ram, A.; Veloso, M.; Weld, D. S.; and Wilkins, D. 1998. PDDL-the planning domain definition language.

Pellier, D.; and Fiorino, H. 2021. From Classical to Hierarchical: benchmarks for the HTN Track of the International Planning Competition. *ArXiv*.

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

Sönnichsen, M.; and Schreiber, D. 2021. The HTN domain "Factories". In *Proceedings of the 10th International Planning Competition: Planner and domain abstracts – hierarchical task network (HTN) planning track (IPC 2020)*, 45–46.

Towards Search Node-Specific Special-Case Heuristics for HTN Planning – An Empirical Analysis of Search Space Properties under Progression

Lijia Yuan, Pascal Bercher

School of Computing, The Australian National University, Canberra, Australia
{lijia.yuan, pascal.bercher}@anu.edu.au

Abstract

In hierarchical task network (HTN) planning, heuristic search is highly effective, but currently, there are only a few available heuristics and they are pre-selected for use. However, during progression-based search, many search nodes exhibit specific properties, e.g., they may become totally ordered or acyclic allowing for the application of specialized heuristics. For these search nodes, we conducted an experimental evaluation, employing reachability analysis, to examine the special cases encountered during search. Measuring how often various special cases (like acyclic problems) occur informs us of which heuristics developed for special cases – selected on a per-search node basis – are most promising.

Introduction

Hierarchical Task Network (HTN) planning (Erol, Hendler, and Nau 1996; Geier and Bercher 2011; Bercher, Alford, and Höller 2019) is a framework within *AI planning* where tasks are organized into hierarchies, consisting of primitive tasks that are directly executable and abstract tasks that require further decomposition. Solving HTN planning problems involves a range of different methods. Among the most successful ones is *progression-based search* (Höller et al. 2020), which operates in a forward manner adhering to specific orderings from left to right. By integrating heuristics, it refines the search trajectory, minimizing exhaustive exploration and effectively guiding the path to the goal.

Heuristic search has consistently demonstrated its effectiveness in HTN planning (Höller, Bercher, and Behnke 2020; Olz and Bercher 2023; Olz, Höller, and Bercher 2023). However, there are only a few heuristics available: There's the *TDG Heuristic* (Bercher et al. 2017), *Relax Composition Heuristic (RC Model)* (Höller et al. 2018, 2020), *ILP HTN Heuristic* (Höller et al. 2020), and a *Landmarks Heuristic* (Höller and Bercher 2021) – to the best of our knowledge, these are the only available ones up to now. All of them are designed for the general case without further restriction on the partially ordered (PO) tasks or how they interact via the task hierarchy. However, some search nodes show specific properties during the search. For instance, even if the initial problem is partially ordered, certain search nodes could become totally ordered (TO) during search or recursive parts might become non-recursive. Designing a heuristic for the general case is complex, so it might be easier to design a heuristic for one of the various special cases. As evidence for this, a recent pruning technique (discarding dead-ends and reducing the branching factor of search) (Olz and Bercher 2023) was developed for TO HTN problems, and yet has to be transferred to partial order HTN planning. We believe that developing pruning techniques or heuristics for special cases like this one (total order) or others, like acyclic problems / search nodes, thus shows great potential. However, choosing such a heuristic or technique would currently only be possible in advance, i.e., before search starts. We however hypothesized that special cases start arising during search, thus allowing to choose specialized heuristics during search thereby increasing their impact as they can be deployed even in problems that don't adhere to the respective special case in advance.

If more heuristics or techniques are tailored to specific special cases, analyzing each search node would enable the selection of a heuristic dedicated solely to that particular case and search node – and thus all search nodes below that one, since once a special case is established, it cannot be violated anymore. This could provide benefits, allowing us to solve planning problems previously impossible or, at the very least, expedite the process compared to before. Special case heuristics could also contribute to that: even if they are not "more informed", they might still be easier to compute.

To assess the potential of choosing specialized heuristics and/or techniques during search, we conducted an experimental evaluation checking how often the various known special cases occur. In this paper we only check TO problems and can hence not report how often TO search nodes occur while solving PO problems. We investigate a range of special cases and document their occurrence in percentage to all search nodes created under the respective search strategy (which was chosen based on the results of the IPC 2024). As a minor side contribution, we also propose how the set of reachable methods – which impact the accuracy of the respective current special case – can be computed in a tighter way, based on a *relaxed reachability analysis*. We report the number of special cases according to both investigations: a naive but quick one, the number and a more informed, but slower one. Finally, we draw a conclusion based on our findings – i.e., whether we believe that specialized heuristics might significantly impact search performance and if so, which special cases are the most promising ones.

HTN Planning Formalism

Our work builds upon the HTN planning formalization initially introduced by Geier and Bercher (2011) and further developed by Bercher, Alford, and Höller (2019), maintaining the core concepts established by Erol, Hendler, and Nau (1996). We would like to note in advance that whereas the formalization provided here is the general one (admitting *any* special case, including "none" by allowing partial order), the empirical study carried out will focus on totally ordered problems only.

A *task network* tn, represented as a tuple (T, $\prec$, α), consists of a finite set of *task id symbols* T, a strict partial order on T denoted by $\prec \subseteq T \times T$ (which is irreflexive, asymmetric, and transitive), and a mapping α that assigns each task id in T to either a *primitive task name* in N_p or *abstract task name* in N_a.

An *HTN domain* D is a tuple (F, N_p, N_a, δ, M), consisting of a finite set of *facts* F, a finite set of primitive task names N_p, a finite set of abstract task names N_a, a mapping $\delta : N_p \to 2^F \times 2^F \times 2^F$ that assigns each primitive task (also called an *action*) to its *preconditions*, *add effects*, and *delete effects*, and a finite set of *decomposition methods* M where each method $m \in M$ is a tuple (c, tn) pairing an abstract task c with a task network tn. An *HTN problem* is a tuple $\mathcal{P} = (D, s_I, \mathrm{tn_I}, g)$, comprising an HTN domain D, an *initial state* $s_I \subseteq 2^F$, an *initial task network* $\mathrm{tn_I}$, and a goal description $g \subseteq 2^F$.

A task network $\mathrm{tn_a}$ = (T_a, $\prec_a$, α_a) will be decomposed by a decomposition method m = (c, $\mathrm{tn_m}$) into a new task network $\mathrm{tn_b}$ = (T_b, $\prec_b$, α_b) if and only if there exists a task identifier $t \in T_a$ such that $\alpha_a(t) = c$ is replaced by subtasks in tn_m, and all ordering constraints from t will be inherited. It is written as $\mathrm{tn_a} \xrightarrow{t,m} \mathrm{tn_b}$. There exists a task network $\mathrm{tn'}$ = (T', $\prec'$, α') equivalent to $\mathrm{tn_m}$ such that $T' \cap T_a = \emptyset$. The only difference between $\mathrm{tn'}$ and $\mathrm{tn_m}$ are task identifiers to avoid repeating task identifiers. The application of m to $\mathrm{tn_a}$ results into the task network $\mathrm{tn_b}$ given as follows.

$$
\begin{aligned}
T_b &:= (T_a \setminus \{t\}) \cup T', \\
\prec_b &:= \prec_a \cup \prec' \cup \prec_x, \\
\alpha_b &:= \alpha_a|_{T_a \setminus \{t\}} \cup \alpha' \\
\prec_x &:= \{(t_a, t_b) \in T_a \times T' \mid (t_a, t) \in \prec_a\} \cup \\
&\qquad \{(t_a, t_b) \in T' \times T_a \mid (t, t_b) \in \prec_a\}
\end{aligned}
$$

The notation $\mathrm{tn} \xrightarrow{*} \mathrm{tn'}$ indicates tn can be decomposed into $\mathrm{tn'}$ by using a sequence of methods.

A task network is *executable* if it has an *executable linearization* of its tasks, where a primitive task $p \in N_p$ linked to action a with $\delta(p) = (pre(a), add(a), del(a))$ is executable in the state s if and only if $pre(a) \subseteq s$, and its execution modifies s to the resulting state $(s \setminus del(a)) \cup add(a)$. An executable linearization for task network tn = (T, $\prec$, α) is a sequence $(t_1, t_2, ..., t_n)$ where each $t_i \in T$ and $\alpha(t_i) \in N_p$ can be executed sequentially. A task network $\mathrm{tn_s}$ = (T_s, $\prec_s$, α_s) is called a solution of an HTN problem $\mathcal{P} = (D, s_I, \mathrm{tn_I})$ if and only if $\mathrm{tn_I}$ is decomposed into $\mathrm{tn_s}$ through a series of decompositions, $\mathrm{tn_s}$ solely comprises primitive tasks ($\forall t \in T_s : \alpha(t) \in N_p$), and $\mathrm{tn_s}$ has an executable linearization. Solution task networks can only be obtained from the initial task network via decomposition without inserting any other tasks.

Known Special Cases

In this section, we provide the definitions for known problem classes, which are *primitive HTN problems*, *totally ordered problems*, *regular problems*, *acyclic problems* (Erol, Hendler, and Nau 1996) and *tail-recursive problems* (Alford et al. 2012; Alford, Bercher, and Aha 2015). We refine the *stratification* by Alford et al. (2012) and propose a more tight *HTN stratification* to assist in defining acyclic and tail-recursive problems. More specifically, our slightly changed formalization of stratifications can be regarded as another minor contribution of the paper, as it has some advantages over the existing one. While the existing one was not wrong, our "tighter version" allows us to differentiate more classes based on the stratification alone, without having to consult the underlying HTN problem.

HTN planning is undecidable (Erol, Hendler, and Nau 1996; Geier and Bercher 2011) but various special cases can make the plan existence problem easier. Erol, Hendler, and Nau (1996) provided tight complexity results (i.e., with matching upper and lower bounds) for primitive HTN problems and for regular problems, and provided upper bounds for totally ordered problems. Alford, Bercher, and Aha (2015) provided matching lower bounds for total-order problems, and tight bounds for tail-recursive problems.

Primitive HTN problems (when all tasks in the initial task network are primitive) are the base case, appearing at the end of the search if a solution exists. Deciding whether a primitive task network has an executable linearization is NP-complete (shown independently by Erol, Hendler, and Nau (1996) and Nebel and Bäckström (1994), later refined by Tan and Gruninger (2014)).

Definition 1 (Totally Ordered Problem)**.** *An HTN problem $\mathcal{P}$ is called* totally ordered *if the ordering of its initial task network $\mathrm{tn_I}$ is totally ordered and all decomposition methods are totally ordered, i.e., for each $m \in M$ with $m = (c, \mathrm{tn_m})$, $\mathrm{tn_m}$ is a totally ordered task network.*

Totally ordered HTN planning is in EXPTIME (Erol, Hendler, and Nau 1996) and EXPTIME-hard (Alford, Bercher, and Aha 2015) and hence EXPTIME-complete.

Definition 2 (Regular Problem)**.** *An HTN problem $\mathcal{P}$ is* regular *if its initial task network $\mathrm{tn_I}$ and $\mathrm{tn_m}$ in all its methods $(c, \mathrm{tn_m}) \in M$ are regular. A task network tn = $(T, \prec, \alpha)$ is regular if*

- *there is at most one task in T that is abstract and*
- *if $t \in T$ and $\alpha(t) \in N_a$, it is the last task in tn, i.e., for all $t' \in T$ with $t' \neq t$, we have $(t', t) \in \prec$.*

In a regular problem $\mathcal{P}$, given that the initial task network $\mathrm{tn_I}$ and task networks $\mathrm{tn_m}$ with all methods $(c, \mathrm{tn_m}) \in M$ are regular, every primitive task in a task network needs to be progressed first, then the abstract task will be decomposed. The abstract task will be the last task in each search node until the primitive task network is found. The largest size

of the search node during the progression search will hence be bounded by the method $(c, \mathrm{tn_m})$ with the largest task network $\mathrm{tn_m}$. These problems were shown to be PSPACE-complete (Erol, Hendler, and Nau 1996).

For the more complex problem restrictions like tail-recursive ones, we provide stratifications, as introduced by Alford et al. (2012) and also used for defining tail-recursiveness (Alford, Bercher, and Aha 2015).

Definition 3 (Stratification)**.** *A set $R \subseteq N_a \times N_a$ is called a* stratification *if it is a total preorder (i.e., reflexive, transitive, and total). A* stratum *is an inclusion-maximal subset $S \subseteq N_a$ such that for all $x, y \in S$ both $(x, y) \in R$ and $(y, x) \in R$ hold.*

According to this definition, stratifications are a concept independent of the underlying HTN problem. In the original definition of tail-recursive problems (Alford, Bercher, and Aha 2015), a specific stratification has to exist for the respective problem to be called tail-recursive. However, although those definitions were sufficient to define tail-recursiveness adequately, there could still be *several* stratifications adhering to the required restrictions. Thus, even for tail-recursive problems, the set of possible stratifications defined over the respective compound tasks wasn't unique. For example, consider a problem an initial abstract task c_I decomposes into the total-order task network $c_1 \rightarrow c_2$ with compound tasks c_1 and c_2. As we will see later, tail-recursiveness will require that $(c_I, c_2) \in R$ and $(c_2, c_I) \notin R$, i.e., it requires c_2 to be on a strictly lower stratum than c_I, but it will *not* impose a restriction on where exactly c_1 sits with regard to c_I. That is, whereas at least $(c_1, c_I) \in R$ is demanded, the original definition of the interplay of stratification and tail-recursiveness would also allow $(c_1, c_I) \in R$ to hold, thus making stratifications not unique. In fact, totality requires that any two compound tasks are "artificially" put into some kind of relationship, even if none of the tasks can be decomposed into another. We propose a stricter definition that removes the requirement of totality and imposes an *exact* relationship between the decomposition hierarchy and the underlying stratification – thus making the stratification a formal one-to-one mapping of the underlying task hierarchy.

Another advantage (on top of having a *unique* stratification per problem, having a *clear, intuitive semantics* for stratifications, and simplified problem definitions for tail-recursive problems) is that in our proposed definition we can differentiate whether singleton stratums represent recursive tasks or not. The original definition required reflexivity, meaning that we have $(c, c) \in R$ for every compound task. This however means that it is impossible to identify, based on the stratification alone, whether the abstract task c can actually reach itself (and hence is recursive) or not, because reflexivity is demanded by definition rather than being a consequence of reachability. In our definition, reflexivity follows only if a task can reach itself.

Definition 4 (HTN Stratification)**.** *Given an HTN domain $D = (F, N_p, N_a, \delta, M)$, a set $R_{HTN} \subseteq N_a \times N_a$ is called an* HTN stratification *of D if and only if it is transitive and it holds $(c', c) \in R_{HTN}$ if and only if c' is reachable from c via decomposition.*

For simplicity, we will use the terms "stratification" and "stratum" and hence skip the "HTN".

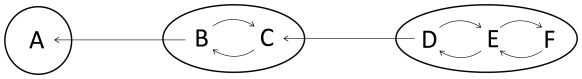


Figure 1: An example of stratification R_{HTN} that has a height of 3, featuring abstract tasks A, B, C, D, E, and F. Circles denote strata, $S_1 = \{A\}$, $S_2 = \{B, C\}$ and $S_3 = \{D, E, F\}$. Directed arrows between circles show the decomposition hierarchy, with arrows pointing from higher to lower levels (e.g., $(A, B) \in R_{HTN}$). Example taken from an ICAPS tutorial on HTN planning (Bercher and Höller 2018).

A directed graph can represent stratification diagrammatically (Figure 1). Task A cannot be decomposed into any other abstract task, and tasks B, C and tasks D, E, F can be decomposed into each other. Due to the requirement of strata being inclusion-maximal subsets of N_a, there are no other strata. As the number of strata in the stratification is 3, the height of the stratification is 3. We also can say that S_2 is a stratum lower than S_3 and S_1 is lower than both S_2 and S_3. Tasks are in different strata if they are in different decomposition hierarchy levels. For instance, task D has a stratum height of 3 since it is at the highest hierarchy level. Primitive tasks, unrelated to the hierarchy, are assigned a stratum height of 0.

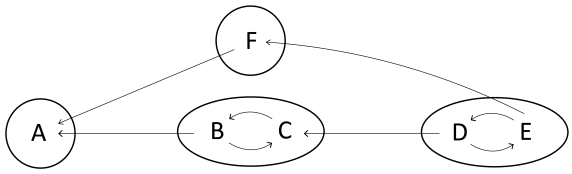


Figure 2: An example of a partially ordered stratification R_{HTN} with a height of 3, featuring abstract tasks A, B, C, D, E, and F. Directed arrows between tasks indicate decomposition methods. Circles denote strata: $S_1 = \{A\}$, $S_2 = \{B, C\}$, $S_3 = \{F\}$, and $S_4 = \{D, E\}$.

Tasks B, C in Figure 2 have stratum height 2 as they are on a strictly higher stratum as A, which is on the lowest level. B and C share a stratum since they can be turned into each other, but they don't share a stratum with A since they can't be turned into each other. Since tasks D, E can be turned into B, C but not vice versa, they are on a stratum with height 3. Now, F is on a higher stratum than A and thus has height 2 or 3. They are not in the same stratum as any of the other tasks, because they cant be turned into each other.

A stratum with a single abstract task c implies that no other task c' exists for which c can reach c' while c' can also reach c via decomposition. However, c may be decomposed into itself, indicating a self-loop if $(c, c) \in R_{HTN}$. Therefore, R_{HTN} effectively differentiates whether c is in a self-loop.

Definition 5 (Acyclic Problem)**.** *An HTN problem $\mathcal{P}$ is acyclic if for its HTN stratification $R_{HTN} \in N_a \times N_a$ holds: if $(c', c) \in R_{HTN}$, then $(c, c') \notin R_{HTN}$.*

The stratification R_{HTN} for the acyclic problem is irreflexive – so demanding that R_{HTN} is irreflexive is an alternative definition. As the stratification is transitive, it becomes asymmetric when it is irreflexive. The search space will be finite during the progression search because the algorithm does not need to deal with recursion. The acyclicity will bring the computational complexity of an HTN problem down to NEXPTIME-complete (Alford, Bercher, and Aha 2015). If an HTN problem is acyclic and totally ordered, it will be PSPACE-complete (Alford, Bercher, and Aha 2015).

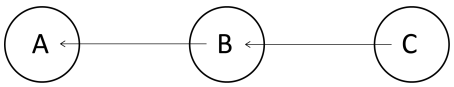
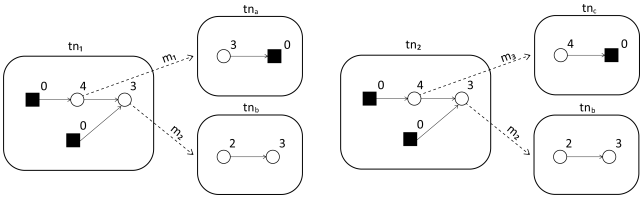


Figure 3: An example of stratification R_{HTN} of the acyclic HTN problem $\mathcal{P}$.

For example, there is an acyclic HTN problem $\mathcal{P}$ where all tasks in N_a are A, B and C, as shown in Figure 3. The stratification of $\mathcal{P}$ is $R_{HTN} = \{(A, B), (B, C), (A, C)\}$ and the strata are $\{A\}, \{B\}$ and $\{C\}$ as per Definition 4. All strata for the acyclic problem contain exactly one abstract task since all abstract tasks are in the different decomposition hierarchy levels.

For the definition of tail-recursiveness, we require the concept of last tasks or non-last tasks, respectively. A task is called *last task* in a task network if and only if all other tasks in that task network are ordered to occur before it. A task is called *non-last task* if and only if it is not a last task. Note that, with the exception of task networks of size 0 or 1, all task networks have non-last tasks (potentially all of them), but not every task network has a last task.

Definition 6 (Tail-Recursive Problem)**.** *An HTN problem $\mathcal{P}$ is* tail-recursive *if for its HTN stratification R_{HTN} and for all methods $(c, \mathrm{tn_m}) \in M$ it holds that for any non-last abstract task c_n in $\mathrm{tn_m}$ with $c_n \in N_a$, $(c, c_n) \notin R_{HTN}$.*



(a) tn_1 with m_1 and m_2 (b) tn_2 with m_3 and m_2

Figure 4: Comparison of tail-recursiveness (left) and non-tail-recursiveness (right). Filled squares denote primitive tasks; circles represent abstract tasks; numbers beside indicate the height of the stratum. Example borrowed from an ICAPS tutorial on HTN planning (Bercher and Höller 2018).

In Figure 4a, the abstract task c in tn_1, which is the one with stratum height 4, can be decomposed into a task network $\mathrm{tn_a}$, which contains tasks that are on a (strictly) lower stratum than c. Therefore, task network $\mathrm{tn_a}$ cannot possibly contradict tail-recursiveness. Also note that the position of c in tn_1 does not play any role, only the position of the tasks within $\mathrm{tn_a}$ are relevant.

The last task c_l in tn_1 can be decomposed into a task network that contains a task that has the same stratum height as c_l. However, that task in $\mathrm{tn_b}$ happens to be the last task, hence this is not a problem. All non-last tasks in $\mathrm{tn_b}$ are indeed on a (strictly) lower stratum than c_l, so this task network also does not contradict tail-recursiveness. Again, the position of c_l within tn_1 was irrelevant, only the position of tasks within $\mathrm{tn_b}$ matters.

In Figure 4b we see an example of a problem that is *not* tail-recursive. Whereas method m_2 is as before and hence doesn't cause problems, method $m_3 = (c, \mathrm{tn_c})$ does. Here, we can see that $\mathrm{tn_c}$ contains a task with stratification height 4, which is the same as the task c from which it got decomposed. Hence that task (in $\mathrm{tn_c}$) would be required to be its last task – but it is not. Again, the position of c within tn_2 was not relevant (it never is); only the positions of tasks within the decomposition method count – interestingly this implies that the form and structure of the initial task network are completely irrelevant. This is interesting because the proof by Erol, Hendler, and Nau (1996) for the undecidability of HTN planning required only two partially ordered compound tasks in the initial task network, whereas all methods were totally ordered. However, tail-recursive methods are enough to achieve decidability, whereas totally ordered methods are not. (This is not a contribution of this paper, but we find this an interesting observation to point out.)

The computational complexity of (ground) tail-recursive problems is EXPSPACE-complete (Alford, Bercher, and Aha 2015), and of tail-recursive and totally ordered problems is PSPACE-complete (Alford, Bercher, and Aha 2015).

Testing for these special cases can clearly be done in polynomial time (with respect to the size of a ground model) since stratifications are essentially the very same as Task Decomposition Graphs (see next section), and checking for the respective properties equals checking simple graph properties. Hence, we will not provide details on how these checks can be done.

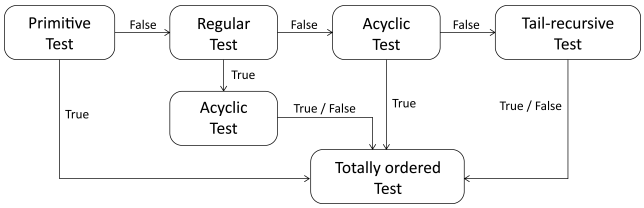


Figure 5: Control flow graph of tests ordering to minimize computational costs and addresses property dependencies efficiently. Arrows with labels to show the path based on test outcomes.

We do, however, mention that the order in which these tests are done can have a large impact on efficiency. Some problem class tests depend on others and are more expensive or time-consuming than others. Therefore, optimizing the sequence of these property tests can lead to greater effi-

ciency (or the other way around: some test orders might be redundant). As shown in Figure 5, if a task network is found to be primitive, it bypasses the need for regular, acyclic, and tail-recursive tests (as they all will be positive), proceeding directly to the totally ordered test. Otherwise, it conducts regular and acyclic tests. Skipping the tail-recursive test is feasible if either the acyclic or regular test is positive, as either acyclicity or regularization is a special case of tail-recursiveness. After these, the task network's total order is assessed, which is independently unaffected by the results of other tests.

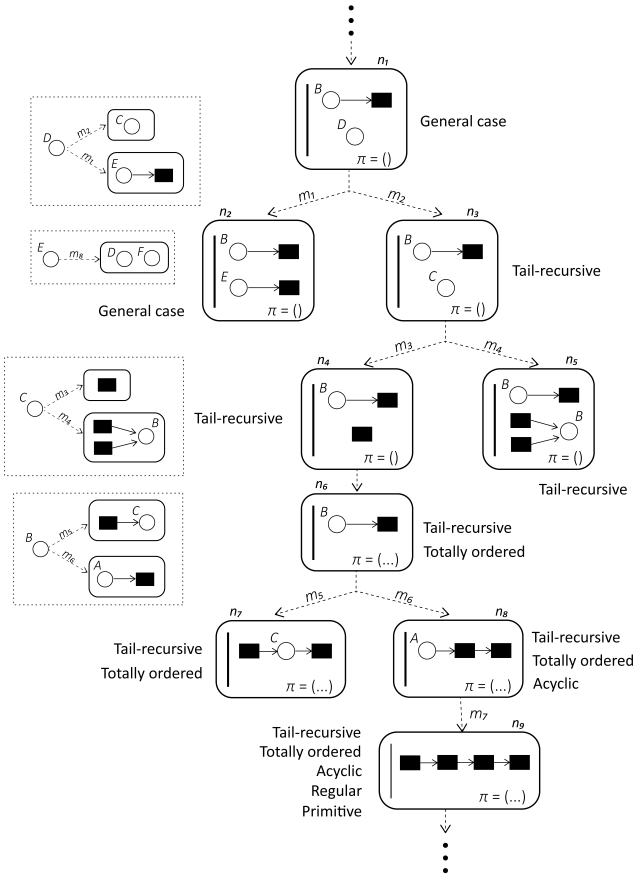


Figure 6: An example of special cases during the search: Filled squares represent primitive tasks, and unfilled circles represent compound tasks. π is the prefix of the generated plan. Vertical lines in search nodes from n_1 to n_9 illustrate the states during the search. We assume no primitive task has preconditions or effects, so the state remains unchanged. The properties of each task network are shown next to each node, with existing methods for tasks B, C, E, D. It has the same stratification as Figure 1.

In progression search, if a task network within a search node possesses a particular property, all task networks within its child nodes will inherit this property. As illustrated in Figure 6, once a property is present in a search node, it propagates to all its descendant nodes. To enhance efficiency, we check the parent node's property before conducting property tests in a search node, potentially reducing overall complexity by avoiding redundant checks.

Reachability Information from Task Decomposition Graphs

In a task network, its properties like total ordering, regularity, acyclicity, or tail-recursiveness depend on both the initial task network and all its reachable methods. The *task decomposition graph (TDG)* is the foundational data structure for hierarchical reachability analysis. It was first introduced by Elkawkagy et al. (2012) and refined by Bercher et al. (2017).

For simplicity, for an HTN problem, we introduce an artificial abstract task c_I that is not originally in the domain. c_I can be decomposed into the initial task $\mathrm{tn_I}$. Subsequently, we incorporate c_I into the domain, establishing $(c_I, \mathrm{tn_I}) \in M$.

Definition 7 (Task Decomposition Graph (TDG))**.** *For a given HTN problem $\mathcal{P}$, the* task decomposition graph (TDG) *is defined as a directed bipartite graph $G = \langle V_T, V_M, E_{T\to M}, E_{M\to T}\rangle$. This graph comprises a set of task vertices V_T, a set of method vertices V_M, edges from tasks to methods $E_{T\to M}$, and from methods to tasks $E_{M\to T}$, such that the following conditions are satisfied:*

1. ***Base Case*** *(task vertex for the given task)*
   $c_I \in V_T$ is the TDG's root.
2. ***Method Vertices*** *(derived from task vertices)*
   Let $v_t \in V_T$ and there is a method $(c, tn) \in M$. Then, for every $v_m \in V_M$, it holds that $(v_t, v_m) \in E_{T\to M}$.
3. ***Task Vertices*** *(derived from method vertices)*
   Let $v_m \in V_M$ with $v_m = (c, (T, \prec, \alpha))$. For each task $t \in T$ where $\alpha(t) = v_t$, it is the case that $v_t \in V_T$ and $(v_m, v_t) \in E_{M\to T}$.
4. ***Tightness***
   G is minimal, such that 1. to 3. hold.

The TDG can represent the hierarchical reachability of an HTN planning problem, i.e. tasks and methods that can be reached via decomposition. According to the simplistic definition provided, it is only based on the reachability from decomposition without considering the states. However, Elkawkagy, Schattenberg, and Biundo (2010) and Bercher et al. (2017) suggested that method nodes containing unreachable primitive tasks based on the state reachability analysis can be removed. (For a more formal definition, consult Def. 4 by Behnke et al. (2020), where this idea is incorporated for grounding lifted HTN planning problems.)

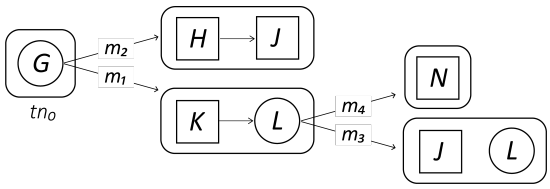


Figure 7: A fragment of a TDG (Bercher et al. 2017): task networks are denoted by surrounding boxes, abstract tasks by circles, and primitive tasks by square boxes.

Bercher et al. (2017) observed that the reachable methods per search node change, as illustrated in Fig. 7. Here, the reachable method set from the initial task network is still m_1 to m_4. However, after m_2 was applied, the reachable methods even became empty, since the new current task network is primitive. If alternatively m_1 would have been chosen, at least m_2 will not be reachable anymore, but potentially even one of m_3 and m_4 as it might be that those actions from m_2's task network were used to (inaccurately) determine the reachability of m_3 or m_4. Hence, recomputing the TDG after some decisions were made can lead to a reduced reachable method set (Bercher et al. 2017).

This is highly relevant for our endeavor of determining search space properties since these properties directly depend on the reachable methods. Going back to our example (Figure 7), if task K cannot be executed, method m_1, and consequently m_3 and m_4 become unreachable. This leaves only m_2 as reachable and thus the sub-TDG makes tn_0 primitive, acyclic, and totally ordered – properties that the initial TDG did not show.

Thus, when computing the properties of a search node – which naturally are based on the set of methods reachable from it – one has two possibilities for what to do:

- We use the initial TDG but restrict it to the methods reachable from the current search node. This process is relatively quick since it does not require TDG recomputation.
- We recompute the TDG from the current search node. This is more expensive but leads to potentially fewer reachable methods (since the initial TDG was computed based on all primitive tasks reachable in it, whereas the made decisions reduced this set and might hence further reduce the size of the new TDG (Bercher et al. 2017)).

In our empirical evaluation we do both. For the recomputation of the TDG, we use the Relaxed Composition Heuristic (Höller et al. 2018, 2020) as a basis. It transforms an HTN search node into a classical problem, which can represent a superset of all reachable decomposition methods (expressed as classical actions). We used this transformation as a basis, estimating all reachable methods by computing a fixed point in a relaxed planning graph (RPG) computed from the classical problem.

Evaluation

We now report on our findings.

Benchmarks.

We analyzed the entire total-order (TO) benchmark set from the IPC 2023[1]. We restrict to TO domains both due to space restrictions and also since the IPC showed that in most instances, the partial order can be compiled away in advance without making the respective problem unsolvable (Wu et al. 2022, 2023).

[1]https://ipc2023-htn.github.io/

Configuration.

Experiments were conducted in a virtualized environment. The underlying hardware was powered by an Intel(R) Core(TM) i9-8950HK CPU, clocking at 2.90GHz, which was allocated a single CPU core and provisioned with 8GB of RAM for the experiments, and 30 minutes limited for each instance (memory and time limit of IPC 2023).

We run the latest progression-based version of the PANDA_π system (Höller et al. 2018, 2020; Höller and Behnke 2021). We opted for the currently best-performing configurations, i.e., Greedy Best First Search (GBFS) combined with the Relax Composition (RC) heuristic (Höller et al. 2018), which utilizes the classical Add heuristic (Bonet and Geffner 2001) that measures the distance to the solution by accounting for both action applications and method decompositions.

Reported Problem Classes.

We analyze the search nodes in terms of the problem classes outlined in Section *Known Special Cases*. However, since several classes overlap (e.g., any regular problem and any acyclic problem are also tail-recursive), we account for those overlappings. We define the following classes:

- *Primitive*: If all tasks in the current task network are primitive. (I.e., defined as usual.)
- *Regular & Acyclic*: If a problem is regular and acyclic, but not primitive.
- *Regular & Cyclic*: If a problem is regular and cyclic.
- *Acyclic*: If a problem is acyclic but not regular.
- *Tail-recursive*: If a problem is tail-recursive but not acyclic.
- *Undecidable*: If a problem is not tail-recursive.

By adhering to these definitions, all classes are exclusive (and so values should add up to 100% per line for the same test in Table 1).

Results.

A summary of our results (per domain) is provided in Table 1. Note that we only include instances that were solved by both runs: those that don't recompute the TDG (referred to as "Simple" in the table) and those that do (referred to as "Reachable"). This is to make the results comparable because only then the explored search spaces are the same.

This is also why we only report on 22 domains although the IPC TO benchmark set encompasses 23. This discrepancy arises because the "Freecell" domain was excluded due to the absence of problem instances that were solved in both runs.

In the table, we report per domain over all solved instances the minimum percentage of the respective problem class ($\downarrow$), the maximum ($\uparrow$), and the average (μ).

Computation Time. Ideally, class identification of problem classes should exert negligible influence on the computational time. In particular, when our ultimate goal is to choose the most informed heuristic based on the respective

Domain		Undecidable						Tail-recursive						Acyclic						Regular & Cyclic						Regular & Acyclic						Primitive					
		Simple			Reachable			Simple			Reachable			Simple			Reachable			Simple			Reachable			Simple			Reachable			Simple			Reachable		
		↓	↑	μ	↓	↑	μ	↓	↑	μ	↓	↑	μ	↓	↑	μ	↓	↑	μ	↓	↑	μ	↓	↑	μ	↓	↑	μ	↓	↑	μ	↓	↑	μ	↓	↑	μ
Assembly	(30)	0	0	0	0	0	0	82	∾A	97	82	∾A	97	0	0	0	0	∾0	∾0	0	7	**2**	0	2	**∾0**	0	9	**∾0**	∾0	9	**2**	∾0	9	1	∾0	9	1
Barman-BDI	(15)	0	0	0	0	0	0	0	0	0	0	0	0	96	∾A	∾A	96	∾A	∾A	0	0	0	0	0	0	∾0	2	∾0	∾0	2	∾0	∾0	2	∾0	∾0	2	∾0
Blocksw.-GTOHP	(29)	0	∾A	**94**	0	∾A	**93**	0	0	0	0	0	0	0	99	**5**	0	99	**6**	0	0	0	0	0	0	0	3	∾0	0	3	∾0	∾0	5	1	∾0	5	1
Blocksw.-HPDDL	(28)	0	0	0	0	0	0	98	∾A	∾A	96	∾A	∾A	0	0	**0**	0	∾0	**∾0**	0	0	0	0	0	0	0	0	0	∾0	2	∾0	∾0	2	∾0	∾0	2	∾0
Depots	(22)	0	∾A	**35**	0	∾A	**32**	0	0	**0**	0	24	**1**	0	∾A	**63**	∾0	∾A	**65**	0	0	0	0	0	0	0	0	0	0	0	0	∾0	8	2	∾0	8	2
Factories	(8)	0	0	0	0	0	0	96	∾A	99	96	∾A	99	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	∾0	4	1	∾0	4	1
Hiking	(24)	87	97	92	87	97	92	3	12	7	3	10	7	0	0	**0**	∾0	1	**1**	0	0	0	0	0	0	0	0	0	0	0	0	∾0	1	1	∾0	1	1
Lamps	(16)	0	0	0	0	0	0	80	∾A	95	80	∾A	95	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	∾0	20	5	∾0	20	5
Logistics-Learned	(44)	0	0	0	0	0	0	98	∾A	∾A	98	∾A	∾A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	∾0	2	∾0	∾0	2	∾0
Minecraft Pl.	(1)	0	0	0	0	0	0	84	84	84	84	84	84	15	15	15	15	15	15	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
Minecraft Reg.	(42)	0	0	0	0	0	0	0	0	0	0	0	0	97	∾A	99	97	∾A	99	0	0	0	0	0	0	∾0	1	∾0	∾0	1	∾0	∾0	1	∾0	∾0	1	∾0
Monroe FO	(17)	74	∾A	95	74	∾A	95	0	0	0	0	0	0	0	17	2	0	17	2	0	0	0	0	0	0	0	4	∾0	0	4	∾0	∾0	14	3	∾0	14	3
Monroe PO	(8)	25	∾A	80	25	∾A	80	0	0	0	0	0	0	0	58	14	0	58	14	0	0	0	0	0	0	0	8	2	0	8	2	∾0	11	5	∾0	11	5
Multiarm-Blocksw.	(74)	0	0	0	0	0	0	98	∾A	∾A	6	96	**50**	0	0	**0**	0	94	**48**	0	0	0	0	0	0	0	0	**0**	∾0	60	**2**	∾0	2	∾0	∾0	2	∾0
Robot	(20)	0	0	0	0	0	0	67	∾A	**97**	4	83	**31**	0	0	**0**	0	∾0	**∾0**	0	0	0	0	0	0	0	0	**0**	0	96	**67**	∾0	33	3	∾0	33	3
Rover	(21)	0	∾A	**89**	0	∾A	**88**	0	0	0	0	0	0	0	97	10	∾0	97	10	0	0	0	0	0	0	0	0	0	0	0	0	∾0	7	2	∾0	7	2
Satellite	(19)	83	99	95	83	99	95	0	0	0	0	0	0	∾0	8	2	∾0	8	2	0	0	0	0	0	0	0	5	1	0	5	1	∾0	8	2	∾0	8	2
SharpSAT	(10)	0	0	0	0	0	0	0	0	0	0	0	0	98	∾A	99	98	∾A	99	0	0	0	0	0	0	0	0	0	0	0	0	∾0	2	1	∾0	2	1
Snake	(2)	0	0	0	0	0	0	∾A	∾A	∾A	99	∾A	∾A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	∾0	∾0	**∾0**	∾0	∾0	∾0	∾0	∾0	∾0
Towers	(13)	0	0	0	0	0	0	75	∾A	97	75	∾A	97	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	∾0	25	3	∾0	25	3
Transport	(25)	88	∾A	97	88	∾A	97	0	0	0	0	0	0	0	2	∾0	0	2	∾0	0	0	0	0	0	0	0	5	1	0	5	1	∾0	12	2	∾0	12	2
Woodworking	(19)	0	0	0	0	0	0	0	0	0	0	0	0	0	∾A	85	0	∾A	85	0	0	0	0	0	0	0	75	8	0	75	8	∾0	25	6	∾0	25	6

Table 1: The minimum (↓), maximum (↑), and average (μ) percentages (%) of undecidability, tail-recursion, acyclicity, regularity, and primitiveness in the search space without and with TDG-recomputation ("Simple" vs. "Reachable"). The computational complexity of properties decreases progressively from left to right. Next to the name of each domain is the number of (solved) problem instances that were used as a basis. "A" (short for "all") represents 100%. Values highlighted in bold show where improvements were achieved due to TDG-recomputation.

search node property, it is essential that the overhead incurred from identifying this special case pays off.

In the "simple" experiment, the total computational time for problem class identification varied across domains, ranging from 0.64% to 42% of total computation time (with an average of 11%), compared to the heuristic's average computation time of 34%. This aligns with expectations, suggesting that class identification times are within acceptable bounds. However, runtimes can still be reduced by optimizing the way classes are identified based on reachable methods (see below).

Conversely, in the "reachable" experiment, the average total computation time for class identification surged to 64%, much overshadowing the heuristic computation time, which averaged 14%. This indicates that class identification significantly impacts the overall computation time in this context, underlining a need for optimization in re-computing the TDG. That said, we did not put any effort into recomputing the TDG effectively, so there is still lots of possible optimizations that could be done. For example, since we use the RC heuristic anyway, we could have used the RPG computed by it when extracting the reachable methods. However, for the sake of simplicity, we re-computed this anyway. Also, we hypothesize that our TDG construction (based on these reachable methods) *and* the way we identify special cases based on the reachable methods could still be optimized (which is the reason why we don't report individual runtimes).

Thus, our reported results can only be used to show which special cases occur and how frequently, but not to draw conclusions about the feasibility of their computation. They do show however that it will be beneficial to optimize TDG recomputation and the identification of special cases.

Difference between Experiments. Before we report on the main findings of our experiments, namely the frequency of certain problem classes, we report on the impact of TDG recomputation. Overall, Table 1 reveals that only a select few domains exhibit notable differences between the "simple" and "reachable" experiments.

The domain exhibiting the most substantial disparity is "Multiarm-Blocksworld" where the average percentage of tail-recursion in the "simple" experiment approaches 100%. However, in the reachable experiment, this average drops to just half. In the remaining 50%, 48% become acyclic (and non-regular), and 2% become regular & acyclic.

"Robot" is another domain showing significant differences. In the "simple" experiment, it only has tail-recursive and primitive search spaces. However, in the "reachable" experiment the average percentage for tail-recursion decreased dramatically from 97 to 31 in favor of 67% regular and acyclic search nodes.

All in all, it seems that TDG recomputation only pays off in a very few domains, so unless recomputation cost can be reduced significantly, our results suggest that on average this might not pay off for the sake of deploying specialized heuristics.

Frequency of Special Cases. We can see that only 5 out of the 22 domains have 90% or more of undecidable search nodes, with 2 further domains having 80% and 89%. Those are the domains where heuristics dealing with the general

case (in TO HTN planning) would be required. So in the huge majority of domains, at least tail-recursiveness could be exploited.

Domains such as "Blocksworld-GTOHP" and "Hiking" exhibit very high ratios of "undecidability", which are over 90%. The number of solved instances of these domains is large, indicating that the existing general heuristic is efficient.

Three domains show nearly 100% or exactly 100% acyclicity: "Barman-BDI", "Minecraft Regular" and "SharpSAT' in Table 1. As the statistics of acyclic search nodes exclude those that are regular (and cyclic) and primitive, these three domains are actually fully acyclic when looking at the data more closely.

In examining the data on acyclicity, it is evident that not all domains exhibit a uniform pattern. For instance, "Blocksworld-GTOHP" ranges wildly in acyclicity, from being virtually cyclic to fully acyclic, as evidenced by its minimum of approximately 0% and a maximum of 99%. Conversely, "Monroe-FO" and "Satellite-GTOHP" show more restrained ranges, peaking at 17% and 8% respectively. It indicates that there are only relatively few nodes in the domain that are acyclic. While the distribution of acyclicity across domains is varied, certain domains inherently exhibit acyclicity, whereas others only manifest it as the problem simplifies during the search. Nonetheless, acyclicity seems to occur often enough within the search space that it seems beneficial to develop specialized heuristics. It is particularly noteworthy that only one single *domain* seems to be acyclic. For all others, acyclicity only occurs within the search space, but not already for the initial search node.

Another *clear* observation is that regularity seems to only occur extremely rarely. In only one domain the ratio of search nodes is about 67%, in one it is 8%, in most others, however, it is 0%, and in just a few 1% or 2%. This however shows a possible optimization as it means that we can stop the TDG recomputation and special case identification once acyclic search nodes are discovered, thus speeding up computation time.

Related Work

There are two lines of related work: one involves choosing a heuristic based on the current search node, which we have not yet explored but forms the motivation for our study; the other is the identification of special cases per search node.

Regarding the first, we are not aware of any such work in HTN planning (which also would not make much sense at the very moment, since no special case heuristics exist as of now). In classical planning, however, Speck et al. (2021) followed a similar idea: selecting the most promising heuristic per search node. Their work does however not do so based on explicit search node properties, but makes dynamic heuristic selection based on Reinforcement Learning. We, however, propose to make this selection dependent based on the specific search node properties and select heuristics that are designed specifically for the respective case.

The other line of related work is investigating search node-specific properties. We are not aware of work along those lines, other than those that investigate properties of entire problem classes. Such a result would be a special case for us. I.e., if exactly 100% of all search nodes show a property P (like being tail-recursive or stronger), then the problem instance has that respective property. Höller (2021) does report on problem class properties (for total-order HTN problems) per problem instance (cf. his Table 1). He reports on a slightly distinct, albeit stronger, criterion concerning tail-recursiveness called non-selfembedding. Any instance that is non-selfembedding, right-recursive (r-rec), and not left-recursive (l-rec) is also tail-recursive. He also reports on acyclic domains ($\neg rec$), but again only per instance.

Höller (2021) does report on problem class properties (for total-order HTN problems) per problem instance (cf. his Table 1). He reports on a slightly distinct, albeit stronger, criterion to decide tail-recursiveness called non-selfembedding. From the instances that are non-self-embedding, those that are only right-recursive (i.e., neither left-recursive nor left-*and* right-recursive), are tail-recursive. He also reports on acyclic domains (¬rec), but again only per instance. A more detailed discussion can be found in Sec. 5 of his most recent work (Höller 2024).

We would also like to note that the PANDA_π planner prints out whether a problem is totally ordered and acyclic when it starts to solve a problem, though it doesn't show the properties of any of the other classes.

Conclusion

Based on the total-order IPC 2023 benchmark set, we analyzed the explored search space of a progression-based plan for the frequency of special cases within the problem classes, such as tail-recursive, acyclic, and regular search nodes. Our motivation for this investigation is the search node-specific deployment of specialized heuristics, tailored towards the respective special case. Our empirical findings indicate a high potential for such heuristics aimed at tail-recursiveness and acyclicity due to the high number of such search nodes. However, computation time for the identification of these classes is still relatively high and thus requires consideration as well.

Acknowledgements

Pascal Bercher is the recipient of an Australian Research Council (ARC) Discovery Early Career Researcher Award (DECRA), project number DE240101245, funded by the Australian Government.

References

Alford, R.; Bercher, P.; and Aha, D. W. 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*, 2–9. AAAI Press.

Behnke, G.; Höller, D.; Schmid, A.; Bercher, P.; and Biundo, S. 2020. On Succinct Groundings of HTN Planning Problems. In *AAAI 2020*, 9775–9784. 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.

Bercher, P.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An Admissible HTN Planning Heuristic. In *IJCAI 2017*, 480–488. IJCAI.

Bercher, P.; and Höller, D. 2018. Tutorial: Introduction to Hierarchical Task Network (HTN) Planning. ICAPS. Available online. Accessed: 25 March 2024.

Bonet, B.; and Geffner, H. 2001. Planning as heuristic search. *AIJ 2001*, 129(1-2): 5–33.

Elkawkagy, M.; Bercher, P.; Schattenberg, B.; and Biundo, S. 2012. Improving Hierarchical Planning Performance by the Use of Landmarks. In *AAAI 2012*, 1763–1769. AAAI Press.

Elkawkagy, M.; Schattenberg, B.; and Biundo, S. 2010. Landmarks in Hierarchical Planning. In *ECAI 2010*, 229–234. IOS Press.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1996. Complexity Results for HTN Planning. *Annals of Mathematics and AI (AMAI) 1996*, 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.

Höller, D. 2021. Translating Totally Ordered HTN Planning Problems to Classical Planning Problems Using Regular Approximation of Context-Free Languages. In *ICAPS 2021*, 159–167. AAAI Press.

Höller, D. 2024. The Toad System for Totally Ordered HTN Planning. *JAIR 2024*, (80): 613–663.

Höller, D.; and Behnke, G. 2021. Loop Detection in the PANDA Planning System. In *ICAPS 2021*, 168–173. AAAI Press.

Höller, D.; and Bercher, P. 2021. Landmark Generation in HTN Planning. In *AAAI 2021*, 11826–11834. AAAI Press.

Höller, D.; Bercher, P.; and Behnke, G. 2020. Delete- and Ordering-Relaxation Heuristics for HTN Planning. In *IJCAI 2020*, 4076–4083. IJCAI.

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 2020*, 67: 835–880.

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

Olz, C.; and Bercher, P. 2023. A Look-Ahead Technique for Search-Based HTN Planning: Reducing the Branching Factor by Identifying Inevitable Task Refinements. In *SoCS 2023*, 65–73. AAAI Press.

Olz, C.; Höller, D.; and Bercher, P. 2023. The PANDADealer System for Totally Ordered HTN Planning in the 2023 IPC. In *IPC: Planner Abstracts – Hierarchical Task Network (HTN) Planning Track (IPC)*.

Speck, D.; Biedenkapp, A.; Hutter, F.; Mattmüller, R.; and Lindauer, M. 2021. Learning Heuristic Selection with Dynamic Algorithm Configuration. In *ICAPS 2021*, 597–605. AAAI Press.

Tan, X.; and Gruninger, M. 2014. The Complexity of Partial-Order Plan Viability Problems. In *ICAPS 2014*, 307–313. AAAI Press.

Wu, Y. X.; Lin, S.; Behnke, G.; and Bercher, P. 2022. Finding Solution Preserving Linearizations For Partially Ordered Hierarchical Planning Problems. In *33rd PuK Workshop "Planen, Scheduling und Konfigurieren, Entwerfen"*.

Wu, Y. X.; Olz, C.; Lin, S.; and Bercher, P. 2023. Grounded (Lifted) Linearizer at the IPC 2023: Solving Partial Order HTN Problems by Linearizing Them. In *IPC 2023*.

Weighted Randomized Anytime Planning in Pyhop

Gabriel J. Ferrer

Hendrix College
1600 Washington Ave.
Conway, Arkansas 72034 USA
ferrer@hendrix.edu

Abstract

Since they produce plans whose quality increases with time, anytime planners are very useful for domains such as robotics and video games. Planners using the SHOP algorithm can operate as anytime planners by retaining the result of the first depth-first search path that reaches the goal, then returning the results of subsequent searches if they improve upon it. However, backtracking to the most recent alternative may not be the best approach for quickly finding a low-cost plan.

In this paper, we replace backtracking depth-first search with a randomized algorithm in the Pyhop implementation of SHOP. Whenever there are multiple options, the planner selects a random operator or method. For every selected option, it records the cost of every plan that includes that option. It uses these cost statistics to make the selection of options that lead to lower-cost plans more probable.

We evaluated the resulting HTN planner on three domains - the Traveling Salesperson Problem, the Pickup and Delivery Problem, and the Satellite Problem. Our experiments show that the weighted-selection approach outperforms both depth-first search and unweighted randomized selection.

Introduction

Anytime planners (Dean and Boddy 1988) produce plans with quality proportionate to available planning time. They are very useful for domains with real-time constraints such as robotics and video games. The Simple Hierarchical Ordered Planner (SHOP) (Nau et al. 1999) was introduced to enable implementing Hierarchical Task Network (HTN) planners for many different domains. The Pyhop implementation of SHOP (Nau 2013) enables an HTN planner to work with arbitrary data structures in the Python language.

We have built an anytime planner atop Pyhop called **pyhop-anytime** (Ferrer 2024d). Within **pyhop-anytime** we have implemented three anytime planners in the `Planner` class in the file `pyhop.py`:

- The time-limit feature described in Section 3.2.4 of the SHOP3 manual (Goldman and Nau 2019), implemented in the `anyhop()` method of the `Planner` class.
- A variant of SHOP in which operators and methods are chosen randomly whenever there is more than one option. There is no backtracking; instead, a new randomized depth-first search is launched whenever the previous search finds a plan or fails. Each plan is generated by the `randhop()` method, called repeatedly by the `anyhop_random()` method until time runs out.
- Another randomized planner, the **action tracker**, in which operators and methods that lead to higher-quality plans have a higher probability of being selected. Each plan is generated by the `make_action_tracked_plan()` method, called repeatedly by the `anyhop_random_tracked()` method until time runs out.

Our anytime planners are designed for domains in which finding an **optimal-cost plan** is **NP-Complete** but finding **non-optimal valid plans** is **feasible in polynomial time**. The planners employ whichever cost function is supplied for a given domain. We experimentally evaluated these planners in the following domains:

- The Traveling Salesperson Problem (TSP).
- The Pickup and Delivery Problem (PDP) is built atop the same graph structure as the TSP, but augmented with packages that must be transported between nodes.
- The STRIPS Satellite domain (sat 2002) from the 2002 International Planning Competition (ipc 2002) provides a number of standardized problems for a traditional STRIPS planning domain.

In nearly all experiments, the purely random planner outperformed the backtracking planner. The action tracking planner sometimes was tied with the backtracking planner but almost always significantly outperformed both of them.

After an overview of the SHOP planning algorithm, we describe our randomized variants, our experiments, and our results, followed by conclusions and future work.

The SHOP Algorithm

In the SHOP planning algorithm (Algorithm 1)[1], **operators** specify state transformations and **methods** decompose tasks into lists of subtasks, in some cases with multiple alternatives (Nau et al. 1999). The subtasks themselves consist of methods or operators. A **plan** is a sequence of operators that completes the given tasks from the given starting state.

Many planning domains are notorious for their intractable computational complexity (Bylander 1994) (Gupta and Nau

[1]Methods `anyhop()` and `pyhop_generator()` of our `Planner` class and `successors()` of our `PlanStep` class.

Algorithm 1: The SHOP algorithm

Input: state, tasks, plan, operators, methods
Output: plan

```
1:  if No tasks remaining then
2:      return plan
3:  else if First task in operators then
4:      Create newstate by applying operator to state
5:      return shop(newstate, remaining tasks, plan with op-
        erator, operators, methods)
6:  else if First task in methods then
7:      for Every method relevant to the first task do
8:          if The method has relevant subtasks then
9:              return shop(state, subtasks + remaining tasks,
                plan, operators, methods)
10:         end if
11:     end for
12: else
13:     return failure
14: end if
```

1992). The SHOP algorithm is well-suited for domains in which finding an optimal-cost plan is NP-Complete but finding non-optimal valid plans is feasible in polynomial time. In these domains, we find a valid plan in polynomial time without backtracking by making arbitrary decisions when nondeterministic choices are encountered[2]. Each backtrack to a nondeterministic choice might produce an improved plan. When a time limit is reached, the best plan found is returned. Increased time limits create more opportunities to find plans that are closer to the optimal cost.

One such domain is the Pickup and Delivery Problem (PDP), which has been heavily studied in the planning literature (Coltin 2014). Each PDP instance consists of an undirected weighted graph, a list of packages to deliver (including origins and destinations), and a robot with a specified carrying capacity. The SHOP methods below[3] find a correct plan in polynomial time without any backtracking. As a preprocessing step, we run the Floyd-Warshall algorithm for finding the shortest paths between all pairs of nodes.

- **deliver-all-packages-from**:
  - If all packages are at their destinations, return success.
  - Otherwise, create a list of pairs of packages and destinations that includes every package aboard the robot along with its destination. If the robot has spare capacity, include every undelivered package with its starting location. Post **possible-destinations** with this list.
- **possible-destinations**: Examine each destination given by the provided destination list.
  - If any destination matches its current location, post **pick-up** or **put-down** depending on the destination's purpose. Then post **deliver-all-packages-from**.
  - If not, nondeterministically choose one neighbor which is part of a shortest path from the current location to a destination in the list. Post **progress-task-list** with that neighbor and a list of only those destinations whose shortest paths include the chosen neighbor.
- **progress-task-list**: Add a **move-one-step** operator to the chosen neighbor. Invoke **possible-destinations** with all of the destinations remaining in its list.

There are three operators in this domain:

- **pick-up**: Pick up an object at the current node.
- **put-down**: Put down an object at the current node.
- **move-one-step**: Move to a neighboring node.

When combined with the SHOP algorithm, the above methods produce a planner that guarantees the delivery of every package. The combination of **possible-destinations** and **progress-task-list** on each invocation will move a package closer to its destination in one of four ways:

1. It picks up a package from its starting location.
2. It puts down a package at its destination.
3. It moves one step closer along the shortest possible path to the destination of a package it has already picked up.
4. It moves one step closer along the shortest possible path to the origin of a package it needs to pick up.

Once a delivery takes place, **deliver-all-packages-from** generates a list of possible pickups and deliveries that includes a strictly smaller number of packages than its previous invocation. Thus, the combination of these three methods is guaranteed to eventually deliver all of the packages, and as every **move-one-step** operator always moves along a shortest path, the overall length of the plan is bounded above by the longest shortest path plus two, multiplied by the number of items to deliver. This is therefore an example of producing a polynomial-time algorithm for finding valid (but not necessarily optimal-cost) plans by combining the SHOP algorithm with suitable domain-specific methods.

Randomizing the SHOP Algorithm

Simple Randomized SHOP

We define an **option** as one possible operator or method that the planner can select. Simple Randomized SHOP[4] is identical to Algorithm 1, except that whenever a method chooses an option nondeterministically (e.g., **possible-destinations**) or multiple methods are options (line 7 of Algorithm 1), it selects one option at random and continues planning without backtracking. After it completes a plan, it keeps generating plans until it reaches its time limit[5].

Weighted Randomized SHOP

Weighted Randomized SHOP[6] is similar to Simple Randomized SHOP, except that instead of selecting options with uniform probability we instead select them according to a dis-

[2]Domains for which the SHOP algorithm might fail to find a plan without backtracking are outside the scope of this paper.

[3]Implemented in `graph_package_world.py` in the `pyhop_anytime_examples` directory

[4]Method `randhop()` of our `Planner` class

[5]Method `anyhop_random()` of our `Planner` class

[6]Method `make_action_tracked_plan()` of `Planner`.

tribution informed by their utility. As with Simple Randomized SHOP, we invoke Weighted Randomized SHOP to repeatedly generate plans until it reaches its time limit[7].

We construct this distribution as follows. For every option we record the following information[8] for every randomly generated plan that employed that option at some point:

- Total cost and number of all successful plans
- Maximum cost of any successful plan
- Number of failed planning attempts

Using the above information, we define a total ordering on options[9] by assigning higher priority to whichever option has lower mean cost[10].

Whenever multiple options have cost information available, each option has a probability assigned as follows[11]:

- The overall set of options is divided into two groups: those that have been selected for a plan and those that have not. Each group is given a probability budget based on the proportion of options in that category.
- Each unselected option has the same probability of selection, receiving an equal share of the unselected budget.
- Those that have been selected are sorted according to the priority scheme outlined above. Each option is assigned a probability double that of the option of next lowest priority, yielding exponentially decreasing probabilities. The sum of these probabilities is the selected-option budget.
- By using a ranking system, we determine selection probabilities independently of the domain's cost function.

We expect that options initially included in plans of low average cost will be included in additional low-cost plans. If this proves to be the case, these options will continue to be selected frequently. If not, these options will diminish in the ranking and be selected less often. Options that initially led to higher cost plans are then more likely to be selected, and if further exploration demonstrates their utility they will be selected more frequently later on.

By default, the tracker does not record outcomes if a given option is the only option. We created a variant[12] in which we record outcomes for every option, even if a given method or operator is the only choice at that point.

Related Work

Another randomized version of **pyhop** is **pyhop-m** (Shao et al. 2021). It expands upon **pyhop-h** (Cheng et al. 2018), an implementation of Pyhop incorporating heuristic search in place of depth-first search. **pyhop-m** replaces **pyhop-h**'s heuristic using the costs of a number of random plans to estimate the quality of each alternative.

[7]Method `anyhop_random_tracked()` of `Planner`.

[8]Recorded in an object of `ActionTracker`.

[9]See our `OutcomeCounter` class.

[10]Although it is beyond the scope of this paper, our implementation also takes plan failures into account by incorporating the number of plan failures into the total ordering.

[11]Method `distribution_for()` of `ActionTracker`.

[12]Set `ignore_single` to `False`.

Our work differs from **pyhop-m** in two critical ways. First, our formulation of anytime planning relies upon a depth-first search implementation of SHOP. As **pyhop-m** employs randomization to serve as an estimator for a heuristic search implementation of SHOP, it is not at all clear to us how it could be converted into an anytime algorithm.

Second, since **pyhop-m** uses random plans to construct a heuristic estimate of distance to a goal, it discards those plans once that estimate has been calculated. In contrast, we employ random plans to both update the probability distribution for operator and method selection that guides our search algorithm as well as to serve as candidate solution plans.

Experimental Analysis

Setup

We evaluated four **pyhop-anytime** variations:

1. **DFS**: Depth-first **pyhop** with a time limit.
2. **Random**: Simple randomized **pyhop**.
3. **Tracker1**: Weighted randomized **pyhop**, tracking all except single-alternative options.
4. **Tracker2**: **Tracker1** modified to track all options.

We tested these four variations in these domains:

1. Traveling Salesperson Problem (TSP)
2. Pickup and Delivery Problem (PDP) with a single robot
3. Satellite coordination problem

TSP shows the performance of our approach in a domain in which every operator selection requires a nondeterministic choice. PDP shows performance in a domain akin to TSP but more similar to a typical HTN planning problem. Satellite shows performance on a well-known benchmark STRIPS planning domain.

Our TSP implementation has one operator (**move** - moves from one city to another) and one method (**complete-tour-from**). The method arbitrarily selects a previously unvisited node to be the next node visited, and then invokes itself recursively. Our PDP implementation is described in the SHOP Algorithm section.

To generate TSP instances, we randomly generate (x, y) coordinates for each city on a 100x100 grid. We then assign edge weights between every pair of cities using the Euclidean distance.

To generate PDP instances, we randomly generate (x, y) coordinates for 36 nodes on a 100x100 grid. For each pair of nodes, there is a 25% chance that we insert an edge. As with TSP, the edge weight is determined by the Euclidean distance. Each of 12 packages are assigned to a distinct random starting node and are assigned a distinct random goal node. The robot can carry up to three packages.

Results and Analysis

We ran four initial experiments (Ferrer 2024b). TSP results are in Figures 1, 2, and 3 and PDP results are in Figure 4. TSP experiments used 15, 30, and 50 cities, with 5, 10, and 30 second limits respectively. PDP used a 30 second limit.

The cost of a TSP plan is the sum of the weights of each edge traversed by the **move** operator. The cost of a PDP

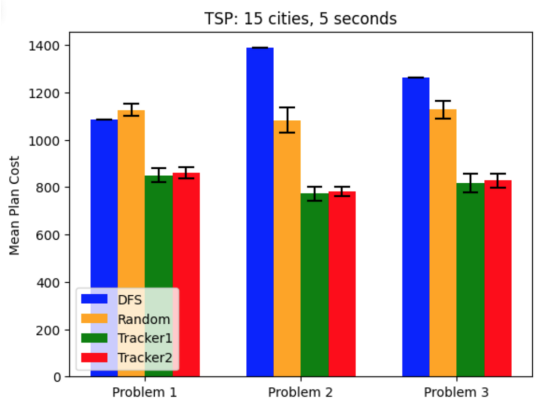


Figure 1: TSP: 15 Cities, 5 seconds

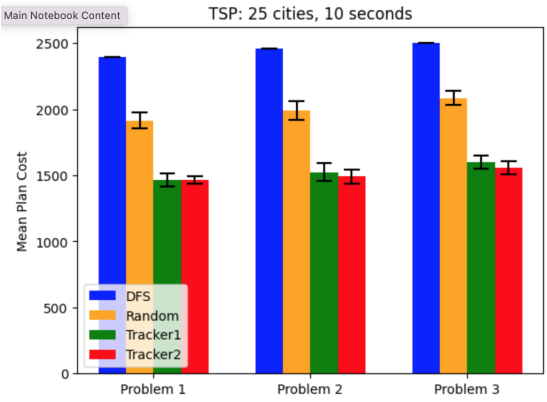


Figure 2: TSP: 25 Cities, 10 seconds

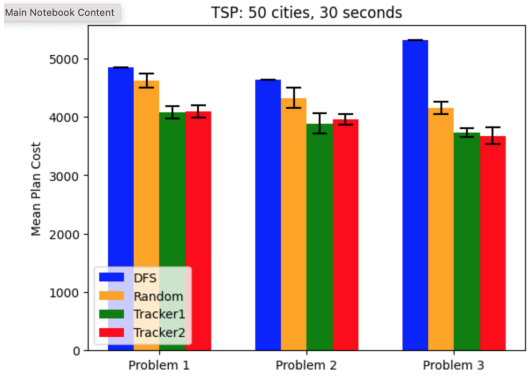


Figure 3: TSP: 50 Cities, 30 seconds

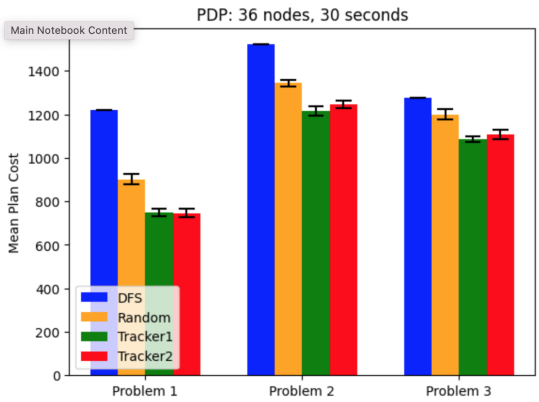


Figure 4: Pickup/Delivery Problem, 36 nodes, 30 seconds

plan is the sum of the weights of each edge traversed by the **move-one-step** operator, with each **pick-up** and **put-down** operator having a cost of 1.

We randomly generated three problem instances for each experiment. For each problem instance, we ran the deterministic DFS planner once and each randomized planner nine times. For the randomized planners, we report the mean plan cost across the nine runs for each problem instance, with error bars indicating a 95% confidence interval.

In the smallest TSP problems (see Figure 1), **Random** typically outperformed **DFS**, but for one problem **DFS** was superior. In all cases, **Tracker1** and **Tracker2** outperformed both **DFS** and **Random** to a degree that their 95% confidence intervals never overlapped. The difference between **DFS**, **Random**, and the **Tracker** implementations diminished on the largest (50 cities) TSP problems as well as the PDP problems but remained statistically significant.

This result seems intuitive. As problem size increases, plan cost as well as the number of distinct operator and method options also increases beyond the degree to which we increased the time budget. Consequently, less information is available for each option, and our probability distribution is less informed. The information is still valuable, but the impact is not as drastic as with smaller problem sizes.

To investigate the impact of increasing the time budget, we ran two additional experiments (Ferrer 2024c) of 200 second duration on the 50 city TSP problem (Figure 5) as well as the PDP problem (Figure 6). On the TSP problem, this replicated the ratio between the costs of the action-tracked plans and the fully randomized plans on the 25 city problem. On the PDP problem, the additional time improved the ratio only slightly. The relative simplicity of the TSP domain seems to make it more amenable to performance improvements from increasing the time budget.

For the Satellite domain, we used the five largest problems in the repository (Ferrer 2024a). In Figure 7, we see that DFS often performed well - in those situations, action tracking matched its performance. In other situations, action tracking greatly outperformed DFS.

Tracker1 was our initial implementation, as we did not find it intuitive to track options that the planner was required to include. **Tracker2** tested that intuition. As they show statistically indistinguishable performance in all experiments, this design decision seems to have no impact at all.

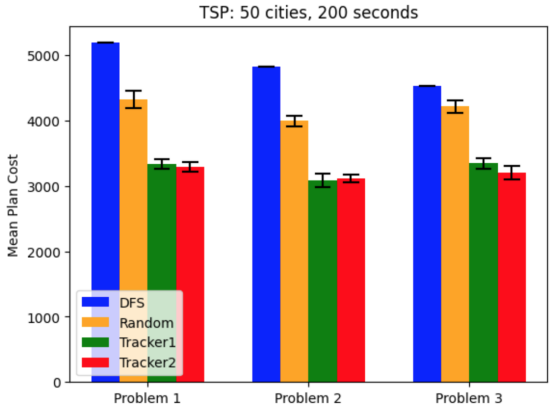


Figure 5: TSP: 50 Cities, 200 seconds

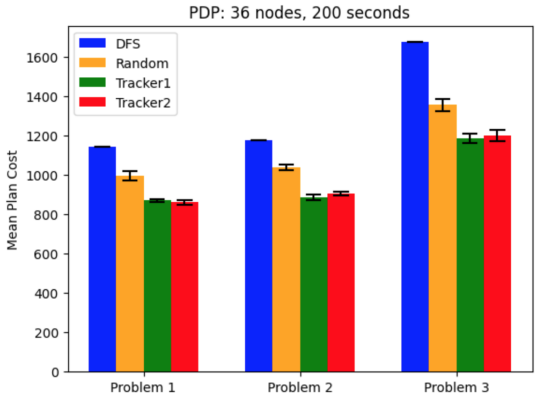


Figure 6: Pickup/Delivery Problem, 36 nodes, 200 seconds

Conclusion

Performance of anytime planning with the SHOP algorithm usually improves by replacing backtracking with randomization. Randomization with operator and method tracking produces significantly larger and more consistent improvements. Future work includes investigating changes to the tracking scheme to enable further improvements with limited time budgets, assessing performance in a wider variety of planning domains, incorporating **pyhop-anytime** into a planning and execution system for the robots in our lab, and potentially incorporating **pyhop-anytime** (Ferrer 2024d) into the **GTPyhop** planner (Nau 2021).

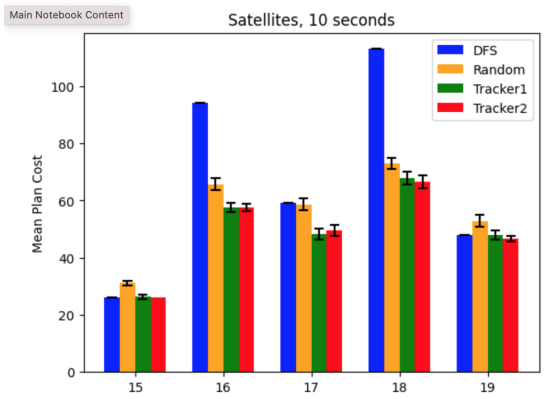


Figure 7: Satellite Domain, 10 seconds

References

2002. International Planning Competition. https://ipc02.icaps-conference.org/.

2002. Satellite STRIPS Domain. https://ipc02.icaps-conference.org/CompoDomains/SatelliteStrips.pddl. Accessed: 2024-05-20.

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

Cheng, K.; Wu, L.; Yi, X. H.; Yin, C. X.; and Kang, R. Z. 2018. Improving Hierarchical Task Network Planning Performance by the Use of Domain-Independent Heuristic Search. *Knowledge-Based Systems*, 142: 117–126.

Coltin, B. 2014. *Multi-Agent Pickup and Delivery Planning with Transfers*. Ph.D. thesis.

Dean, T.; and Boddy, M. 1988. An Analysis of Time-Dependent Planning. In *Proceedings of the 7th National Conference on Artificial Intelligence*.

Ferrer, G. 2024a. Anyhop Satellite Experiments. https://www.kaggle.com/code/gabrielferrer/anyhop-satellite-experiments. Accessed: 2024-05-20.

Ferrer, G. 2024b. Bar Plots for ICAPS-HPlan 2024 paper. https://www.kaggle.com/code/gabrielferrer/bar-plots-for-icaps-hplan-2024-paper. Acccessed: 2024-04-24.

Ferrer, G. 2024c. Extended Experiments for ICAPS-HPlan 2024 paper. https://www.kaggle.com/code/gabrielferrer/extended-experiments-for-icaps-hplan-2024-paper. Accessed: 2024-04-24.

Ferrer, G. 2024d. pyhop_anytime. https://github.com/gjf2a/pyhop_anytime. Accessed: 2024-04-24.

Goldman, R. P.; and Nau, D. 2019. SHOP3 Manual. https://shop-planner.github.io. Accessed: 2024-03-19.

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

Nau, D. 2013. Pyhop. https://bitbucket.org/dananau/pyhop/src/master/. Accessed: 2024-03-19.

Nau, D. 2021. GTPyhop. https://github.com/dananau/GTPyhop. Accessed: 2024-03-19.

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

Shao, T.; Zhang, H.; Cheng, K.; Zhang, K.; and Bie, L. 2021. The Hierarchical Task Planning Method Based on Monte Carlo Tree Search. *Knowledge-Based Systems*, 225.

Toward Planning with Hierarchical Decompositions and Time-frames

Mica Gardone[1], **Rogelio E. Cardona-Rivera**[1 2]

Laboratory for Quantitative Experience Design
[1]Kahlert School of Computing, [2]Division of Games
University of Utah, Salt Lake City, UT, USA
{m.gardone | r.cardona.rivera}@utah.edu

Abstract

The semantics of temporal hierarchical planners are limited. In hierarchical paradigms, temporal reasoning has largely focused on durative constraints of primitive actions, which may be added directly or appear post-expansion. We propose extending temporal reasoning to composite actions, specifically within decompositional partial order causal linked planning. We outline how a general-purpose hierarchical planner can approach temporal reasoning outlined in a STRIPS-like formalism. We build upon existing temporal and hierarchical semantics, and sketch two novel approaches: time-frame planning and decompositional time-frame planning.

1 Introduction

Hierarchical planning has enjoyed uses in robotics, space, and business applications. However, space agencies (European Space Agency 2019; United States National Aeronautic Space Administration 2023) have posted open questions and concerns with time for a variety of reasons. One such open question is dealing with time as both a window and an end time. *Temporal hierarchical* planning has received comparatively less time in research than its non-temporal cousins, and as such there are still many elements under- and undefined.

Temporal planning has focused a variety of topics like planner-schedulers (Parimi, Rubinstein, and Smith 2022), portfolio planning (Furelos-Blanco and Jonsson 2018), and many other forms of planning (Younes and Simmons 2003; Turi and Bit-Monnot 2022; Do and Kambhampati 2014). There still is not an agreed-upon, general purpose formalism, however, there are common attributes among all systems. There has been work to discover better heuristics through non-temporal means (Cavrel, Pellier, and Fiorino 2023). Representing time in planners – e.g., *timelines* (Frank 2013), *temporal constraint networks* (Dechter, Meiri, and Pearl 1991), and *chronicles* (Rahmani, Shell, and O'Kane 2021) – has been a major research area, for the sake of improving both knowledge representation and search. Some planners (Dvorak et al. 2014; Bit-Monnot et al. 2020) approach temporal hierarchical reasoning, yet continue to assume that composite actions are non-temporal.

Within the hierarchical community, there have been open challenges (Kiam, Bercher, and Schulte 2021) and proposing semantics (Smith and Cushing 2008; Pellier et al. 2022). The challenges produce domain-specific solutions that we could draw upon. Temporally-aware hierarchical reasoning proposals state only primitive actions are capable of having duration semantics for simplicity. However, this cannot be the case as an "instantaneous" composite action composed of durative primitive actions is logically inconsistent. In domains or specifications where expansion is particularly expensive, delayed, or non-desirable (Gréa, Matignon, and Aknine 2018), instantaneous composite actions are not representative enough and can lead to undesirable outcomes.

In this paper, we approach a potential solution to allowing composite actions to have temporal information by:

1. Proposing a novel time-frame based paradigm: time-frame planning, and
2. Propose how to combine time-frame planning and decompositional planning.

We believe the extensions provided here will help foster further discussion around a general purpose, temporal hierarchical planning formalism.

2 Related Work

There is an on-going discussion of the semantics of temporal hierarchical planning. We initially draw upon work done recently by (Pellier et al. 2022). We relax the need of duration being only on primitive actions/tasks. As well, we introduce potential search space constraints that can impact what expansions are selected. Action Notation Modeling Language (Smith and Cushing 2008) (ANML) and PDDL 2.1/2.2 (Fox and Long 2003; Edelkamp and Hoffman 2004) also discuss some basic forms of hierarchy, yet leave temporal hierarchical semantics undefined.

There have been many domain-specific planners that have built their own solution. While each solution is unique and ground breaking for its area, none build are a formal, universal framework. Some early temporally-aware software systems utilize a strongly built library of actions based on empirical data, one being the Heuristic Scheduling Testbed System (HSTS) (Muscettola et al. 1992). These actions, and environments, are believed to be common occurrences in the domain they are applied; making them tightly coupled to their originating domain area. Other planners that utilize a planner-scheduler hybrid planner (Cesta et al. 2007)

are also difficult to generalize due to a specialized language and/or formalism specific to the problem. FAPE is one influential planning system that defines its own hierarchical and temporal planning for acting through a combined planner-executor (Bit-Monnot et al. 2020). One key difference between what we propose and what FAPE implements is that the latter is specific to ANML and chronicle planning. FAPE's temporal extents are only known on fully expanded hierarchical tasks. In this paper, we propose planners have the ability to reason over temporal extents on *un-expanded* hierarchical actions.

3 Background

In this section, we describe the elements necessary from simple temporal planning to understand time-frame planning. We will also establish a baseline for hierarchical planning to discuss how to combine the two paradigms.

3.1 Simple Temporal Domains & Problems

To begin our discussion of time-frame planning, we refer to the simple temporal problem. The notation we use is both STRIPs-like and derived from COLIN (Coles et al. 2012). A simple temporal problem with discrete effects from PDDL 2.1 can be represented as $\langle \mathcal{I}, \mathcal{A}, \mathcal{G} \rangle$ where:

1. $\mathcal{I}$ is the initial state which contains a set of propositions and an assignment of values to a set of numeric variables.
2. $\mathcal{A}$ is the set of actions, where each action (a) is defined as $\langle \text{pre}_{\vdash}, \text{pre}_{\leftrightarrow}, \text{eff}_{\vdash}, \text{pre}_{\dashv}, \text{eff}_{\dashv}, \text{dur} \rangle$, such that:
   (a) pre_x denotes the conditions that must be maintained both *at start* ($\text{pre}_{\vdash}$) and *at end* ($\text{pre}_{\dashv}$) of a.
   (b) eff_x denotes the effects that are applied after the conditions of a are met in the start ($\text{eff}_{\vdash}$) and end ($\text{eff}_{\dashv}$). Both effect collections are further defined as:
      i. eff_x^-, propositions to be removed from the world,
      ii. eff_x^+, propositions to be added to the world,
      iii. eff_x^n, modifications on numeric variables.
   (c) $\text{pre}_{\leftrightarrow}$ denotes the invariants (*over all*); these are conditions that must be maintained between the start and end of a.
   (d) dur denotes the duration constraints which defines the duration between a's start and end. These constraints are further refined with respect to ordering constraints. This allows for a special parameter, `?duration`.
3. $\mathcal{G}$ is the goal of the problem: a set of propositions and values that must be achieved.

Further, the definition of a duration in an action can take either one or two constraints. A constraint takes the form: $\langle$`?duration`, op, $c\rangle$ where `?duration` is the special purpose parameter, op $\in \{>, >=, <, <=, =\}$, and $c \in \mathbb{R}$.

Two constraints define two unique bounds on the operator's minimum and maximum. The equality operator cannot be used in the definition of two constraints. The two constraint tuple takes the form: $\langle\langle$`?duration`, $\text{op}_1, c1\rangle$, $\langle$`?duration`, $\text{op}_2, c_2\rangle\rangle$ where $\text{op}_1 \in \{>, >=\}$, $\text{op}_2 \in \{<, <=\}$, $c_1, c_2 \in \mathbb{R}$ and c_1 does not have to equal c_2. The two constraints can approximate the behavior of the equality operator.

Flaws & Refinements. Simple temporal planning in partial-order causally linked (POCL) planning entails two basic types of flaws: open conditions and causal threats. Open conditions are unsatisfied preconditions. In temporal planning, open conditions can be in either the *at start* condition block, *at end* condition block, or *invariant* block. An open condition is resolved one at a time, en-queuing all potential fixes either from instantiating new actions or reusing steps in the plan. Causal threats arise when a causal link would be undone by an inverse effect (e.g., p and $\neg p$). Causal threats are solved be one of three methods: promotion (ordering the threatening step after the causal link's consumer), demotion (ordering the threatening step before the causal link's producer), or non-codesignation (in the event of lifted actions).

Refinements to the plan are made per *refinement strategies*, which are processes of which flaws are selected in some order. All solutions generated by the flaw are then queued back onto the search fringe. Refinement strategies can come in different forms and deal with a variety of flaws (Pollack, Joslin, and Paolucci 1997). Planners can also change strategies at run-time (Younes and Simmons 2003).

Solutions. A solution to the given problem is a sequence of actions from $\mathcal{A}$ that establishes all goal conditions in the problem. A solution must respect the duration constraints of every action in the solution; that is, no action should last longer than its maximum defined duration or be scheduled such that it takes less time than minimally allowed. The solution is the tuple $\langle \mathcal{S}, \mathcal{O}, \mathcal{L} \rangle$ where:

1. $\mathcal{S}$ is the set of actions instantiated into the plan, referred to as *steps*. All $s \in \mathcal{S}$ correspond to an $a \in \mathcal{A}$ from the problem definition.
2. $\mathcal{O}$ is the orderings over the steps in the solution. The ordering system is temporally-aware. Every $o \in \mathcal{O}$ takes the form $p_{s/e} \prec c_{s/i/e}$, where $p, c \in S$. s, i, e correspond to start, invariant, and end blocks.
3. $\mathcal{L}$ are links between an effect and a precondition. A causal link $l \in \mathcal{L}$ is the tuple $\langle p_{s/e} \prec c_{s/i/e}, q \rangle$ where $p, c \in S$ and q is a predicate effect in the producer (p).

3.2 Hierarchical Reasoning

There are several different variations of hierarchical planning (Bercher, Alford, and Höller 2019). We specifically utilize the decompositional POCL (DPOCL) (Young and Moore 1994; Winer and Cardona-Rivera 2018) formalism as our approach to hierarchical reasoning.

A standard decompositional problem takes the same form as in Section 3.1. The key difference lies in the set of actions, $\mathcal{A}$, where each a is defined as $\langle \text{pre}, \text{eff}, \text{composite}, \Lambda \rangle$. Each element is defined as:

1. pre is the action's preconditions, which must be maintained at the start.
2. eff is the action's effects, which affect the world. Effects take the same form as in Section 3.1.
3. composite is a true/false flag to indicate it is a composite header step that needs to be decomposed or expanded. If the flag is true, then the step is a composite step.

4. Λ is the set of schemas that can be used to expand the composite action. A schema $\lambda \in \Lambda$ takes the form $\langle \mathcal{S}, \mathcal{O}, \mathcal{L} \rangle$ where:
   (a) $\mathcal{S}$ is the set of pseudo-actions in the decomposition that must be added to the plan. All $s \in \mathcal{S}$ can either be a composite or a primitive, allowing for the nesting of composite pseudo-actions.
   (b) $\mathcal{O}$ is the set of orderings over the steps in $\mathcal{S}$.
   (c) $\mathcal{L}$ is the set of causal links in the decomposition that links effects to preconditions.

Flaws & Refinements. On top of the open condition and causal threat flaws in POCL, a decomposition flaw is introduced. This flaw signals to the planner that the given step is composite and thus must be expanded. All causal links that link to and from the composite step must be updated to the newly created dummy start and end.

DPOCL, as it was introduced, requires decomposition flaws to be resolved first before any other flaw. We do not make that strong of commitment to decomposition first, as there are situations in planning where this is not desired (Gréa, Matignon, and Aknine 2018).

Expanding Schemas. When expanding schemas, it is important to modify all existing orderings such that everything that comes after, before, and during the composite step is maintained. For DPOCL, a decompositional link is generated on expansion to keep associated actions together.

Solutions. A solution in a standard hierarchical problem is: $\langle \mathcal{S}, \mathcal{O}, \mathcal{L}, \mathcal{D} \rangle$. $\mathcal{S}, \mathcal{O}$, and $\mathcal{L}$ are similar to the simple temporal solution without time. $\mathcal{D}$ is the set of decomposition links.

4 Towards Temporal Decomposition

We extend on the prior notation of a simple temporal solution to a novel planning paradigm: time-frame planning. While we outline a sketch here, space precludes us from diving in to the deeper technical representations. A parameter is added to the problem representation to support new reasoning, creating the tuple of $\langle \mathcal{I}, \mathcal{A}, \mathcal{G}, \mathcal{T} \rangle$ where:

1. $\mathcal{I}, \mathcal{A}$, and $\mathcal{G}$ are the same as before.
2. $\mathcal{T}$ is a constraint on the duration of the solution much in the same way as dur is for actions. That is, $\mathcal{T}$ defines a minimum and/or maximum duration that bounds the solution. $\mathcal{T}$ utilizes the special parameter provided in temporal-metric planning, `total-time`, to define its own constraints. However, this duration constraint is not modified based on what is in the plan: it defines what a solution to the problem must satisfy. The parameter `total-time` also takes the form of a tuple, $\langle min, max \rangle$, which indicates the absolute minimum and maximum time.

A time-frame is composed in the same way the duration of an action is: there exists either a single constraint, or two constraints. The single constraint can define either one bound with the set $\{>, <, >=, <=\}$. The two constraint tuple can only use $\{>, >=\}$, which defines a minimum, and $\{<, <=\}$, which defines a maximum. The prior definitions of duration constraint tuples for both single- and two constraints applies to a time-frame as well.

New Flaws. As we have defined a new constraint to satisfy, there must also be some way for a planner to know when these issues arise. Overtime flaws are generated when at least one chain of actions in the plan could run longer than the maximum duration the problem defines. An overtime flaw can be defined as the tuple: $\langle \texttt{total-time}_{max}, >, c \rangle$ where $\texttt{total-time}_{max}$ is the max duration of the plan and $c \in \mathbb{R}$ is a constant defined as the solution's maximum. In this case `total-time` exceeds c, indicating we could potentially go over time. This type of failure can be found in space: attempting to facilitate a spacewalk longer than the available oxygen in the astronaut's system.

Conversely, undertime flaws are generated when no chain of actions reach the minimum threshold defined by the problem. This flaw can be expressed in the tuple: $\langle \texttt{total-time}_{min}, <, c \rangle$ where `total-time` is the special parameter and $c \in \mathbb{R}$ is a constant defined as the solution's minimum. This indicates that our longest minimum time is still below our threshold, and must be increased. This flaw type can be observed when attempting a slingshot maneuver: burn your engines for too little time, the rocket might end up being pulled in closer to the planet which results in negative consequences.

Solutions. A solution in time-frame planning adheres to the same principles as a solution in simple temporal problems: a series of actions that respect each other's duration constraints. A time-frame solution differs, however, as the duration constraint $\mathcal{T}$ must also be respected. Moreover, the solution has the potential to offer a family of solutions much in the same way a solution that is not total-ordered offers multiple solutions, conditioned on not using the equality ('=') operator in actions. The optimal solution returned from a simple temporal planner might not be complete in a time-frame setting.

4.1 Decompositional Time-frame Planning

Combining time-frame and hierarchical reasoning has a fair number of questions and concerns such as: 1. how should duration constraints be treated, 2. should decompositional actions always be bound, 3. should schemas be able to affect the top-level bounds, and many other considerations, not including to specific planning styles (e.g., HTNs vs DPOCL). Actions are expanded to be the tuple $\langle \text{pre}_{\vdash}, \text{pre}_{\leftrightarrow}, \text{eff}_{\vdash}, \text{pre}_{\dashv}, \text{eff}_{\dashv}, \text{dur}, \text{composite}, \text{bound}, \text{rel}, \Lambda \rangle$ where:

1. The following remain the same as in Section 3.1 and Section 3.2: pre_x, eff_x, dur, composite, and Λ.
2. *bound* states how to treat the action's temporal duration during parsing. There are two types of bounds: *strict* and *flexible*. *strict*-bounds are top-to-bottom: all schemas associated with the action must adhere to the constraints. *flexible*-bounds are a bottom-up approach to temporal bounds: the top-level action's duration is defined by the schemas. *flexible*-bounded actions may find that they are

infinite in maximum duration due to not all open conditions and causal threats being resolved in at least one schema. When planning, all bounds are treated as strict by the planner.

3. rel is how long schemas can take relative to their starting time: given the current minimum duration of the schema, how far is the maximum. If a schema has no open conditions or causal threats, rel is ignored. Otherwise, the minimum time is determined from orderings.

With the expansion of a composite action, we must also discuss changes to schemas. As expanding a composite action leads to the inheritance of all flaws that come with the given schema, we must contend with inherited temporal flaws. A schema's tuple is thus expanded as: $\langle \mathcal{S}, \mathcal{O}, \mathcal{L}, \text{pre}_{\leftrightarrow}, \text{dur}, \text{rel} \rangle$. $\mathcal{S}, \mathcal{O}$, and $\mathcal{L}$ are the representation of sets as in Section 3.2. However, the other components are defined as:

1. $\text{pre}_{\leftrightarrow}$ are *schema-level* invariants. These are added to the plan as open conditions to the dummy start, as schemas might have contradicting invariants.
2. rel is the same as in the composite operator definition. This rel overrides the composite operator's rel, if defined.
3. dur, unlike in the action scheme, cannot be directly defined by the domain engineer. The maximal and minimal extents, the components of dur, are defined by the schema's steps and orderings. Should there exist an open condition or causal threat, then the maximum duration of the schema is defaulted to infinite unless rel is defined either in the schema itself, or the top-level action.

Expanding Temporal Schemas. When expanding a temporal schema, the *at start*, *at end*, and *over all* blocks of the top-level action must be maintained. *at start* and *at end* can be decomposed into two pairs of actions with a duration of 0 with their respective blocks. For example, say a composite step has an at start condition a and at start effect b. The new stand-in for the composite start has a condition of a, and its at end effect has an effect of a and b. *at end* is decomposed in the same way, just using its condition and effect blocks.

over all, when expanded, asks: what does it mean to have an invariant over multiple actions? We represent such cases as causal links between the *at start-at end* effect and *at end-at start* condition. Schema-level invariants are appended to the end of at end effects. Of course, invariants also need to be satisfied as open conditions which we can solve by placing them as *at start* at end conditions.

Decomposition links should be generated the same; however, with the added notation of time-frames we must also record how long this decomposition can be. Modifying the duration of a top-level composite action before expansion should affect the search space. By further constraining the action, schemas are culled from potential expansions. Decomposition links should reflect these constrained bounds.

Bound Interactions. Another question we must consider is, what happens if a composite step is *strict*-bounded and a schema runs over or under time?

For example, a top level action that in some domain has a defined constraint of $\langle 5, 10 \rangle$ with a *strict*-bound. If there is a schema that has a calculated duration constraint of $\langle 6, 8 \rangle$, there are no issues. If a schema has a duration constraint of $\langle 6, 12 \rangle$ or $\langle 2, 8 \rangle$, the parser should not error out as there is a potential solution (reducing or increasing the actions). If there is a schema that is $\langle 2, 5 \rangle$ or $\langle 11, 17 \rangle$, then the parser should produce an error and prevent the planner from running as there are no actual solutions. If a schema has no maximum due to there being a flaw in it, then the maximum is either $\text{schema}_{\text{min}} + \text{rel}$, if rel is defined, or the maximum allowed by the action. Given the same duration constraints, if the action was *flexible*-bound then no error would be produced in any of the cases. The action's duration would not be $\langle 5, 10 \rangle$ but $\langle 2, 17 \rangle$.

Modified Flaws. How does temporal hierarchical planning interact with time-frame planning? The overtime and undertime flaws need to be modified slightly as both flaws exist as plan-wide issues. We fix this by adding a reference to a decomposition link (d) to the definitions. Thus, dur can be either the `total-time` or the duration of a specific decomposition, depending on if $d = \emptyset$ or $d \in \mathcal{D}$, respectively. The latter statement can be derived by examining the distance between the schema's start's end and the schema's end's start. A planner should resolve these flaws much in the same way as a plan-wide overtime/undertime flaw with the key difference being to begin at the expanded schema's end. Actions that are linked only to the schema's start are not considered for the solution, as they only shift the sub-plan and don't contribute to going over the duration.

Fundamentally, the decomposition flaw remains the same as described in Section 3.2.

Solutions. A given solution to a temporally-aware, hierarchical reasoning planner is: $\langle \mathcal{S}, \mathcal{O}, \mathcal{L}, \mathcal{D} \rangle$. $\mathcal{S}, \mathcal{O}$, and $\mathcal{L}$ are the same as in simple temporal and time-frame planning. $\mathcal{D}$ is the set of temporally-aware decomposition links. All solution requirements are the same as in time-frame planning and classical hierarchical planning.

5 Review & Future Work

In this paper, we defined two novel paradigms: time-frame planning and decompositional time-frame planning. The former operates over primitive operators and allows problems to specify further solution constraints in both time windows and time periods. We described two new flaws, overtime and undertime, how they're identified, and how a planner could resolve them. We extended these notions to decompositional planning, giving composite actions duration constraints. We defined new and necessary elements for decompositional time-frame planning at both the action level, and its schemas. We also added more information to the time-frame flaws to constrain decomposed actions' sub-plans to a specified, desired time. By doing so, we stated expansion itself can be impacted by the composite action's duration changing and that some schemas may not be added as potential solutions. Future implementations can utilize the same domains and problems seen in prior work (Yorke-Smith 2005) for testing.

Acknowledgements

This material is based upon work supported by the U.S. National Science Foundation (Grant #2046294). We also thank our anonymous reviewers for their feedback during peer review.

References

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*, 6267–6275.

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

Cavrel, N.; Pellier, D.; and Fiorino, H. 2023. On Guiding Search in HTN Temporal Planning with non Temporal Heuristics. *ArXiv*, abs/2306.07638.

Cesta, A.; Cortellessa, G.; Fratini, S.; Oddi, A.; and Policella, N. 2007. The MEXAR2 Support to Space Mission Planners. In *Proceedings of the 17th European Conference on Artificial Intelligence*.

Coles, A. J.; Coles, A. I.; Fox, M.; and Long, D. 2012. COLIN: Planning with Continuous Linear Numeric Change. *Journal of Artificial Intelligence Research*, 44: 1–96.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence*, 49(1-3): 61–95.

Do, M.; and Kambhampati, S. 2014. Sapa: A domain-independent heuristic metric temporal planner. In *Proceedings of the 6th European Conference on Planning*, 57–68.

Dvorak, F.; Bartak, R.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014. Planning and Acting with Temporal and Hierarchical Decomposition Models. In *Proceedings of the 2014 IEEE 26th International Conference on Tools with Artificial Intelligence*, 115–121.

Edelkamp, S.; and Hoffman, J. 2004. PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition. Technical Report 195, Albert-Ludwigs-Universität Freiburg, Institut für Informatik.

European Space Agency. 2019. APSI - Advanced Planning and Scheduling Initiative. https://essr.esa.int/project/apsi-advanced-planning-and-scheduling-initiative (Last accessed: May 24, 2024).

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20: 61–124.

Frank, J. 2013. What is a Timeline? In *Proceedings of the 4th Workshop on Knowledge Engineering for Planning and Scheduling at the 23rd International Conference on Automated Planning and Scheduling*, 31–38.

Furelos-Blanco, D.; and Jonsson, A. 2018. CP4TP: A Classical Planning for Temporal Planning Portfolio. In *Temporal Track of the International Planning Competition (IPC) 2018*.

Gréa, A.; Matignon, L.; and Aknine, S. 2018. HEART: HiErarchical Abstraction for Real-Time Partial Order Causal Link Planning. In *Proceedings of the 1st ICAPS Workshop on Hierarchical Planning at the 28th International Conference on Automated Planning and Scheduling*, 17–25.

Kiam, J. J.; Bercher, P.; and Schulte, A. 2021. Temporal Hierarchical Task Network Planning with Nested Multi-Vehicle Routing Problems – A Challenge to be Resolved. In *Proceedings of the 4th ICAPS Workshop on Hierarchical Planning at the 31st International Conference on Automated Planning and Scheduling*, 71–75.

Muscettola, N.; Smith, S.; Cesta, A.; and D'Aloisi, D. 1992. Coordinating space telescope operations in an integrated planning and scheduling architecture. *IEEE Control Systems*, 12(1): 28–37.

Parimi, V.; Rubinstein, Z. B.; and Smith, S. F. 2022. T-HTN: Timeline Based HTN Planning for Multiple Robots. In *Proceedings of the 5th ICAPS Workshop on Hierarchical Planning 32nd International Conference on Automated Planning and Scheduling*, 59–67.

Pellier, D.; Fiorino, H.; Grand, M.; Albore, A.; and Bailon-Ruiz, R. 2022. HDDL 2.1: Towards Defining an HTN Formalism with Time. ArXiv:2206.01822 [cs].

Pollack, M. E.; Joslin, D.; and Paolucci, M. 1997. Flaw Selection Strategies for Partial-Order Planning. *Journal of Artificial Intelligence Research*, 6: 223–262.

Rahmani, H.; Shell, D. A.; and O'Kane, J. M. 2021. Planning to Chronicle. In *Proceedings of the Algorithmic Foundations of Robotics XIV Conference*, 277–293.

Smith, D. E.; and Cushing, W. 2008. The ANML Language. In *Proceedings of the Workshop on Knowledge Engineering for Planning and Scheduling at the 18th International Conference on Automated Planning and Scheduling*.

Turi, J.; and Bit-Monnot, A. 2022. Guidance of a Refinement-based Acting Engine with a Hierarchical Temporal Planner. In *Proceedings of the Workshop on Integrated Action and Execution at the 32nd International Conference on Automated Planning and Scheduling*.

United States National Aeronautic Space Administration. 2023. Planning & Scheduling Group. https://www.nasa.gov/intelligent-systems-division/autonomous-systems-and-robotics/planning-and-scheduling-group/ (Last accessed: May 24, 2024).

Winer, D. R.; and Cardona-Rivera, R. E. 2018. A Depth-Balanced Approach to Decompositional Planning for Problems where Hierarchical Depth is Requested. In *Proceedings of the 1st Hierarchical Planning Workshop at the 28th Conference on Automated Planning and Scheduling*, 1–8.

Yorke-Smith, N. 2005. Exploiting the Structure of Hierarchical Plans in Temporal Constraint Propagation. In *20th AAAI National Conference on Artificial Intelligence*.

Younes, H. L.; and Simmons, R. G. 2003. VHPOP: Versatile Heuristic Partial Order Planner. *Journal of Artificial Intelligence Research*, 20: 405–430.

Young, R. M.; and Moore, J. D. 1994. DPOCL: A Principled Approach to Discourse Planning. In *Proceedings of the 7th International Workshop on Natual Language Generation*, 13–20.