30th International Conference on
Automated Planning and Scheduling

October 19 – 30, 2020, ~~Nancy (France)~~ online!

IPC 2020

Proceedings of the

10th International Planning Competition:

Planner and Domain Abstracts –

Hierarchical Task Network (HTN) Planning Track

Preface

Since its first edition in 1998, the International Planning Competition (IPC) has been an integral event of the planning community. For more than 20 years, it established unified input languages for planners, enabled an objective comparison between them based on an accessible benchmark set. The IPC drove the development of planners and fostered research. Thus, the IPC enabled planning researchers to compare their own work against the work of others – not only within the competition, but also outside of it. Due to the IPC almost all contemporary planners understand (some form of) PDDL, which allows for using IPC benchmarks across a multitude of planners.

The first two IPCs had – in addition to the regular track – a track on hand-tailored planners in which the planners could be provided with additional information or select their algorithms based on the input domain. Among these planners, some used Hierarchical Planning – most notably SHOP. Following the second IPC in 2000 the hand-tailored track was discontinued. Hierarchical planning was thereafter not part of the IPC any more. Research in the field however continued.

The International Planning Competition 2020 features for the first time a track dedicated to hierarchical planning. In contrast to the previous track on hand-tailored planners we don't want to evaluate how good planners can become given any possible additional knowledge, but ask how well planners can exploit a given hierarchical refinement structure. We therefore faced several unique challenges. We had to establish a common input language for all planners such that all of them operate on the very same model. We also had to specify a plan-output format and provide a verifier, since we had to ensure that the found plans satisfy the decompositional structure of the given task hierarchy. Further, we had to gather a comprehensive set of benchmark domains, since no such set existed before. We hope that this first competition for Hierarchical Task Network planners will foster future research into hierarchical planning and provide a common basis for many researchers – by establishing a unified input language, a common benchmark set, and an evaluation of the state of the art in HTN planning. We hope that many future editions of this competition will follow.

Gregor, Daniel, and Pascal
Organizers of the IPC 2020,
May 2021

Organizing Committee

Gregor Behnke	University of Freiburg, Germany
Daniel Höller	Saarland University, Saarbrücken, Germany
Pascal Bercher	The Australian National University, Canberra, Australia

Timeline of the Competition

Tracks Announced	October 18th 2019
Initial Submission of Planners	June 9th 2020
Submission of Benchmark Domains	June 9th 2020
Final Submission of Planners	August 9th 2020

Input Language

The competition used domains and problems formulated in HDDL.[1] In addition to the original HDDL definition, we require support for universal quanfication in preconditions. We have made some technical clarifications regarding allowed type names and requirements.[2] For convenience, we provide an automated translation of HDDL into (J)SHOP2 syntax.[3] Note that there are domains which can be expressed in HDDL, but for which there is no equivalent SHOP2 model. The issue lies in the restrictions SHOP2 puts on the allowed ordering. If competitors needed a different input format (notably SIADEX), we provided assistance and added a translator from HDDL into the needed input format. These translators are always included in the planner directly, i.e. any competitor was given the same HDDL file as input.

Output Format

We have defined an output format in which the planners have to provide their plans, which includes the decompositions that were applied in order to obtain them.[4] We have furthermore developed a verifier that checks the correctness of the returned plan against the domain and problem file it was derived for.[5] Mauricio Cecilio Magnaguagno kindly created a tool for visualising the output format.[6]

[1]HDDL – A Language to Describe Hierarchical Planning Problems. Daniel Höller, Gregor Behnke, Pascal Bercher, Susanne Biundo, Humbert Fiorino, Damien Pellier, and Ron Alford. Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI 2020). AAAI Press. 9883–9891. 2020.

[2]https://gki.informatik.uni-freiburg.de/competition/hddl.pdf

[3]https://github.com/panda-planner-dev/pandaPIparser

[4]https://gki.informatik.uni-freiburg.de/ipc2020/format.pdf

[5]https://github.com/panda-planner-dev/pandaPIparser

[6]https://maumagnaguagno.github.io/HTN_Plan_Viewer/

Tracks

Based on the results of a community questionnaire, we originally announced that the 2020 IPC will feature three different sub-tracks. The technical meaning of the tracks will be explained below.

- partial-order, recursive
- total-order, recursive
- total-order, non-recursive

After domain and planner submission we have however decided to cancel the third track. All planners that wanted to participate in the third track also wanted to participate in the second track. Further only three non-recursive domains were submitted: Barman-BDI, Childsnack, and Woodworking.

All tracks share common restrictions on the input. All preconditions and effects contain only literals, negated literals, conjunctions, and universal quantifiers (forall), all actions have unit-cost, and methods may contain preconditions (conjunctions of literals). For the specification of the tracks and technical requirements we defined the meaning of the following words.

- total-order: A domain is totally ordered iff the subtasks in all methods and in the initial task network form a sequence, i.e. the declared ordering arranges the tasks in a sequence.
- partial-order: A domain is partially ordered iff it is not totally ordered, i.e. there is at least one method whose subtasks are not totally-ordered or the initial task network is not a sequence.
- non-recursive: A domain is non-recursive if there is a total order on its lifted tasks (primitive and abstract) such that for every method that decomposes an abstract task A, the tasks in the methods' task network occur after A in the total order. In other words, it is not possible to derive a task network containing a task A by decomposing the task A.
- recursive: A domain that is not non-recursive.
- Method precondition: A list of literals that may be associated with a method. A method may only be applied if its method precondition holds, as explained next. For totally-ordered HTN planning problems, the precondition has to hold in the state in which the first action (that originates via further decompositions from the method's subtasks) is executed. For partially-ordered HTN planning problems, the precondition has to hold in some state before the first action (that originates via further decompositions from the methods' subtasks) is executed, but after the state resulting from the last necessarily preceding tasks for the tasks contained in the method.[7]

All data of the IPC 2020 can be found at http://ipc2020.hierarchical-task.net.

[7]See the paper describing HDDL for a more detailed discussion.

Rules

To evaluate planners, we used the agile metric used in the last IPC, i.e. a planner is regarded better wrt. that metric if it finds any solution to the problem faster. The score of a planner on a solved task is 1 if the task was solved within 1 second and 0 if the task was not solved within the resource limits. If the task was solved in t seconds ($1 \leq t \leq 1800$) then its score is $\min\{1, 1 - \frac{\log(t)}{\log(1800)}\}$. The score of a planner is the sum of its scores for all tasks.

For each problem, each planner is given 1 CPU core, 8 GB of memory, and 30 minutes of runtime. All produced plans were verified. If an invalid plan is returned, all tasks in the domain are counted as unsolved. If that happens in more than one domain, the planner is disqualified.

Domains

In the IPC 2020, we have used the following domains. If this booklet contains a paper describing the domain, we reference to it.

Partial Order Track

- Barman-BDI (page 39)
- Monroe-Fully-Observable[8]
- Monroe-Partially-Observable[8]
- PCP (page 24– 25)
- Rover
- Satellite (page 40– 42)
- Transport
- UM-Translog
- Woodworking (page 43– 44)

Total Order Track

- AssemblyHierarchical (page 19– 20)
- Barman-BDI (page 39)
- Blocksworld-GTOHP (page 21– 23)
- Blocksworld-HPDDL (page 31)
- Childsnack (page 21– 23)
- Depots (page 21– 23)
- Elevator-Learned-ECAI-16 (page 26– 30)
- Entertainment
- Factories-simple (page 45– 46)
- Freecell-Learned-ECAI-16 (page 26– 30)
- Hiking (page 21– 23)

[8]Partially Described in: Plan and Goal Recognition as HTN Planning. Daniel Höller, Gregor Behnke, Pascal Bercher, and Susanne Biundo. Proceedings of the 30th International Conference on Tools with Artificial Intelligence (ICTAI 2018). IEEE Computer Society. 466–473. 2018.

Awards

Partial Order Track

Total Order Track

[9]Described in: Construction-Planning Models in Minecraft. Julia Wichlacz, Alvaro Torralba, and Jörg Hoffmann. Proceedings of the Second ICAPS Workshop on Hierarchical Planning. 1–5. 2019.

Table of Contents

Addressing HTN Planning with Blind Depth First Search[*]

Juan Fernandez-Olivares, Ignacio Vellido, Luis Castillo
Department of Computer Science and Artificial Intelligence
University of Granada
faro@decsai.ugr.es, ignaciove@correo.ugr.es, l.castillo@decsai.ugr.es

Abstract

This paper briefly describes SIADEX, the HTN planner winner of the Partial Order track in the 2020 International Planning Competition. We also show a discussion of the results regarding run time and memory usage for the different problems configured in the competition.

Introduction

In this work we describe SIADEX, an HTN planner (Castillo et al. 2006) based on the same foundations than SHOP2 (Nau et al. 2003). It follows a progression search with a blind depth first search process and allows for partially ordered and recursive tasks networks. The planner was designed as a simple search process yet successfully applied to many real applications (Fdez-Olivares et al. 2006; González-Ferrer et al. 2013; Fdez-Olivares et al. 2019; Fernandez-Olivares and Perez 2020). Its power lies in that it is guided by the knowledge represented in the HTN planning domain, which is written in the hierarchical planning language HPDL (Castillo et al. 2006; González-Ferrer et al. 2013). This language is a hierarchical extension of PDDL 2.2 level 3 (Edelkamp and Hoffmann 2004), so the planner can reason about numeric and temporal information. The 2020 International Planning Competition (IPC2020) is concerned with only STRIPS-like domains, thus these features are obviated in this brief description. We refer to Castillo et al. (2006) and González-Ferrer et al. (2013) for a detailed description of the temporal and numeric capabilities of both the planner and the language.

SIADEX is the winner of the Partial Order track in the IPC held at ICAPS2020. A summary of the results obtained by the planner in this competition is shown in Table 1. SIADEX solved 95 out of 224 (42% overall coverage) problem instances configured in the competition. It shows 100% coverage (solved all the instances) and scores above 0.92 in 3 domains (Barman, Satellite, UMTranslog). For Barman and Satellite it scored 1, what means that it took less than 1 second to solve all the instances. In Rover domain the coverage and score are reasonable (70% and 0.70 respectively). But in domains for which it scores under 0.7, SIADEX shows poor coverage. This may be due to the fact that the task networks in Barman, Rover, Satellite and UMTranslog contain advice to solve the problems, and in other domains the planning knowledge embodies recursive tasks that cannot be adequately handled by the blind DFS search process implemented. For example, in Transport domain or Monroe (either full or partially observable alternatives) recursive tasks are defined as left recursive, while in UMTranslog are right recursive, with additional methods to appropriately guide to a hierarchical, recursive path planning process. In PCP domain SIADEX could not solve any instance. A detailed explanation of the results is shown below in the following sections. It is important to remark that in the verification tests SIADEX did not provide invalid plans for any instance.

Id	Domain	Instances	Solved	% Cov.	Score
4	PCP	17	0	0	0.00
7	Transport	40	1	2.5	0.03
3	Mon-PO	25	2	8	0.05
9	WoodW	30	3	10	0.10
2	Mon-FO	25	8	32	0.24
5	Rover	20	14	70	0.70
1	Barman	20	20	100	0.92
8	UM-Tra	22	22	100	1.00
6	Satellite	25	25	100	1.00

Table 1: Summary of the results of SIADEX in IPC2020. The columns describe the domain id for the Partial Order track, its name, number of instances configured, number of instances solved, coverage expressed as a percentage, and the score calculated as $min\{1, log(t)/log(T)\}$ for t the time to solve all the instances in each domain and $T = 1800$ seconds.

In the following sections we provide a brief, yet detailed description of the main features of SIADEX. Then we discuss the results of SIADEX analyzing its behavior in each domain, ending the paper with some concluding remarks.

[*]This work has been partially supported by Spanish Government Project MINECO RTI2018-098460-B-I00

SIADEX in a nutshell

As explained above, SIADEX is based on a non-informed search process, i.e., the planner does not use numeric heuristic information (the states are not evaluated in any way), thus the only heuristic used to guide the search is that represented in the HTN domain. That is to say, both the search process and the language are strongly headed to provide advice to the planning process. Search nodes are stored in a stack (fringe set) that is efficiently implemented and represents not only the basic information in standard HTN progression search (world state, current task network, solution plan prefix) but an agenda that stores the pending planning decisions. Since the planner follows a lifted planning approach, the main planning decisions (choice points in the search process) are related to tasks unification (either primitive or compound), precondition unification (either actions or methods), and alternative methods to be applied to a compound task (alternative methods for any task are stacked in a linear structure, therefore tasks's methods resemble if-then-else control structures).

HPDL incorporates almost all the features of HDDL (Höller et al. 2020), the language used as standard in this competition, and there is a specific translator from HDDL to HPDL based on the parser provided for the competition. HPDL primitive actions can be defined either as non temporal PDDL actions or PDDL durative actions, inheriting all the features that PDDL provides for preconditions and effects. Compound tasks are defined with a header (name and list of typed parameters) and have associated a set of methods to map compound tasks to predefined task networks. Every method inherits the parameters of the compound task it is associated with, but the language allows for the use of additional variables in a method if necessary. This feature allows for easily translating HDDL methods descriptions, which have their own list of typed parameters, into HPDL methods without affecting their semantics.

Furthermore, HPDL allows to restrict the type of a parameter inside a method, it is even possible to use variables in a method which are not defined in the paramaters of its associated task. That is, a method in HPDL can have a different set of variables than the parameters of the task its is attached to, just as happens in the methods of HDDL or SHOP. Anyway, we think that the syntax of HDDL is better than HPDL for this feature, in the sense that it directly provides a way to define the "local" parameters of a method. On the other hand, maybe HPDL provides more flexibility to manage variables, since one can use as many variables as needed in the "body" of a method without the need to define them in a "header" (HDDL forces to do it).

Method preconditions are described in the same way than PDDL actions preconditions (allowing for universal quantification), but the language extends preconditions expressions with additional features, like a special predicate to *bind* variables to symbolic or numerical expressions, and a *sort-by* structure, borrowed from SHOP2 (Nau et al. 2003), to provide an order between variable unifications according to a given criterion.

Task networks are represented as a partially ordered set of compound/primitive tasks, allowing for recursive definitions. The language also allows for the definition of *inline* tasks, i.e., tasks without name nor parameters which can be defined on-the-fly, which are interpreted as primitive actions and mostly used carry out *ad hoc* inference, by asserting or retracting facts in the world state. In real applications, this is a very welcome feature for knowledge engineers. Regarding the syntax to specify partially-ordered task networks, we borrowed from SHOP the two task ordering operators: the `(:ordered t1 .. tn)` of SHOP in HPDL is `(t1 .. tn)` and `(:unordered t1..tn)` is `[t1 .. tn]`, where `ti` does not refer to the label of a task, but to the header of the task itself. It is possible to combine them, for example, `(t1 [t2 t3] t4)`. We think that using labels for tasks (like in HDDL or in the former proposal of Erol, Hendler, and Nau (1994)) is more expressive since some ordering patterns cannot be expressed with () and []. For example, the following ordering constraint expressed in HDDL `:ordering ( (< t1 t4) (< t2 t4) (< t2 t5) (< t3 t5))` cannot be expressed with this syntax unless using ad-hoc predicates to force the ordering between `[t1, t2]` and `t4`, and `[t2, t3]` and `t5`. Nevertheless, HPDL embodies the capability to represent temporal constraints over start/end points of tasks (either primitive or compound) allowing to describe more expressive ordering patterns (in fact, all the relations of Allen's algebra can be represented (Castillo et al. 2006)). We think that it would be possible to define a compilation process that translates HDDL ordering constraints into HPDL decomposition methods including temporal constraints, but this feature is not used in the current competition.

Results discussion

Regarding runtime and memory usage, Tables 2 and 3 show that in general SIADEX uses around 0.3 seconds and 3.6 KB to solve almost every single instance, without practically no dispersion, except for the hardest problems in domains BarmanBDI and Monroe. Further, we have observed in the datasets of the competition that PyHiPOP (the other nondisqualified contestant) shows more runtime and memory usage than SIADEX (overall, at least an order of magnitude higher). This may partially be due to the fact that data structures in SIADEX are better handled. For example, the representation of nodes in the fringe set in SIADEX is optimized in such a way that, for each node, only the necessary information to revert the changes in case of backtracking is stored. This would allow SIADEX to be more efficient in both planning time and memory usage

In the following we briefly describe the results for each domain, aimed to analyse how complex are the problems proposed to SIADEX, as well as the diversity in the complexity of the instances.

PCP domain. SIADEX was unable to solve any instance in this domain. It is the only domain based on a propositional representation, and we think that this poor behaviour may be due to some failure in de parser from HDDL to HPDL, an issue that needs further study.

Wood Working domain. In this domain 3 out of 30 instances (10% coverage) were solved with a peak memory average around 3.6KB without almost variation among the

Id	Barman [1,20]		Monr-FO [21,45]		Monr-PO [46,70]		PCP [71,87]		Rover [88,107]		Satellite [108,132]		Transport [133,172]		UM-Tr [173,194]		WoodW [195,224]	
1	0.3	20	U	U	U	U	U	U	0.2	16	0.2	5	0.2	8.0	0.3	26	U	U
2	0.3	44	U	U	U	U	U	U	0.3	27	0.2	5	U	U	0.3	13	0.3	3
3	0.3	68	U	U	U	U	U	U	0.3	28	0.2	7	U	U	0.3	14	U	U
4	0.3	42	U	U	U	U	U	U	0.3	19	0.2	11	U	U	0.3	11	0.3	7
5	0.3	65	U	U	U	U	U	U	0.3	39	0.2	10	U	U	0.3	12	U	U
6	0.4	90	28.00	27	U	U	U	U	U	U	0.2	10	U	U	0.3	9	0.3	6
7	0.3	46	U	U	U	U	U	U	0.3	55	0.2	9	U	U	0.3	7	U	U
8	0.3	68	U	U	U	U	U	U	0.3	67	0.2	13	U	U	0.3	9	U	U
9	0.3	94	U	U	U	U	U	U	0.3	65	0.3	17	U	U	0.3	11	U	U
10	0.3	70	17.78	40	U	U	U	U	0.3	75	0.3	12	U	U	0.3	11	U	U
11	0.5	142	U	U	U	U	U	U	0.3	75	0.2	16	U	U	0.5	37	U	U
12	0.4	114	U	U	U	U	U	U	0.3	38	0.2	16	U	U	0.3	17	U	U
13	0.8	214	0.56	28	0.33	35	U	U	0.3	98	0.2	15	U	U	0.3	27	U	U
14	0.4	114	0.82	35	U	U	U	U	U	U	0.2	16	U	U	0.3	9	U	U
15	0.9	238	U	U	129	35	U	U	U	U	0.3	17	U	U	0.3	10	U	U
16	1.8	334	2.80	22	U	U	U	U	0.3	97	0.3	19	U	U	0.3	11	U	U
17	4.2	478	27.78	27	U	U	U	U	U	U	0.3	22	U	U	0.3	9	U	U
18	11.0	718	28.09	25	U	U	-	-	0.5	115	0.3	22	U	U	0.3	9	U	U
19	23.2	958	U	U	U	U	-	-	U	U	0.3	28	U	U	0.3	30	U	U
20	41.7	1198	1.69	24	U	U	-	-	U	U	0.3	27	U	U	0.3	17	U	U
21	-	-	U	U	U	U	-	-	-	-	0.2	18	U	U	0.3	26	U	U
22	-	-	U	U	U	U	-	-	-	-	0.3	25	U	U	0.3	18	U	U
23	-	-	U	U	U	U	-	-	-	-	0.2	5	U	U	-	-	U	U
24	-	-	U	U	U	U	-	-	-	-	0.2	5	U	U	-	-	U	U
25	-	-	U	U	U	U	-	-	-	-	0.2	7	U	U	-	-	U	U

Table 2: Run time average (in seconds) and plan length for the instances solved by Siadex in the domains of the competition. "U" stands for unsolved and "-" for no configured. The instance index interval for each domain is shown under its name. The table only shows 25 rows for each domain, since this is the maximum number of instances that SIADEX reached to solve for all the domains.

instances. This is the only domain where PyHiPOP superseded SIADEX. Without a deeper understanding of the planning domain we are unable to provide an explanation of this fact, nevertheless, we think that PyHiPOP solved more instances (concretely instances 197 and 199) because of the use of unbound variables in the definition of the HTN problem.

Transport domain. 1 out of 40 instances (instance id 133) were solved, with a memory average of 0.00346KB and a runtime of 0.2 seconds, showing a similar memory usage than in other domains. On the other hand, PyHiPOP solved 2 out of 40 for the same domain with an average memory of 1.5 MBytes and an average runtime of 18.6 seconds, both almost without dispersion. The behaviour of SIADEX in this domain may be explained by the way in which the tasks used to recursively solve path planning problems are represented. The order in which methods have to be applied matters, and in the Transport domain the task *get-to* is represented as a left recursive task in which the first method to be used contains the recursive decomposition. If SIADEX firstly addresses the task decomposition with that recursive method, it easily falls into an infinitely recursive loop. This could have been fixed with an improvement of the parsing process, by identifying the appropriate order in which methods should be applied.

Monroe domain. In domain Monroe-Partially-Observable 2 out of 25 instances (instances 59 and 61) were solved, with a memory peak average of 3604.8 KB and 7773394.4 KB, and a runtime of 0.3 and 128.9 seconds, respectively. In Monroe-Fully-Observable 8 out of 25 (32% coverage) instances were solved with an important dispersion among the runtime and memory usage. In fact there is a coefficient of variation[1] of 0.85 in runtime values and 0.87 in memory usage, while in other domains (except BarmanBDI which amounts to 1) this coefficient is less than 0.17. In both domains run time and memory usage is clearly superior to that of other domains. The values found in that domain are an indication that the problems configured in the Monroe domain (in both versions) are diverse in complexity and harder to solve by SIADEX.

Rover domain. SIADEX solved 14 out of 20 problem instances (70% of coverage) with a score of 0.72. It was unable to solve the instances 93, 101, 102, 104, 106 and 107 of this domain. In this domain there is a 17% of variation with respect to the mean value in runtime and a 6% of variation for peak memory. This means that SIADEX solved all the problems in almost the same run time showing a similar use of memory in all of them. We can conclude that the 70% instances in this domain are of similar computational complexity and that the problems are not diverse.

[1]The coefficient of variation measures the significance of the standard deviation with respect to the average.

Id	Bar	M-FO	Rov	Sat	UM	Wwo
1	3.5	U	3.6	3.5	3.5	U
2	3.4	U	3.5	3.5	3.6	3.5
3	3.5	U	3.5	3.5	3.6	U
4	3.5	U	3.6	3.6	3.6	3.5
5	3.4	U	3.5	3.5	3.5	U
6	3.4	2286.6	0	3.5	3.4	3.7
7	3.5	U	3.6	3.5	3.4	U
8	3.4	U	3.5	3.7	3.5	U
9	3.5	U	3.4	3.5	3.5	U
10	3.5	1581.3	3.5	3.6	3.5	U
11	3.4	U	3.4	3.4	13	U
12	3.4	U	3.6	3.5	3.5	U
13	3.5	20.1	3.6	3.7	3.4	U
14	3.6	43.8	0	3.5	3.5	U
15	3.6	U	0	3.4	3.7	U
16	3.6	246.9	3.7	3.5	3.6	U
17	3.9	2281.6	0	3.6	3.5	U
18	11.1	2558.3	3.5	3.5	3.5	U
19	22.6	U	0	3.5	3.5	U
20	31.2	136.1	0	3.5	3.6	U
21	-	U	-	3.5	3.4	U
22	-	U	-	3.5	3.6	U
23	-	U	-	3.5	-	U
24	-	U	-	3.5	-	U
25	-	U	-	3.4	-	U

Table 3: Memory usage of SIADEX (in KBytes). PCP, Transport and Monroe-PO are not shown, but described in the text. “U” stands for unsolved and “-” for no configured.

Barman-BDI domain. In this domain all the instances were solved with a score of 0.92. This domain shows the greater dispersion among the run time and memory usage values, therefore the problems configured are diverse (from the SIADEX perspective) and it seems that the problem configuration could be considered as a good testbed to evaluate the performance of the planner.

Satellite and UM-Translog domains. In both domains SIADEX achieved a 100% coverage and all the problems were solved in less than one second. Similar values in average run time and memory usage are shown, with very little variation, though some variation in the length of plans. These are the easier problems to SIADEX, and we think that for these domains there is more than only domain dynamics encoded in the decomposition methods.

Conclusions

In summary we are presenting an HTN planning algorithm based on DFS that handles almost all the features of HDDL. The planner shows a reasonable behaviour with respect to the domains an problems provided for the competition. All the domains and problems have been successfully translated from HDDL to HPDL, and the planner provides valid solutions in all the problems for which it finds one. However the planner does not provide solutions to several problems in concrete domains. We think that this is due to recursion problems which are difficult to overcome with a blind DFS process, and which in real applications are avoided by knowledge engineering, injecting specific knowledge to the planner, after a thorough analysis of the domain. In the domains without domain-specific advice for how to solve a problem, this planner is bound to lack behind those that apply heuristic guidance. Since SHOP also relies on expert knowledge, having a comparison with it would be very interesting. Anyway, our aim in this competition is to be aware of the situation of our planner with respect current developments, and to deeply analyse the limitations of our planning techniques in order to establish research directions for the improvement of both, the planning process and the planning language.

Acknowledgments

We are very grateful to the organizers of the competition, and specially to Gregor Behnke for his invaluable help and effort to develop the HDDL to HPDL parser. Many of their tests led us to discover both some limitations of our language, and ways to improve them in future versions of our planner.

References

Castillo, L.; Fernández-Olivares, J.; García-Pérez, Ó.; and Palao, F. 2006. Efficiently handling temporal knowledge in an htn planner. In *ICAPS*, 63–72.

Edelkamp, S., and Hoffmann, J. 2004. Pddl 2.2: The language for the classical part of the 4th international planning competition, albert ludwigs universität institüt fur informatik. Technical report, Germany, Technical Report.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. Htn planning: Complexity and expressivity. In *AAAI*, volume 94, 1123–1128.

Fdez-Olivares, J.; Castillo, L.; Garcıa-Pérez, O.; and Palao, F. 2006. Bringing users and planning technology together. experiences in siadex. In *Proc. ICAPS*, 11–20.

Fdez-Olivares, J.; Onaindia, E.; Castillo, L.; Jordán, J.; and Cózar, J. 2019. Personalized conciliation of clinical guidelines for comorbid patients through multi-agent planning. *Artificial intelligence in medicine* 96:167–186.

Fernandez-Olivares, J., and Perez, R. 2020. Driver activity recognition by means of temporal htn planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, 375–383.

González-Ferrer, A.; Ten Teije, A.; Fdez-Olivares, J.; and Milian, K. 2013. Automated generation of patient-tailored electronic care pathways by translating computer-interpretable guidelines into hierarchical task networks. *Artificial intelligence in medicine* 57(2):91–109.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. Hddl: An extension to pddl for expressing hierarchical planning problems. In *AAAI*, 9883–9891.

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

HyperTensioN
A three-stage compiler for planning

Maurício Cecílio Magnaguagno[1], **Felipe Meneguzzi**[2] and **Lavindra de Silva**[3]
[1]Independent researcher
[2]School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil
[3]Department of Engineering, University of Cambridge, Cambridge, UK
maumagnaguagno@gmail.com
felipe.meneguzzi@pucrs.br
lavindra.desilva@eng.cam.ac.uk

Abstract

Hierarchical Task Networks (HTN) planners generate plans using a decomposition process with extra domain knowledge to guide search towards a planning task. While many HTN domain descriptions are made by experts, they may repeatedly describe the same preconditions, or methods that are rarely used or possible to be decomposed. By leveraging a three-stage compiler design we can easily support more language descriptions and preprocessing optimizations that when chained can greatly improve runtime efficiency in such domains. In this paper we present the HyperTensioN HTN planner, as it was submitted to the HTN IPC 2020.

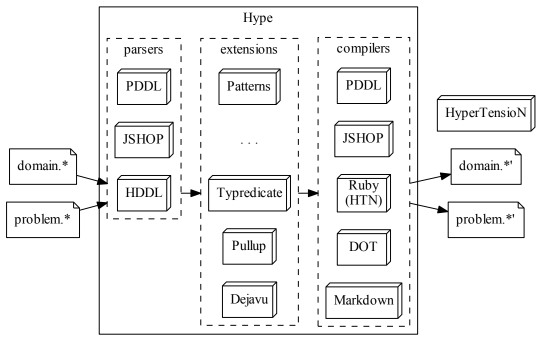

Figure 1: Hype acts as a three-stage compiler before linking Ruby outputs with the HyperTensioN planner.

Introduction

Hierarchical planning was originally developed as a means to allow planning algorithms to incorporate domain knowledge into the search engine using an intuitive formalism (Ghallab, Nau, and Traverso 2004). Hierarchical Task Network (HTN) is the most widely used formalism for hierarchical planning, having been implemented in a variety of systems rendered in different (though conceptually similar) input languages (de Silva, Lallement, and Alami 2015; Nau et al. 2003; Ilghami and Nau 2003). Recent research has re-energized work on HTN planning formalisms and search procedures, leading to a new generation of HTN planners (Bercher et al. 2017; Höller et al. 2018; Höller et al. 2020a; Höller et al. 2020b). In this paper, we outline key design elements, features, and optimizations of the HyperTensioN planner, as submitted to the 2020 International Planning Competition (IPC)[1]. Specifically, we focus on the compilation of HTN instances into Ruby programs, as well as the optimizations based on transformation of HTN domains and problems to minimize backtracking.

Three-stage design architecture

HyperTensioN was originally developed to automatically convert classical planning instances to hierarchical planning instances (Magnaguagno and Meneguzzi 2017). This required at least a PDDL (McDermott et al. 1998) parser (front-end) and a (J)SHOP (Ilghami and Nau 2003) description compiler (back-end). By keeping front-end and back-end separate it was also possible to add a Ruby compiler to generate code compatible with our implementation of a lifted Total Forward Decomposition (TFD) (Ghallab, Nau, and Traverso 2004, chapter 11) planner. This compilation approach is very similar to that in JSHOP (Ilghami and Nau 2003). Parser and compiler modules share an Intermediate Representation (IR) that represents the planning instance data, which middle-end extensions can further process. Extensions fill gaps between description languages, analyze or optimize descriptions, independent of the target planner/output language.

This level of flexibility facilitates developing support for new languages, while remaining compatible with the already available extensions. For example the DOT (Ellson et al. 2001) compiler for debugging and the HDDL (Höller et al. 2020a) parser for the IPC. As the project grew, the three-stage compiler and the TFD planner modules split in two, as shown in Fig. 1. The Hype tool controls module execution at each stage, allowing multiple middle-ends to run, even repeatedly, before compilation into the target representation. The HyperTensioN TFD planner completes the Ruby (HTN) compiler output to finish this pipeline with the plan output. Eventually, we extended the core HyperTensioN search procedure to a variety of other planning tasks, including search on hybrid symbolic-numeric domains (Magnaguagno and Meneguzzi 2020).

[1]ipc-2020.hierarchical-task.net

Domain transformation

To improve planning speed the compiler was optimized to compress the state structure by removing rigid predicates and treating them as "constant information". More importantly, we developed extensions to improve the IR to support: (1) better unification exploiting type information; (2) early testing of rigid parts of method/action preconditions during decomposition; and (3) a cycle detection mechanism.

Typredicate

This extension involves constraining the substitutions attempted on variables occurring in predicates, by making better use of constant/parameter types (if the domain expert has not already done so). For example, suppose the predicate *(at ?obj – object ?pos – position)* is defined in the domain, which is used in the action *(move ?obj – vehicle ?pos – position)* to both check and update a vehicle's position. Suppose also that we are given the following type hierarchy: "*person vehicle position – object*". Then, though the *move* action will never require nor modify the position of a person, the *?obj* variable occurring in the precondition of the action may still be substituted by constants of type *person*, as *?obj* is defined in the predicate to be of the parent type *object*. Since constants of type *person* and *vehicle* are mutually exclusive by virtue of being subtypes of the same parent type, we preclude such substitutions by specializing *(at ?obj – object ?pos – position)* into predicates *(at-vehicle ?obj – vehicle ?pos – position)* and *(at-person ?obj – person ?pos – position)*, and replacing occurrences of *(at ?obj ?pos)* with the appropriate specialized predicates in the domain, initial state and goal. Typredicate currently only specializes predicates to the leaves of the type hierarchy, but it can be straightforwardly extended to specialize to intermediate levels. Typredicate is not limited to typed domains, as it can infer types based on unary rigid predicates contained in preconditions, e.g. *(person ?obj)*. By specializing predicates we make planning more efficient, as unification uses smaller (disjoint) sets of objects, i.e., without extraneous objects, while also making the Pullup extension more "complete".

Pullup

The Pullup extension implements the main optimization technique that underpins HyperTensioN's performance by "pulling up" preconditions in the hierarchy. A literal in the precondition (which is a conjunction/set of literals) of an action occurring in a method is added (after variable substitutions) to the precondition of the method if the literal is not possibly brought about by an earlier step in the method, i.e., any solution for the method will require the literal to hold at the start; a literal is deemed to be possibly brought about (cf. "mentioned" (de Silva, Sardina, and Padgham 2016)) by a step if there is a literal asserted by an action yielded by the step s.t. the two literals have the same predicate symbol.[2] We pull up method preconditions as follows. A (possibly pulled up) literal in the precondition of a method is deemed to be part of the precondition of the task that is accomplished by the method if the literal is "locally rigid", i.e., shared by all method preconditions related to the same task. Given a planning problem, each iteration of the algorithm pulls up literals by one level, starting from preconditions of actions, and the algorithm terminates when it reaches a fixed point–when no literals "moved" in the previous iteration.

A literal that is always pulled up from an action/method precondition is removed from it, as the literal will be tested earlier in the decomposition. Moreover, using the planning problem, literals that are always true (w.r.t. the problem) are removed from preconditions based on the unifications that are possible, and actions/methods that contain contradictions in preconditions are removed together with their associated "branches". Interestingly, branch removal may enable pulling up additional literals by exposing "hidden" (see (de Silva, Sardina, and Padgham 2016)) rigid literals.

Dejavu

Some domains may have methods with direct recursion, where a method includes the same task that it decomposes, or indirect recursion, requiring further decomposition before the (same) task is encountered. Without "visited" predicates used by a domain expert to mark (register) and query visited partial states, such domains can induce an infinite loop for a TFD (Ghallab, Nau, and Traverso 2004) search procedure. Dejavu transforms the domain by adding "unobservable" primitive tasks (that are not part of valid plans) to mark and unmark the fact that a particular non-primitive task is being decomposed, and predicates to detect when the task is being recursively (re)attempted. Information relating to such cycles is stored across decomposition branches using an external cache structure, as the state loses the marked information upon backtracking. The cache saves which methods and unifications have been explored in previous branches to avoid repeating decompositions that previously led to failure. Domains with cyclic tasks without parameters lack the required information to cache the task signature, which contains the variable bindings for the method decomposing the task. In such domains we fallback to a full state comparison with previously visited states at each cyclic task. HyperTensioN can still detect stack overflows, and safely backtrack in case the cycle detection mechanism fails. Dejavu, while limited, proved critical for HyperTensioN's performance, as it allows TFD to efficiently drive search, while avoiding its key limitation in recursive domains.[3]

Comparison

We now compare the improvements obtained by the above extensions w.r.t. some sample (pre-competition) and accepted IPC 2020 domains. We selected 4 domains in which the improvements were more visible. The experiments were run on Windows 7, on an Intel E5500 2.8GHz CPU with 3.25GB of RAM, using a 60s time-out.

[2]We also implemented a stronger notion, closer to that of "mentioned", but saw no improvement w.r.t. the sample IPC domains.

[3]We only compare Dejavu's ability to detect indirect recursion, as detecting direct recursion is currently always active.

Woodworking

Woodworking (Bercher, Keen, and Biundo 2014) is based on a benchmark from earlier IPCs. It describes tasks for working with wood, such as cutting, polishing and finishing. With Pullup, two extra problems were solved within our time limit, with one of them taking less than a second as shown in Fig. 2. Many problems in this domain seemed to require selecting the right values among many available objects before continuing exploration, as otherwise too much time was spent on backtracking, causing time-out.

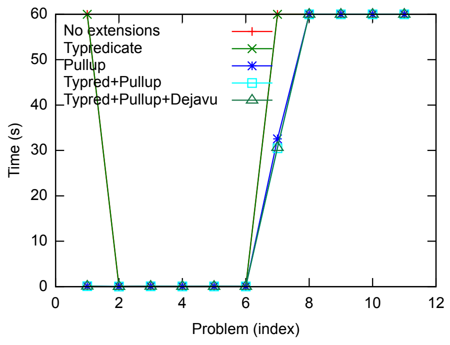

Figure 2: Time in seconds to solve Woodworking problems.

Rover

Rover involves robots navigating a planet, collecting information and sending it to a lander. The HTN domain was developed for SHOP (Nau et al. 1999) based on problem instances from earlier IPCs. Pullup improves the results for Rover, although most instances are solvable even without extensions; only one reaches the time-out as shown in Fig. 3.

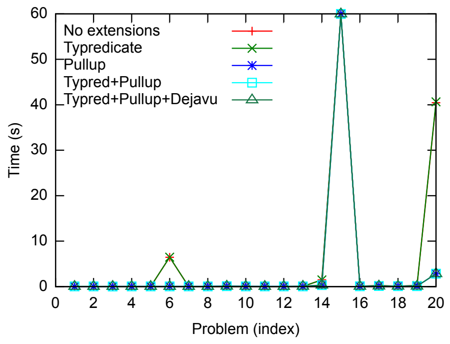

Figure 3: Time in seconds to solve Rover problems.

Transport

Transport (Behnke, Höller, and Biundo 2018) describes a domain where delivery trucks with limited capacity must pick and drop packages at specific cities connected by a road network. Transport is one of the few domains where each extension shows an impact on planning time, as shown in Fig. 4. Typredicate is able to specialize the "at" predicate, avoiding some unifications with non-vehicle objects. Pullup is able to move important constraints only defined in the leaves of the HTN structure, e.g. the need for a road between two cities in order to drive between them. Note that this domain does not contain method preconditions. With Typredicate and Pullup combined the Transport instances are solvable in less than 0.2s.

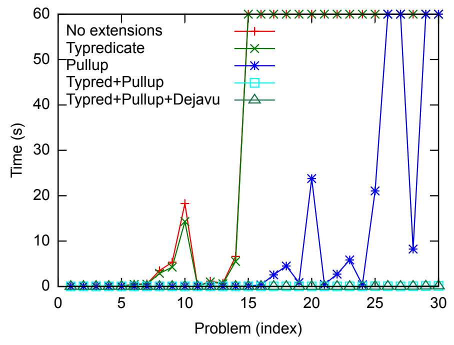

Figure 4: Time in seconds to solve Transport problems.

Snake

In Snake,[4] one or more snakes need to move to clear locations or strike nearby mice in a grid/graph-based world. The domain benefits from Dejavu, i.e., the planner avoids unifications that recursively expands the same task, which may start an infinite loop. Since Dejavu's direct recursion detection is always used, its effect is not visible in the graph, but required to avoid reaching the same positions repeatedly. Observe from Fig. 5 that Pullup shows a bigger improvement in planning time in the most complex instances.

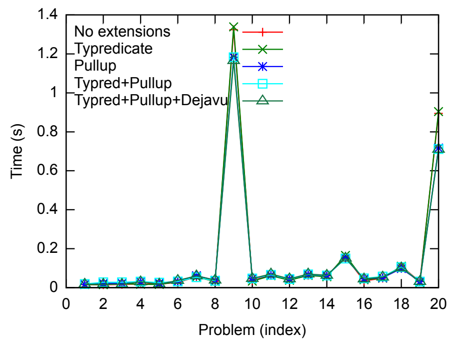

Figure 5: Time in seconds to solve Snake problems.

The IPC release of HyperTensioN was not able to parse the Entertainment and Monroe domains correctly. With the parser fixed, new results were obtained on the same machine, which matched the IPC samples' timings. The first 5 of 12 Entertainment instances were solved in under 1s, the sixth in 42s, the seventh in 740s, and the eighth in 235s; others exceeded the IPC time limit (1800s). All Monroe-Fully-Observable instances were solvable, most in a few seconds and the last two in 32s. The Monroe-Partially-Observable instances were not solvable within the time limit. The results

[4]https://github.com/Maumagnaguagno/Snake

Table 1: HyperTensioN's (Hype) fixed and IPC results.

Domain(instances)	Fixed	Hype	Lilotane	PDDL4J-TO
AssemblyHierarchical(30)	3	3	**5**	2
Barman-BDI(20)	**20**	**20**	16	11
Blocksworld-GTOHP(30)	16	16	**22.1**	16
Blocksworld-HPDDL(30)	**30**	**30**	1	0
Childsnack(30)	**30**	**30**	29	20.9
Depots(30)	**24**	**24**	23.4	23
Elevator-Learned(147)	**147**	**147**	**147**	2
Entertainment(12)	$\sim$**5.9**	0	4.6	4.6
Factories-simple(20)	3	3	**4**	0
Freecell-Learned(60)	0	0	**7.7**	0
Hiking(30)	**25**	**25**	21.3	17
Logistics-Learned(80)	22	22	**43.2**	0
Minecraft-Player(20)	**5**	**5**	1	1
Minecraft-Regular(59)	**57.1**	**57.1**	29.2	23
Monroe-FO(20)	$\sim$17.7	0	**20**	**20**
Monroe-PO(20)	0	0	**20**	1
Multiarm-Blocksworld(74)	**8**	**8**	4	0
Robot(20)	**20**	**20**	11	6
Rover-GTOHP(30)	**30**	**30**	21.3	27.5
Satellite-GTOHP(20)	**20**	**20**	15	**20**
Snake(20)	**20**	**20**	17.1	**20**
Towers(20)	**17**	**17**	10	16
Transport(40)	**40**	**40**	35	33.2
Woodworking(30)	7	7	**30**	6
Total(892)	$\sim$**567.7**	544.1	537.9	270.2
Normalized(24)	$\sim$**14.88**	13.50	11.60	7.47

are shown in Table 1 with the highest values (sometimes obtained by two participants) in bold. Only participants who obtained the highest value at least once were included.

Conclusion

This paper presented HyperTensioN, an approach to planning using a three-stage compiler designed to support optimizations in multiple domain description languages. The flexibility introduced by the front and back-end modules makes it easy to support new domain description languages, while the middle-end pipeline opens the door for multiple transformation and analysis tools to be executed before planning. The key to its performance in the IPC is a set of domain transformation techniques that replicates domain-knowledge optimizations commonly used to speed up search in previous HTN planners such as JSHOP2 (Ilghami and Nau 2003), as well as the optimizations often used by agent interpreters, e.g. (Thangarajah, Padgham, and Winikoff 2003). With our domain transformations it was possible to not only improve the HTN structure for SHOP-like (blind Depth First-Search) planners using Typredicate and Pullup, but also to avoid recomputing parts of complex combinatoric domains such as Transport and Snake using Dejavu. Future work includes stronger tree modifications/specializations, support for more complex domain descriptions, and a compilation to a low-level language to obtain a native planner executable.

Acknowledgements: Felipe Meneguzzi acknowledges support from CNPq with projects 407058/2018-4 (Universal) and 302773/2019-3 (PQ Fellowship).

References

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT-Totally-Ordered Hierarchical Planning Through SAT. In *AAAI*, 6110–6118.

Bercher, P.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An Admissible HTN Planning Heuristic. In *IJCAI*, 4384–4390. IJCAI.

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid planning heuristics based on task decomposition graphs. In *SoCS*.

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

de Silva, L.; Sardina, S.; and Padgham, L. 2016. Summary information for reasoning about hierarchical plans. In *ECAI*, 1300–1308.

Ellson, J.; Gansner, E.; Koutsofios, L.; North, S. C.; and Woodhull, G. 2001. Graphviz—open source graph drawing tools. In *GD*, 483–484. Springer.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated planning: theory & practice*. Elsevier.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2018. A Generic Method to Guide HTN Progression Search with Classical Heuristics. In *ICAPS*.

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 *AAAI*, 9883–9891. AAAI Press.

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

Ilghami, O., and Nau, D. S. 2003. A General Approach to Synthesize Problem-Specific Planners. Technical Report CS-TR-4597, Maryland University, Dept of Computer Science, College Park, Maryland.

Magnaguagno, M. C., and Meneguzzi, F. 2017. Method composition through operator pattern identification. *KEPS 2017* 54.

Magnaguagno, M. C., and Meneguzzi, F. 2020. HTN Planning with Semantic Attachments. In *AAAI*. AAAI Press.

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

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

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

Thangarajah, J.; Padgham, L.; and Winikoff, M. 2003. Detecting & Avoiding Interference Between Goals in Intelligent Agents. In *IJCAI*, 721–726.

Lifted Logic for Task Networks: TOHTN Planner Lilotane in the IPC 2020

Dominik Schreiber
Karlsruhe Institute of Technology
dominik.schreiber@kit.edu

Abstract

We present our contribution to the International Planning Competition (IPC) 2020. Our planner *Lilotane* builds upon ideas established by *Tree-REX* and encodes a Totally Ordered Hierarchical Task Network (TOHTN) planning problem into incremental formulae of propositional logic (SAT). *Lilotane*, however, instantiates reductions and actions lazily and minimalistically without the need for full grounding, hence accelerating the planning process significantly. We discuss the results of the IPC and conclude that *Lilotane*, scoring second in the Total Order track, is an overall competitive system, what demonstrates the viability of our approach and its significance for future research.

Overview

In this report we present *Lilotane* (’lɪ·lo·teɪn, *Lifted Logic for Task Networks*), the first Satisfiability (SAT) based planner for Totally Ordered Hierarchical Task Network (TOHTN) problems that operates on a lifted planning problem. The design of Lilotane is heavily motivated by the observation that grounding an HTN planning problem (Ramoul et al. 2017; Behnke et al. 2020) induces an unavoidable worst-case combinatorial blowup with respect to the input size, and that this blowup can hinder SAT-based HTN planners to scale to larger problems even if they are logically of simple nature. *Lilotane*, by contrast, fully circumvents the stage of grounding and instead encodes a lifted problem representation into propositional logic.

The general planning procedure of our planner is similar to the planning pipeline known from its predecessor *Tree-REX* (Schreiber et al. 2019) as well as from totSAT (Behnke, Höller, and Biundo 2018):

1. The formal description of a planning problem $\Pi = (D, s_I, T)$, where D is an HTN planning domain, s_I is an initial state, and T is a sequence of initial tasks, is parsed and preprocessed in some way.
2. Propositional logic clauses describing the problem’s upmost yet unencoded *hierarchical layer* L_l are added, a fully expanded task network is assumed, and a SAT solver is run on the resulting formula.
3. If the solver finds a model, a plan is decoded from the satisfying assignment to the Boolean variables and returned. Otherwise, go to 2.

Algorithm 1: Lilotane Procedure (simplified)

```
   Input: $\Pi = (D, s_I, T)$
   Result: Plan $\pi$
1  Preprocess $\Pi$;      // parsing, simplification
2  $H := \langle\rangle$;
3  $L_0 := \langle$ CreateInitialPosition$(T, s_I)$ $\rangle$;
4  Encode$(L_0)$;             // encode first layer
5  $H := H \circ \langle L_0 \rangle$;
6  for $l = 0, 1, \ldots$ do
       // instantiate new layer
7      $L_{l+1} := \langle\rangle$;
8      $S := (s_I, \emptyset)$;             // reachable facts
9      $x' := 0$;
10     for $x = 0, \ldots, |L_l| - 1$ do
           // generate child positions of $P_{l,x}$
11         $e_{l,x} := \max\{1, \max\{|subtasks(r)| \mid r \in P_{l,x}\}\}$;
12         for $z = 0, \ldots, e_{l,x} - 1$ do
13             $P_{l+1,x'} :=$ Instantiate$(P_{l,x}, z, S)$;
14             $L_{l+1} := L_{l+1} \circ \langle P_{l+1,x'} \rangle$;
15             $S := S \cup possibleFactChanges(P_{l+1,x'})$;
16             $x' := x' + 1$;
17         end
18     end
       // encode new layer
19     Encode$(L_{l+1})$;
       // finalize layer, attempt to solve
20     $H := H \circ L_{l+1}$;
21     $result :=$ Solve$(H)$;
22     if $result$ is SAT then
23         return Decode$(H, result)$;
24     end
25 end
```

The main difference between previous approaches and *Lilotane* is that the latter avoids the complete grounding of the problem in step 1; instead we perform lazy instantiation of operators and methods in step 2, just in time for when they are needed. We avoid to instantiate all free arguments of an action or a reduction occurring at some place of the hierar-

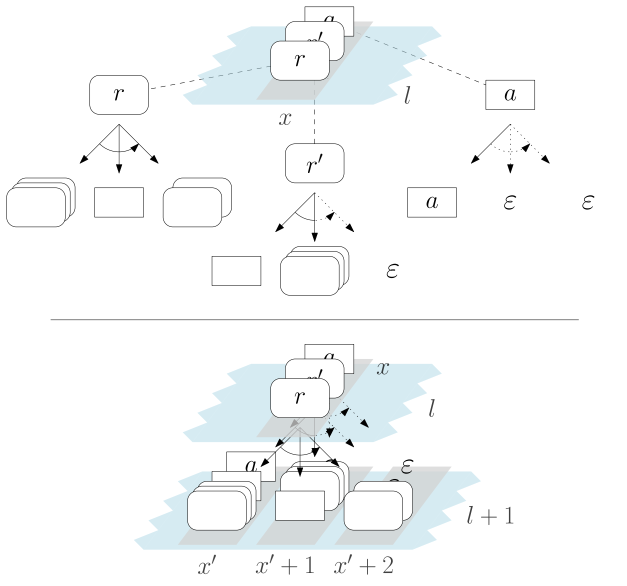

Figure 1: Sketch of *Lilotane*'s instantiation procedure. Above: Position x at layer l contains three operations (reductions r and r', and action a) with different possible children. Below: New positions $x', \dots, x' + 2$ are appended to layer $l + 1$ by aggregating the respective possible children.

chy. Instead we introduce *pseudo-constants* whose semantics we define directly in propositional logic. In this report, we briefly elaborate on these techniques in the upcoming sections, provide some technical insights, discuss the performance of *Lilotane* within the IPC, and provide a brief conclusion and outlook. For a more detailed presentation and discussion of the work at hand we refer to Schreiber (2021).

Instantiation

The base algorithm of our approach is illustrated in Alg. 1. After receiving the lifted problem description from *pandaPIparser*, we instantiate the problem's hierarchy from top to bottom, i.e., we begin with an initial layer following from the problem description (line 3) and then construct layer $l+1$ on the basis of layer l. Each operation (i.e., action or reduction) at some position of layer l can induce one or several new positions at layer $l + 1$. Each such new position may again feature a variety of different operations, as illustrated in Fig. 1. Only one such operation at each position will be chosen by a SAT solver for the final plan. This approach is based on *Tree-REX* (Schreiber et al. 2019) where the same layout of layers was used but its construction was based on a problem's full grounding. By contrast, we instantiate operations just when needed to achieve some subtask, and we preserve free arguments of methods instead of instantiating them with all possible combinations of constants.

As we instantiate each layer in chronological order ("from left to right"), we can maintain sets S of positive and negative facts which may possibly occur, beginning with the initial state (line 8) and adding any direct or indirect effects of inserted operations (line 15). We can use these fact collections to discard any operations with a precondition that turns out to be impossible to achieve in line 13. In line 15 we determine the possible effects of a given operation using a conservative overestimation which we compute by a traversal of the (lifted) recursive children of a method. In addition, we logically infer new preconditions for a method by recursively aggregating the preconditions and effects of its possible children: This helps us to profit from the described pruning methods even on domains which do not natively feature any method preconditions.

By allowing for free arguments to remain in an operation, we significantly reduce the number of instantiated actions and reductions. Consider an example task `(navigate ?rover ?from ?to)` which, according to its parent task `(investigate A)`, evaluates to `(navigate ?rover ?from A)`. Performing a conventional instantiation we receive tasks `(navigate R1 B A)`, `(navigate R1 C A)`, `(navigate R1 D A)`, `(navigate R2 B A)` and so on. Our algorithm avoids this blowup by instantiating only one task: `(navigate` α β `A)`. Thereby, α and β are new symbols which did not occur in the problem before and which we call *pseudo-constants*. With our novel SAT encoding we can let the solver decide which particular constant to substitute each pseudo-constant with. Our instantiation algorithm introduces a pseudo-constant whenever the valid domain of a free variable is larger than one, i.e., whenever there is a nontrivial choice to make regarding the substitution.

We introduced several further techniques to increase performance, such as (i) the sharing of pseudo-constants among multiple operations and the notion of an operation *dominating* another operation if it represents a superset of ground operations; (ii) the retroactive pruning of any subtree of operations which turned out to be impossible to achieve; (iii) the transformation of certain reductions into equivalent actions; and more.

Encoding

The general structure of our propositional logic encoding is an adaptation of the *Tree-REX* encoding (Schreiber et al. 2019). The main difference is that we now must deal with actions, reductions, and facts containing pseudo-constants. We now provide some central, slightly simplified clause definitions for illustration purposes and refer to Schreiber (2021) for the complete specification.

As in previous work we use one Boolean variable for each occurring reduction, action, and fact per position per layer of the problem. These variables are assigned regardless of whether the object contains pseudo-constants or not. Also, we have one variable *primitive*(l, i) representing whether position i at layer l features a primitive operation, i.e., an action and not a reduction.

In addition, we introduce global variables $[\kappa/c]$ that correspond to substituting some pseudo-constant κ with an actual constant c. For each pseudo-constant κ we add clauses

$$\bigvee_{c \in dom(\kappa)} [\kappa/c] \quad \wedge \bigwedge_{c_1 \neq c_2 \in dom(\kappa)} \neg[\kappa/c_1] \vee \neg[\kappa/c_2],$$

i.e., exactly one of the possible substitutions of κ with a constant from its possible domain, $dom(\kappa)$, must hold.

Next, we define the semantics of facts containing pseudo-constants, which we call pseudo-facts. Let f_p be a pseudo-fact and for each of its pseudo-constants κ let c_κ be one of the possible constants to be substituted such that substituting each κ with c_κ leads to ground fact f.

$$\Big(\bigwedge_{\kappa \in f_p} [\kappa/c_\kappa]\Big) \Rightarrow \big(holds(f_p, l, i) \Leftrightarrow holds(f, l, i)\big)$$

In words, we enforce a pseudo-fact to be equivalent to the ground fact it corresponds to when performing particular substitutions. This rule does imply that we need to fully instantiate all potentially occurring facts at the respective position; yet, we claim that there are commonly much fewer ground facts than there are actions or reductions.

Frame axioms are encoded only for ground facts, as the meaning of pseudo-facts is well-defined by the previous sets of clauses. We add clauses as follows:

(i) If a fact f changes its value, then either the position is non-primitive, or some action *directly* supports this fact change, or some pseudo-action *indirectly* supports the fact change. (ii) If fact f changes its value and action a from the indirect support is applied, then some set of substitutions must be active which unify some effect f_p of a with f.

Note that for (ii), in the general case a transformation of a Disjunctive Normal Form (DNF) into Conjunctive Normal Form (CNF) is required when a features many different pseudo-facts as effects which can be unified to f. We use a simple compilation which builds a tree of literals and then obtains CNF clauses by traversing it.

Compared to a SAT encoding based on a ground representation, there are some subtle new edge cases to consider. For instance, we need to add further clauses which constrain the sets of possible substitution combinations (due to invariant preconditions which we do not encode directly), retroactively restrict the domain of a pseudo-constant to incorporate argument type restrictions of a child operation, and conditionally disable certain negative action effects if an equivalent fact also occurs as a positive effect in the action.

We consider our new encoding to be structurally more complex than that of *Tree-REX* but observed empirically that our approach not only significantly cuts the time spent for instantiation but also leads to much smaller formulae, oftentimes by orders of magnitude.

Technical Remarks

Our planner is written from scratch in C++ (i.e., we did not reuse any code from previous planners). In the competition version we use SAT solver Glucose (Audemard and Simon 2009) with kind permission of the authors: Empirically we found this solver to work best on the class of formulae generated by our approach. We build upon *pandaPIparser* (Behnke et al. 2020) for parsing planning problems specified in HDDL and for performing light preprocessing tasks on the problem's lifted representation. *Lilotane* is free software licensed under the GNU General Public License (GPL) v3.0; additional legal constraints may apply depending on the licensing of the particular SAT solver *Lilotane* is compiled with. Our software is available at `github.com/domschrei/lilotane`.

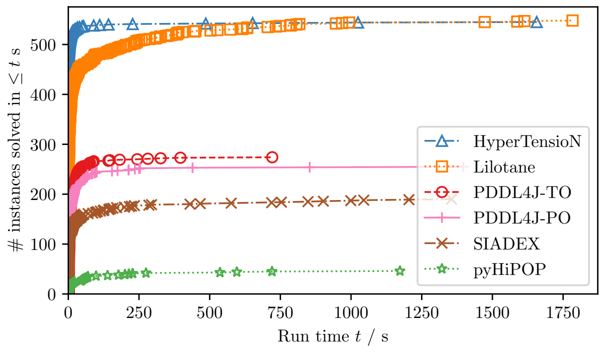

Figure 2: Run times overview of the IPC Total Order track. Each point (t, y) of participant p corresponds to an instance solved at least once by p in t seconds on average.

Post-IPC Discussion

We now discuss the results of the International Planning Competition (IPC) 2020.

A large set of diverse benchmark problems from various authors was gathered for the IPC, what will certainly facilitate thorough evaluations of TOHTN planners in the future. Compared to most previous evaluations in TOHTN planning (Schreiber et al. 2019; Behnke, Höller, and Biundo 2018) we observed that the peak difficulty of problems has been increased substantially: Oftentimes a domain known from previous evaluations was extended by ten more instances, each of which larger than any previous instance. This means that a planner reaching a near-perfect score on some domain is generally a much stronger result than before.

Lilotane scored the second place in the Total Order track of the IPC 2020. It found a plan for 548 out of 892 instances in at least one out of ten repetitions and reached an IPC score of 11.6. *Lilotane* was outperformed by progression search planner *HyperTensioN* which reached a considerably better score of 13.51 and found a plan for 545 instances in at least one repetition. *HyperTensioN* solved 84% of its instances in less than a second. *Lilotane* only solved 41% of its instances in under a second and solved 84% in under one minute. Overall we observed that while the IPC score benefits the overall much faster execution times of *HyperTensioN*, *Lilotane* performed similarly to *HyperTensioN* in terms of robustness and, unlike *HyperTensioN*, was able to solve some instance(s) on every single domain.

All further competitors scored significantly lower. In particular, *Lilotane* outperformed the only ground approach participating, *PDDL4J*, on all but four domains. *HyperTensioN* scored best on 15/24 domains and *Lilotane* scored best on 8/24 domains; only a single domain (Entertainment) was neither won by *HyperTensioN* nor by *Lilotane*.

Lilotane's worst performances are on the domains Blocksworld-HPDDL, Minecraft(-Player), and Multiarm-

Blocksworld. We noticed that each of these domains leads to deep and large hierarchical task networks which favor greedy progression search planners over planners such as *Lilotane* which are required to instantiate the entire hierarchy with all possible alternatives up to the layer where a plan can be found. Furthermore, compiled universal quantifications in Blocksworld-HPDDL and Multiarm-Blocksworld lead to many preconditions per operator which are comparably costly for our encoding.

By contrast, our planner excelled on domains such as Monroe (complex goal and task recognition problems on top of a disaster management domain, see Höller et al. 2018) and Woodworking. The latter domain encompasses large manifacturing and processing tasks and notably features a high number of arguments per operator and method. As our approach keeps free arguments lifted, it can handle this domain very well. We are also pleased to observe that *Lilotane* scored well on the Childsnack domain: This domain is a textbook example for a logically trivial domain which leads to huge ground representations. Hence, prior SAT-based approaches have considerable problems with this domain while our approach solves even large problems with relative ease.

Although the IPC was an agile competition where only run times were of interest, we also want to shed light on the length of the plans found by the best competitors (with respect to the number of actions in a plan). We found that *Lilotane* produced considerably shorter plans than the winner: We filtered out all 439 instances for which both *Lilotane* and *HyperTensioN* found a plan on some runs and then averaged the found plan length over all successful runs for each instance. On 264 instances *Lilotane* found shorter plans on average, on 77 instances the found plans are of equal average length and on 98 instances *HyperTensioN* found shorter plans on average. Summed up over all these instances, the number of actions reported by *HyperTensioN* corresponds to 229% of the number of actions reported by *Lilotane*. This significant difference in plan quality can be explained by the careful iterative deepening procedure of *Lilotane*: Any found plan length is bounded by the size of the layer where it was found, and *Lilotane* finds a plan on the very first layer where any plan *can* be found.

Conclusion and Outlook

We presented our submission to the IPC 2020 named *Lilotane* which is the first lifted SAT-based HTN planning system. *Lilotane* showed promising performance and convinced on a large and diverse set of benchmarks with respect to its robustness and the high-quality plans it finds. As such, the performance of *Lilotane* in the IPC 2020 demonstrates that SAT-based HTN planning without grounding is not only viable, but in fact a highly appealing approach if done carefully. We expect these results to open up new perspectives for SAT-based planning in related problem classes. We refer to a separate article (Schreiber 2021) which discusses the research at hand in more detail, provides proofs of correctness, and describes further improvements of *Lilotane* integrated after the planner submission deadline of the IPC.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500).

The author would like to thank Marvin Williams for his persevering exploration of lifted SAT encodings for classical planning (Williams 2020), motivating the author to pursue a lifted SAT encoding for HTN planning as well.

Furthermore, the author thanks the IPC organizers for their diligent work on this important competitive event and specifically Gregor Behnke for fruitful discussions regarding the results of the IPC 2020 and for providing an HDDL parser the author made thankful use of.

Last but not least, many thanks to Damien Pellier, Humbert Fiorino, and Tomáš Balyo who introduced the author to the exciting topic of SAT-based planning and HTN planning.

References

Audemard, G., and Simon, L. 2009. Predicting learnt clauses quality in modern SAT solvers. In *Twenty-first International Joint Conference on Artificial Intelligence*, 399–404.

Behnke, G.; Höller, D.; Schmid, A.; Bercher, P.; and Biundo, S. 2020. On succinct groundings of HTN planning problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, 9775–9784.

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, 6110–6118.

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

Ramoul, A.; Pellier, D.; Fiorino, H.; and Pesty, S. 2017. Grounding of HTN planning domain. *International Journal on Artificial Intelligence Tools* 26(05):1760021.

Schreiber, D.; Pellier, D.; Fiorino, H.; et al. 2019. Tree-REX: SAT-based tree exploration for efficient and high-quality HTN planning. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling*, 382–390.

Schreiber, D. 2021. Lilotane: A lifted SAT-based approach to hierarchical planning. *Journal of Artificial Intelligence Research* 70:1117–1181.

Williams, M. 2020. Partially instantiated representations for automated planning. Master's thesis, Karlsruhe Institute of Technology.

PYHIPOP– Hierarchical Partial-Order Planner

Charles Lesire[1] **and Alexandre Albore**[2]
ONERA/DTIS, University of Toulouse
2 av. Edouard Belin, 31055, Toulouse, France
[1]charles.lesire@onera.fr
[2]alexandre.albore@onera.fr

Introduction

PYHIPOP is a hierarchical partial-order planner, aimed at solving Hierarchical Task Network (HTN) planning problems. The current planner version is a re-coding of a version originally developed by Patrick Bechon (Bechon et al. 2014). In Bechon's original work, HIPOP solved HTN problems with Task Insertion (TI-HTN), meaning that inserting new tasks in addition to the pure HTN decomposition was allowed during the search. Bechon proposed some heuristics for solving such problems using a hybrid algorithm: a POP (Partial Order Planning) algorithm with hierarchical task decomposition. HIPOP has also been extended to manage plan repair (Bechon et al. 2015), and multi-robot mission planning repair with communications losses (Bechon, Lesire, and Barbier 2020).

PYHIPOP differs from the original HIPOP in:

- PYHIPOP is coded in pure Python3; this choice has been made to ease the integration with other tools for plan repair or interactive planning;
- PYHIPOP currently manages HTN problems only – no Task Insertion is allowed;
- PYHIPOP's preprocessing and grounding steps have been improved to use recent works from the state of the art;
- PYHIPOP's heuristics have been adapted, as its original heuristics worked well for TI-HTN, but not so well for pure HTN problems.

Implementation details

In the HTN paradigm, plans are not considered as totally ordered sequences of actions. When searching in the plan space, plans are rather a *partially ordered* sequence of actions, that the planner orders so to generate a solution plan.

HTN planning relies on the concept of task decomposition (Erol, Hendler, and Nau 1994). While the goal in classical (STRIPS-style/non-hierarchical) planning is to find an action sequence that drives the domain from an initial state to a goal final state, the goal in hierarchical planning is to find a refinement of an initial partial plan into a plan that contains no abstract tasks, nor flaws.

The Hybrid Planning domains $\mathcal{D}$ considered here, consist of a set of fluents, a finite set of abstract and primitive tasks, and a set of methods M that describe the different ways an abstract task can be decomposed. The goal is either a subset g of the fluents of $\mathcal{D}$, or a goal task top to decompose.

A partial plan Π is a tasks tree with its root in top. Partial plans may contain primitive and abstract tasks.

Given the constraints of $\mathcal{D}$, a natural ordering can be determined between tasks, such that for two instances $u \prec v$, u is first task that supports the fluent f, while v is the task that needs f as a precondition. This defines a causal link (u, f, v) between the two tasks.

In a partial order plan, we consider three kinds of *flaws*. Namely, *open links*, where no causal link guarantees the precondition of a task in the plan; *threats*, when a task could delete a fluent in a causal link while the link is still active; *abstract tasks*, when a non-primitive task is present. We will see that managing the flaws, and selecting the ones to be refined, is central for the planning algorithm performance.

Preprocessing and grounding

During the preprocessing phase, we ground all the operators, and we smartly prune the set of grounded operators, and compute some information useful during the search.

The proprocessing steps are the following:

- as often done in classical planning, we compute all the possible literals of the problem, by grounding the predicates on the objects, and we determine which literals cannot be modified by any operator (the *rigid* literals);
- all operators (actions, tasks, methods) are then grounded; we remove the groundings that are impossible due to known rigid literals (including equality tests), similarly to Behnke et al. (2020);
- we compute the h_{add} heuristics for literals and actions, based on the algorithm proposed by Vidal (2011);
- we compute the Task Decomposition Graph (TDG) (Bercher et al. 2017), and from it we prune actions/methods whose preconditions are not reachable according to the h_{add} relaxation, methods and tasks whose subtasks and methods have been respectively removed;
- based on the TDG, we compute: 1) the minimal cost when decomposing a task h_{TDG_c}, as proposed in (Bercher et al. 2017); 2) the maximal h_{add} cost in a decomposition,

noted h_{add}^{max}; 3) an optimistic task effect, consisting in the union of all effects of the actions in any possible task decomposition; this optimistic effect is inspired from angelic HTN planning (Marthi, Russell, and Wolfe 2008).

Search algorithm

The search of a solution plan is performed in the plan space. Any valid instance of the methods and tasks – meaning that they respect the ordering constraints of the problem – is a solution plan for a problem $\mathcal{D}$: all preconditions will be supported by a causal link that is not threatened, and all abstract tasks will be decomposed into primitive tasks. We designed a domain-independent search strategy, where a search node is given by a partial plan and an ordered set of its associated flaws (open links, threats, abstract flaws). The main search loop is described in Alg. 1.

Algorithm 1: Solve algorithm

```
1 $OPEN \longleftarrow \{top\}$;
2 while OPEN not empty do
3     $n \longleftarrow OPEN.pop()$;
4     if $n.flaws \neq \emptyset$ then
5         return $n.plan$;    // solution found
6     $f \longleftarrow n.flaws.pop()$;
7     for $r \in resolvers(f, n.plan)$ do
8         $OPEN \longleftarrow r$;
9 return Failure;
```

The search makes use of an Open list (a heuristically ordered queue representing the fringe of the search) and a closed list (not reported in Alg. 1) to detect and prune already visited nodes. In line 1, the Open list is populated with the initial node, including the initial partial plan and a single abstract flaw, represented by the abstract task *top*. In line 3, the most promising search node n is popped from the Open list, initially populated with the initial node *top*. Lines 4–5 check and return a solution. In line 6 we select the most promising flaw f of the current node. Lines 7–8 generate resolvers for f and insert the newly generated nodes r, with the partial plans and their respective flaws, in the Open list. The resolvers are the list of plans that solve the flaw f. An open link is solved by finding the causal links that add a needed precondition. A threat to an open link is solved by moving the execution of the threatening action before or after the open link. An abstract task is solved by refining it, instantiating methods or primitive tasks. During the computation of the resolvers, we look one step ahead, and verify if their flaws can be solved. When generating a resolver r, we check that: (1) threats can be solved (i.e., a threatening action has no ordering constraint stuck to it during the causal link), (2) open links may have a support, either from an action in the plan, or using the optimistic task effects computed during the grounding. If one of these condition is not fulfilled, r is discarded.

Heuristics

To perform a search in the space of plans, we use different heuristic functions to drive the search.

In the first place, a partial plan selection heuristic is used: in Alg. 1 at line 3. We order the nodes in the Open list following h_{add}: we sum the h_{add} values of the literals in open links, and the h_{add}^{max} of abstract flaw tasks, and use h_{TDG_c} to estimate the cumulative costs of the primitive actions in the plan. Secondly, a flaw selection heuristic is used at line 6. Flaws are ordered following their kind. We first solve threats, then open links, and eventually expand abstract flaws, as originally proposed by Bechon et al. (2014). Several heuristics are available to sort open links, based on the current plan partial-order, or on h_{add}. The competing implementation uses *earliest*: the open link from the action coming earlier in the plan are resolved first. Abstract flaws are also sorted using *earliest*: the tasks coming earlier in the plan are decomposed first.

Empirical evaluation

PYHIPOP participated to the 2020 IPC for Hierarchical Planning (Behnke, Höller, and Bercher 2020), in the Partial Order track, and the Total Order track. Here, a domain is partially ordered when the subtasks in all methods and in the initial task network may have any order (in opposition to the total-order, where the declared ordering arranges the tasks in a sequence). The evaluation was performed on a single CPU core, with 8 GB memory limit, and a cut-off time of $T = 30mn$.

For the competition, the planners were evaluated following a flexible metric, which evaluates better a planner when it finds any solution to a problem faster. The score of a planner on a solved task is 1 if the task was solved within 1 second and 0 if the task was not solved within the cut-off limits. If the task was solved in t seconds, with $1 \leq t \leq T$, then its score is $\min(1, 1 - log(t)/log(T))$. The IPC score of a planner is the sum of its scores for all tasks.

At the Partial Order track, three planners participated: SIADEX (de la Asunción et al. 2005) ended at the first place, PYHIPOP at the second place, and PDDL4J-PO (Pellier and Fiorino 2020). The latter was disqualified because it returned an invalid plan in more than one domain. It is the Partial Order track results that we're going to comment below.

The IPC score represents quite well both the coverage and the solving time (Table 1). For instance, in Satellite domain, SIADEX with a score of 1.0 finds a solution for all the instances (25 out of 25) within 1s, while PYHIPOP solves less instances (9/25) and scores 0.21. On Woodworking, PYHIPOP solves three instances more (6/30) than SIADEX (3/30), but employs more time, which is reflected in the slight score difference 0.05 versus 0.03.

The number of solved instances per planner is detailed in Table 2. During the IPC, all experiments were executed 10 times with a different seed, we consider here the maximum number of solved instances for each domain in all the seeds. PYHIPOP performs relatively well in Satellite, UM-Translog, and Woodworking domains. On the other hand, the planner terminates the search without a plan in Monroe-

Domain	*# inst.*	PYHIPOP	SIADEX
Monroe Full. Obs.	25	0.00	**0.24**
Monroe Part. Obs.	25	0.00	**0.05**
PCP	17	0.00	0.00
Rover	20	0.05	**0.70**
Satellite	25	0.21	**1.00**
Transport	40	**0.05**	0.03
UM-Translog	22	0.79	**1.00**
Woodworking	30	**0.13**	0.10
total	*204*	*1.24*	*3.12*

Table 1: IPC scores for PYHIPOP and SIADEX. *# inst.* indicates the total number of instances per domain.

Domain	*# inst.*	PYHIPOP	SIADEX
Monroe Full. Obs.	25	0	10
Monroe Part. Obs.	25	0	2
PCP	17	0	0
Rover	20	2	14
Satellite	25	9	25
Transport	40	4	1
UM-Translog	22	21	22
Woodworking	30	6	3
total	*204*	42	77

Table 2: Coverage for PYHIPOP and SIADEX. *# inst.* is the total number of instances per domain.

Fully-Observable, Monroe-Partially-Observable, and PCP. It solves few instances of Rover and Transport, while it times-out in the rest of them.

Comparing the winner and the runner-up planners performances is not an easy task, as the coverage differs greatly. In general, the average time for synthesising a solution plan is lower than $1s$ for both planners. In the case of PYHIPOP, then, the total time is split in parsing, grounding, and search time. For solved instances of Satellite, search time represents almost 100% of the total time: for *2obs-2sat-2mod* search is $\sim 100s$, while grounding is $\sim 1s$, but for other problems, the preprocessing and grounding can represent the whole time, mainly because of the creation of the TDG, e.g. in *UM-Translog 19-A-TankerTraincarHub*, search is $\sim 0.8s$ while the grounding takes $\sim 52.6s$. The bad performance of PYHIPOP in this first grounding step is one of the reason that a lot of instances could not be solved: PYHIPOP timed out event before the end of the grounding. While the computed TDG contains useful information for the search, the computation of the TDG itself is greedy: a complete TDG is first build, then the several prunings are applied one after the other. Instead, we should prune the TDG on-the-fly while building it, improving the performance of the grounding step.

The second reason why PYHIPOP performs not so well on some instances is that the heuristics used in the competition are mainly based on h_{add}. On the domains where h_{add} is not well informed (when the hierarchy is a lot more constraining than the establishment of causal links), then the substask decomposition in PYHIPOP can be inefficient, stuck in a search plateau where the open list contains a lot of plans with very close heuristics values.

Conclusion and future work

The PYHIPOP implementation, starting from the results by Bechon et al. (2014), extended the original POP algorithm with an improved preprocessing phase, and adapting the heuristic search for the HTN paradigm.

Future work is aimed at improving the search algorithm, developing different heuristics to be used in a multi-queues best-first-search setting, combining different aspects of the heuristic evaluation of the problem, without aggregating them into a single function. We hope that this will produce a more efficient and flexible planner, fitting complex multi-robot mission planning tasks.

Also, we will rewrite the grounding step in order to build and prune the TDG on-the-fly, making PYHIPOP able to tackle more complex instances.

In fact, PYHIPOP is thought to be applied to hierarchical robotic tasks. There, a future implementation including planning repair will solve issues with communication losses between robots, or sensor/actuator failures requiring mission on-the-fly modifications. In order to address these missions, we will re-introduce in PYHIPOP the management of durative actions and time constraints, as originally addressed in (Bechon, Lesire, and Barbier 2020).

References

Bechon, P.; Barbier, M.; Infantes, G.; Lesire, C.; and Vidal, V. 2014. HiPOP: Hierarchical Partial-Order Planning. In *Starting AI Researchers Symp. (STAIRS)*.

Bechon, P.; Barbier, M.; Lesire, C.; Infantes, G.; and Vidal, V. 2015. Using hybrid planning for plan reparation. In *European Conf. on Mobile Robots (ECMR)*.

Bechon, P.; Lesire, C.; and Barbier, M. 2020. Hybrid planning and distributed iterative repair for multi-robot missions with communication losses. *Autonomous Robots* 44(3-4):505–531.

Behnke, G.; Höller, D.; Schmid, A.; Bercher, P.; and Biundo, S. 2020. On succinct groundings of HTN planning problems. In *AAAI*, volume 34, 9775–9784.

Behnke, G.; Höller, D.; and Bercher, P. 2020. IPC for hierarchical planning. http://gki.informatik.uni-freiburg.de/competition, accessed 2021-01-26.

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

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

Erol, K.; Hendler, J.; and Nau, D. S. 1994. HTN planning: Complexity and expressivity. In *AAAI,* vol. 94, 1123–1128.

Marthi, B.; Russell, S.; and Wolfe, J. 2008. Angelic Hierarchical Planning: Optimal and Online Algorithms. In *ICAPS*.

Pellier, D., and Fiorino, H. 2020. Totally and Partially Ordered Hierarchical Planners in PDDL4J library. *arXiv preprint arXiv:2011.13297.*

Vidal, V. 2011. YAHSP2: Keep It Simple, Stupid. In *Int. Planning Competition.*

Totally and Partially Ordered Hierarchical Planners in PDDL4J Library

Damien Pellier, Humbert Fiorino
Univ. Grenoble Alpes - LIG
F-38000 Grenoble, France
{Damien.Pellier, Humbert.Fiorino}@imag.fr

Abstract

In this paper, we outline the implementation of the TFD (Totally Ordered Fast Downward) and the PFD (Partially ordered Fast Downward) hierarchical planners that participated in the first HTN IPC competition in 2020. These two planners are based on forward-chaining task decomposition coupled with a compact grounding of actions, methods, tasks and HTN problems.

Introduction

The TFD (Totally Ordered Fast Downward) and PFD (Partial ordered Fast Downward) hierarchical planners are based on forward-chaining task decomposition used by the SHOP2 planner (Nau et al. 2003) coupled with a compact grounding of actions, methods, tasks and HTN problems. Both planners accept as input HDDL (Hierarchical Domain Description Language) proposed by (Höller et al. 2020) and are implemented on top of the PDDL4J library (Pellier and Fiorino 2018). In this short paper we present first the compact representation used by TFD and PFD as well as the grounding procedure implemented. Finally, we conclude with a brief presentation of the search strategy implemented in both planners.

Grounding technique

Most modern planners work with grounded representations of the planning problem. However, planning domains are commonly defined with a lifted description language such as PDDL (Ghallab et al. 1998) or HDDL (Höller et al. 2020). Thus, planning systems have to generate a grounded representation of the lifted domain in a preprocessing step, the objective of which is to generate the most compact grounded representation possible without removing any action, method or fluent needed for a solution plan. The more compact the grounded representation is, the more efficient is the search for a solution plan as reducing the size of the search space speeds up search and heuristic value computation. In practice, computing a grounded representation from a lifted representation is quite straightforward. All possible instantiations of ground predicates, primitive actions, abstract tasks and methods must be computed, and appropriately replaced by their ground versions in the lifted representation.

In the context of classical (non hierarchical) planning, the planners FF (Hoffmann and Nebel 2001) and FastDownward (Helmert 2006) have implemented techniques for transforming lifted to ground planning representation that are still used in many planners today. Regarding hierarchical planning, (Ramoul et al. 2017) inspired from (Koehler and Hoffmann 1999) have been the first to propose an efficient grounding preprocessing, and it was successfully applied to the planners proposed by (Schreiber et al. 2019). Recently, (Behnke et al. 2020) has proposed novel techniques.

In the TFD and PFD planners, the grounding combines the approches proposed by (Ramoul et al. 2017) and (Behnke et al. 2020). It is based on 6 steps:

1. Encode the lifted domain into an integer representation,
2. Simplify the lifted representation and infer types from predicates,
3. Instantiate the set of actions by removing unreachable actions with respect to the inertia principle (Koehler and Hoffmann 1999),
4. Filter out actions grounded in reachability analysis (Hoffmann and Nebel 2001),
5. Instantiate the set of methods by removing unuseful methods with respect to the inertia principle, and by recursively decomposing the initial tasks network of the planning problem. Tasks that are not reachable, i.e., tasks for which there are no relevant ground actions or methods are pruned.
6. Encode the actions and the methods into bitset representation.

More details on the implementation of the different instantiation techniques are available in PDDL4J opensource repository: `https://github.com/pellierd/pddl4j`.

Search procedures

The non-deterministic TFD procedure for solving a HTN planning problem is given in Algorithm 1. This procedure is based directly on the recursive definition of a solution plan for HTN planning problems.

The TFD procedure takes as input a problem $P = (s_0, T, A, M)$ where s_0 is the initial state, $T =$

$\langle t_1, t_2, ..., t_k\rangle$ is a list of tasks, A the set of actions, and M the set of methods, all in their ground representation. First, the procedure tests if the list of tasks T is empty (line 2). In this case, no task has to be executed, thus the empty plan is returned. Then the procedure gets the first task t_1 of the list T. Two cases must be considered depending on the type of t_1:

Case 1. If t_1 is primitive (line 3) then the procedure computes the set of all the ground actions that accomplishes t_1 and that are applicable in s_0 (line 4). If there is no action (line 5), the procedure fails because no action accomplishes the goal task t_1. Then the procedure non-deterministically chooses an action that accomplishes the task (line 6), and calls itself recursively on the planning problem $P' = (\gamma(s_0, a_1), T - \{t_1\}, A, M)$ (line 7). Finally, if the recursive call to the procedure fails to find a plan π, it returns failure (line 8); otherwise it returns the plan that is the concatenation of a and π (line 9).

Case 2. If t_1 is non-primitive (line 10) then the procedure computes the set of ground decompositions that accomplish t_1 and that are applicable in s_0 (line 11). If there is no decomposition to accomplish t_1 (line 12) then the procedure returns failure. Otherwise the procedure non-deterministically chooses a decomposition d that accomplishes the task t_1 (line 13), and recursively returns the solution plan for the problem $P' = (s_0, \mathit{subtasks}(d) \oplus \langle t_2, \ldots, t_k\rangle, A, M)$ (line 14).

Practically non-deterministic choices are made by systematically choosing the task networks with the least amount of non-decomposed tasks. In the case where several networks have the same number of tasks remaining to be decomposed, the task network containing the least number of actions is chosen.

The search procedure implemented in PFD is almost similar. The main difference is no longer to choose the first t_1 task in the task network but to choose the first task that does not have any predecessor task in the task network. In addition, each time case 2 applies, it is necessary to check the consistency of the ordering constraints of the task network in order to generate a-cyclic task networks. This check is performed before line 14 by calculating the transitive closure of the ordering constraints. The computation of the transitive closure is based on Warshall algorithm. The complexity is $O(n^3)$ where n is the number of tasks of the task network.

Algorithm 1: TFD(s_0, T, A, M)

```
1  Let $T = \langle t_1, \ldots, t_k\rangle$
2  **if** $k = 0$ **then** return the empty plan $\langle\rangle$
3  **if** $t_1$ *is primitive* **then**
4      $A' \leftarrow$ the set of relevant actions for $t_1$ and
         applicable in $s_0$
5      **if** $A' = \emptyset$ **then return** failure
6      non-deterministically choose an action $a \in A'$
7      $\pi \leftarrow TFD(\gamma(s_0, a), \langle t_2, \ldots, t_k\rangle, A, M)$
8      **if** $\pi = \mathbf{failure}$ **then** return **failure**
9      **else return** $a \oplus \pi$
10 **else if** $t_1$ *is a non-primitive task* **then**
11     $M' \leftarrow$ the set of relvant methods for $t_1$ and
         applicable in $s_0$
12     **if** $M' = \emptyset$ **then return** failure
13     non-deterministically choose a decomposition
         $m \in M'$
14     **return** TFD($s_0$, *subtasks*($d$)$\oplus \langle t_2, \ldots, t_k\rangle, A, M$)
```

References

[Behnke et al. 2020] Behnke, G.; Höller, D.; Schmid, A.; Bercher, P.; and Biundo, S. 2020. On succinct groundings of HTN planning problems. In *The AAAI Conference on Artificial Intelligence*, 9775–9784. AAAI Press.

[Ghallab et al. 1998] Ghallab, M.; Howe, A.; Knoblock, G.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. *PDDL: The Planning Domain Definition Language*. Artificial Intelligence Planning Systems.

[Helmert 2006] Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

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

[Höller et al. 2020] 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 *The AAAI Conference on Artificial Intelligence*, 9883–9891. AAAI Press.

[Koehler and Hoffmann 1999] Koehler, J., and Hoffmann, J. 1999. Handling of inertia in a planning system. Technical report.

[Nau et al. 2003] Nau, D. S.; Au, T.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: an HTN planning system. *J. Artif. Intell. Res.* 20:379–404.

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

[Ramoul et al. 2017] Ramoul, A.; Pellier, D.; Fiorino, H.; and Pesty, S. 2017. Grounding of HTN planning domain. *Int. J. Artif. Intell. Tools* 26(5):1760021:1–1760021:24.

[Schreiber et al. 2019] Schreiber, D.; Pellier, D.; Fiorino, H.; and Balyo, T. 2019. Tree-rex: SAT-based tree exploration for efficient and high-quality HTN planning. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2018, Berkeley, CA, USA, July 11-15, 2019*, 382–390.

AssemblyHierarchical – Connecting Devices through Cables

Gregor Behnke
University of Freiburg
Freiburg im Breisgau, Germany
behnkeg@informatik.uni-freiburg.de

Abstract

We report on the AssemblyHierarchical domain, which encodes the task of connecting devices through a set of availables cables.

Introduction

We report on the idea and structure behind the domain AssemblyHierarchical, which was part of the benchmark set of the total order track of the IPC 2020. The AssemblyHierarchical domain was inspired by the Assembly Assistant (Bercher et al. 2014; 2015; 2017; 2018) developed in the Transregional Collaborative Research Centre SFB/TRR 62 "Companion-Technology for Cognitive Technical Systems" funded by the German Research Foundation (DFG). The assistant was designed to help a (usually novice) user to set up his or her home entertainment system. In this setting, there are multiple signal sources (satellite receiver, VCR, DVD player, . . .) as well as multiple signal sinks (TV, speaker, . . .). These devices need to be connected using only the available cables and intermediate devices (e.g. amplifiers). Adapters might have to be used to connect a given cable with a device. The assistant determines – based on the available cables and devices – how the cables should be used and instructs its user to plug them into the appropriate devices.

The assistance system implemented within the SFB/TRR 62 used planning to determine how to connect cables to devices. The AssemblyHierarchical domain is more general than the original one used by the assistant in the sense that it describes any arbitrary flow of an opaque signal from one device to another – this can be a video, audio, network, USB, or any other type of signal. The original domain was rather restrictive in the allowed operations and plans. The difficulty in modelling is due to the duality of the problem in this setting – plugging in cables *and* transmitting a signal – which can only be easily modelled using e.g. derived predicates, which in turn are computationally expensive. The problem stems from situations as follows. Consider four devices A, B, C, and D and cables connecting A and B, B and C, and C and D. To transmit a signal from A to D we can plug in these cables in any order. This signal is at any time only transmitted as far as the outgoing connection of A is plugged in. Notably, it is possible to plug in the cable from A to B last which will instantaneously cause the signal to be available at D. As noted before, this can be modelled with derived predicates or additional actions for signal transmission. The latter becomes much more complicated if we also allow for cables to be unplugged since signals also have to be "un-propagates". The AssemblyHierarchical domain we present in this paper actually supports unplugging cables.

Hierarchical planning is rather well suited to model this task. We can use a generator-style recursion of the HTN to allow for cables to be plugged in. The last decomposition of this generator will result in an abstract task that checks whether the connection has actually been established. This can be done solely via method preconditions.

Domain

As all other domains in the IPC, the AssemblyHierarchical domain is formulated in HDDL (Höller et al. 2020). We distinguish three types of `AbstractDevices`: `Devices` (representing larger devices), `Cables`, and `Adapters`. Each `AbstractDevice` has a number of `Ports`, which have a `PlugFace` (either male or female) and a `PlugDirection` (in, out, or both). A `Port` describes any point of an `AbstractDevice` that can be connected to anoter `AbstractDevice`. For example a `Port` of a TV might be connectable to the end of a cable – which is also a `Port`. A male port can only be connected to a female port. Further, an in-port can only be connected to an out- or both-type port and an out-port only to an in- or both-type port. Each `Port` further has a `PlugType` and can only be connected to ports of the same `PlugType`.

To connect ports, the AssemblyHierarchical domain has `connect` actions. Since the IPC 2020 does not allow for arbitrary formulae in preconditions, there are in total eight versions of the `connect` actions (named `connect_1` to `connect_8`). Given two ports `?p1` and `?p2` they represent the different possible configuration regarding the `PlugFace` and `PlugDirection` of the two ports. The domain contains an abstract task `connect` that can be decomposed into any of the eight concreate connect actions. The domain further contains a `disconnect` action that disestablishes an existing port connection.

The remainder of the AssemblyHierarchical domain features just three further abstract tasks: `ConnectDevices`, `ValidateDeviceConnection`, and `ValidatePortConnection`. `ConnectDevices` ensures that a signal can be transmitted between the two `AbstractDevices` `?d1` and `?d2` which are its parameters. In order to do so, it first allows – via methods – to generate an arbitrarily long sequence of `connect` and `disconnect` tasks. This recursion is ended with a method adding the action `guard` to the plan (for an explanation of this action see below). The method ending the recursion also inserts an instance of the abstract task `ValidateDeviceConnection` with the two arguments `?d1` and `?d2`. This task starts the validation of the (hopefully) established connection between `?d1` and `?d2`. To this end, it has to select an (outgoing-)port `?p1` of `?d1` and an (ingoing-)port `?p2` of `?d2`. These two ports are then passed on as arguments to the only subtask: `ValidatePortConnection`. This task validates via a recursive decomposition that there is infact a path from `?p1` to `?p3` via properly connected cables. If so, the last decomposition method simpy adds the action `ok`, which makes the goal fact `pAim` true.

Since the AssemblyHierarchical domain was only used in the total order track and all methods in the domain are totally ordered, the validation of the connection always happens after all cables have been plugged in. This would however not be the case if the initial plan was to contain multiple `ConnectDevices` actions – as disconnect actions of the second, if ordered after the first in the initial plan – might invalidate the connection established by the first. If multiple devices shall be connected, the respective `ConnectDevices` tasks must therefore be partially ordered in the initial plan. This however also not gurantees that connections are only validated after the last `connect` or `disconect`. For this purpose, the `guard` action adds the fact `pGuard` to the state. All `connect` and `disconect` actions have `(not (pGuard))` as their preconditions and the `ValidateDeviceConnection` task always occurs strictly after the `guard` action. This way, validation always happens after all `connect` and `disconnect` actions.

Instances

In the IPC 2020, only totally ordered instances of the AssemblyHierarchical domain were included in the benchmark set. As such, each instance contained only a single task in its inital plan, namely one `ConnectDevices` task for two specific devices. Each instance is solely described by a natural number i. Each instance has only two true devices called `pc` and `printer`. The instance numer i contains i additional cables, called `cableWithPlugTypeX` where `X` $\in \{1, \dots i\}$. Each cable has two bi-directional ports. One port of cable one is male and fits into the `pc`'s sole port while one port of the last cable is also male and fits into the `printer`'s sole port. Apart from that, cable i always has a port with which it can be connected to cable i, i.e. every cable has exactly two ports. There are also i `PlugTypes` named `plugTypeX` for `X` $\in \{1, \dots i\}$. Both of the ports of cable i have plug type i. The `pc`'s sole port always has plug type 1 and the `printer` always has plug type i. For each $j \in \{1, \dots, i-1\}$ there is an adapter called `adapterFromPlugTypeXToPlugTypeY` (with `X` $= j$ and `Y` $= j+1$) which as two ports: one of type j and one of type $j+1$.

With this setup there is only a single possible way to use all cables and adapters to connect the `pc` to the `printer`. As such, each problem of the AssemblyHierarchical domain has only one "true" solution. The domain however introduces a factorial amount of symmetric solutions – as the order in which the cables and adapters are plugged into one another can be choosen freely.

The domain in the IPC 2020 contained in total 30 instances from $i = 1$ to $i = 30$.

Performance in the IPC

Even though the setup of the AssemblyHierarchical is very simple, all participating planners in the IPC 2020 struggled with solving more than a few instances. The best planner on this domain – Lilotane (Schreiber 2021b; 2021a) solved only five out of the 30 total instances.

References

Bercher, P.; Biundo, S.; Geier, T.; Hörnle, T.; Nothdurft, F.; Richter, F.; and Schattenberg, B. 2014. Plan, repair, execute, explain – How planning helps to assemble your home theater. In *Proc. of the 24th Int. Conf. on Autom. Plan. and Sched. (ICAPS 2014)*, 386–394. AAAI Press.

Bercher, P.; Richter, F.; Hörnle, T.; Geier, T.; Höller, D.; Behnke, G.; Nothdurft, F.; Honold, F.; Minker, W.; Weber, M.; and Biundo, S. 2015. A planning-based assistance system for setting up a home theater. In *Proc. of the 29th AAAI Conf. on AI (AAAI 2015)*, 4264–4265. AAAI Press.

Bercher, P.; Richter, F.; Hörnle, T.; Geier, T.; Höller, D.; Behnke, G.; Nielsen, F.; Honold, F.; Schüssel, F.; Reuter, S.; Minker, W.; Weber, M.; Dietmayer, K.; and Biundo, S. 2017. *Advanced User Assistance for Setting Up a Home Theater.* Cognitive Technologies. Springer. chapter 24, 485–491.

Bercher, P.; Richter, F.; Honold, F.; Nielsen, F.; Schüssel, F.; Geier, T.; Hörnle, T.; Reuter, S.; Höller, D.; Behnke, G.; Dietmayer, K.; Minker, W.; Weber, M.; and Biundo, S. 2018. A companion-system architecture for realizing individualized and situation-adaptive user assistance. technical report, Ulm University.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. HDDL – A language to describe hierarchical planning problems. In *Proc. of the 34th AAAI Conf. on AI (AAAI 2020)*, 9883–9891. AAAI Press.

Schreiber, D. 2021a. 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).*

Schreiber, D. 2021b. Lilotane: A lifted SAT-based approach to hierarchical planning. *Journal of Artificial Intelligence Research* 70:1117–1181.

From Classical to Hierarchical: benchmarks for the HTN Track of the International Planning Competition

Damien Pellier, Humbert Fiorino
Univ. Grenoble Alpes - LIG
F-38000 Grenoble, France
{Damien.Pellier, Humbert.Fiorino}@imag.fr

Introduction

In this short paper, we outline nine classical benchmarks submitted to the first hierarchical planning track of the International Planning competition in 2020. All of these benchmarks are based on the HDDL language (Höller et al. 2020). The choice of the benchmarks was based on a questionnaire sent to the HTN community (Behnke et al. 2019). They are the following: Barman, Childsnack, Rover, Satellite, Blocksworld, Depots, Gripper, and Hiking. In the rest of the paper we give a short description of these benchmarks. All are totally ordered. A first hierarchical version of the domains Barman, Childsnack, Rover, Satellite, Blocksworld were proposed by (Ramoul et al. 2017; Schreiber et al. 2019). All the domains presented here are available online as part of the PDDL4J library (Pellier and Fiorino 2018). The writing of the domains has been a collective work. We would like to thank all the other contributors, D. Ramoul, D. Schreiber and A. Lequen.

Barman

In this domain, a barman robot manipulates drink dispensers, shots and a shaker. The goal is to find a plan that serves a targeted set of drinks. Action negative effects encode relevant knowledge given that robot hands can only grasp one object at a time and given that glasses need to be empty and clean to be filled. This domain was first proposed by S. Jiménez for STRIPS during IPC 2014.

Our domain is composed of 12 actions, 5 methods and 5 tasks. The actions are the same as in the STRIPS IPC domain. Each task has its own decomposition method. The domain has 2 high-level tasks. The first one describes how to serve a shot and the second one how to serve a cocktail. Serving a shot breaks down into 3 atomic sub-tasks: (1) grasp a shot; (2) fill the shot and (3) leave the shot on the table. Serving a cocktail is divided into 4 sub-tasks: (1) grasp a container; (2) get the first ingredient of the cocktail; (3) get the second ingredient and (4) shake the cocktail. The 3 last sub-tasks are not atomic. Getting an ingredient consists in 3 atomic sub-tasks: (1) fill a shot, (2) pour the shot to use a shaker and (3) clean the shot used. Finally, the last task breaks down into 6 atomic sub-tasks: (1) grasp the shaker; (2) shake the shaker; (3) pour the shaker into a shot ; (4) empty the shaker and (5) clean the shaker and (6) finally leave the shaker on the table. Note that this domain is not recursive.

Childsnack

Childsnack domain is for planning how to make and serve sandwiches for a group of children in which some are allergic to gluten. There are two actions for making sandwiches from their ingredients. The first one makes a sandwich and the second one makes a sandwich taking into account that all ingredients are gluten-free. There are also actions to put a sandwich on a tray, to move a tray from one place to another and to serve sandwiches. Problems in this domain define the ingredients to make sandwiches at the initial state. Goals consist of having all kids served with a sandwich to which they are not allergic. This domain was proposed by R. Fuentetaja and T. de la Rosa for STRIPS during IPC 2014.

Our domain is composed of 6 actions, 2 methods and 1 task. The actions are the same as in the STRIPS IPC domain. The high level task of the domain consists in serving sandwiches for a group of children. There is two methods to do it. The first one for the children who are gluten intolerant and the others. Serving a sandwich to an intolerant (resp. tolerant) child breaks down 5 atomic sub-tasks: (1) make a sandwich with no gluten (resp. with gluten); (2) put the sandwich on the tray, (3) move the tray from the kitchen to the child's place; (4) serve the sandwich to the child and finally (5) move back the tray to the kitchen. This domain is not recursive.

Rover

Inspired by planetary rover problems, this domain requires that a collection of rovers navigate a planet surface, finding samples and communicating them back to a lander.

Our domain is composed of 11 actions, 13 methods and 9 tasks. The 3 high-level tasks of the domain consist in getting soil and rock samples or images in a specific location. Getting soil samples breaks down in 4 sub-tasks : (1) navigate to the location to get the data; (2) empty the store of the rover; (3) take a soil sample and (4) send the soil data to the lander. The navigate task is a compound recursive task that consists in exploring all the possible paths by remembering location already explored. Finally, getting an image is a compound tasks that is divided in several sub-tasks including a task of camera calibration, image capture and image transmission to the lander.

Satellite

Inspired by space-applications, the original domain was a first step towards the ”ambitious spacecraft” described by David Smith at AIPS’00. It involves planning and scheduling a collection of observation tasks between multiple satellites, each equipped in slightly different ways.

Our domain is composed of 5 actions, 8 methods and 3 tasks. The actions are the same as in the STRIPS IPC domain. The domain has one high-level task that consists in observing stars. This task can be divided in 3 sub-tasks: (1) activate the instrument to carry out the observation ; (2) point towards the star to observe and (3) take the image of the star. Activating an instrument is a compound task. The activation procedure depends on the instrument to be activated and sometimes requires an instrument-specific calibration step. This domain does not have any recursive method.

Blocksworld

Probably the most known planning domain, in blocksworld stackable blocks need to be re-assembled on a table with unlimited space. A robot arm is used for stacking a block onto another block, unstacking a block from another block, putting down a block, or picking up a block from the table. The initial state specifies a complete world state, and the goal state only specifies the stacking relations required between any two blocks.

Our domain is composed of 5 actions, 8 methods and 4 tasks. The actions are the same as in the STRIPS IPC domain. Just one ”nop” action has been added to indicate the end of block stacking or unstacking. The high-level tasks of the domain consist in specifying the desired stacking of the blocks. Each stack requires either taking a block from the table or from a stack of blocks. In the latter case, either the block is at the top of the stack and the block can be taken directly, or it is necessary to recursively unstack all the blocks that are stacked on top of it before taking it.

Depots

This domain was devised to see what would happen if two previously well-known domains were joined together. These were the logistics and blocks domains. They are combined to form a domain in which trucks can transport crates around, and then the crates must be stacked onto pallets at their destinations. The stacking is achieved using hoists, so the stacking problem is like a blocks-world problem. Trucks can behave like ”tables” since the pallets on which crates are stacked are unlimited.

Our domain is composed of 6 actions, 12 methods and 6 tasks. The actions are the same as in the STRIPS IPC domain. Just one ”nop” action has been added to indicate the end of crate stacking or unstacking as in the blocksworld domain. The domain has recursive methods. The high-level method consists in defining the desired final position of the crates. As in the blocksworld domain there are recursively defined methods for stacking and unstacking, and methods defining how to move a crate from one location to another.

Gripper

In this domain, there is a robot with two grippers. It can carry a ball in each. The goal is to take N balls from one room to another; N rises with problem number. Some planners treat the two grippers asymmetrically, giving rise to an unnecessary combinatorial explosion. The first STRIPS version was proposed by J. Koehler for IPC 1998.

Our domain is composed of 3 actions, 4 methods and 3 tasks. The actions are the same as in the STRIPS IPC domain. The high level tasks specify the desired location of the balls. There are 2 methods to move a ball from one room to another. Either the robot moves just one ball or it uses its two arms to move two balls at the same time. This domain has no recursive methods.

Hiking

Suppose you want to walk with your partner a long clockwise circular route over several days (e.g., in the Lake District in NW England), and you do one “leg” each day. You want to start at a certain point and do the walk in one direction, without ever walking backwards. You have two cars that you must use to carry your tent/luggage and to carry you and your partner to the start/end of a leg, if necessary. Driving a car between any two points is allowed, but walking must be done with your partner and must start from the place where you left off. As you will be tired when you have walked to the end of a leg, you must have your tent up ready there so you can sleep the night before you set off to do the next leg in the morning.

Our hiking domain is composed of 8 actions, 15 methods and 8 tasks. The higt-level task consists in making hiking everyone at a specific location. This tasks breaks down into 2 sub-tasks: (1) prepare the trip and (2) make the trip. Preparing the trip consists in (1) bringing the tent and (2) bring the car. These two tasks are also broken down into sub-tasks depending on the position of the tent or the car. Finally, making the trip breaks down in several sub-tasks depending on the means of transport used (on foot or by car). This domains has no recursive methods.

Conclusion

This work is a first step towards the development of a set of benchmarks for the evaluation of hierarchical planners. There are still many STRIPS domains that can be transposed for hierarchical planning. Some are very easy to transpose. The methods are easy to write down. For other domains, on the contrary, it is difficult to identify relevant methods. In our opinion, the transposition effort must be continued in order to better understand for which type of fields a hierarchical representation is more appropriate.

References

[Behnke et al. 2019] Behnke, G.; Höller, D.; Bercher, P.; Biundo, S.; Pellier, D.; Fiorino, H.; and Alford, R. 2019. Hierarchical planning in the IPC. In *Proceedings of the Workshop on the International Planning Competition (ICAPS).*

[Höller et al. 2020] 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 *The AAAI Conference on Artificial Intelligence*, 9883–9891. AAAI Press.

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

[Ramoul et al. 2017] Ramoul, A.; Pellier, D.; Fiorino, H.; and Pesty, S. 2017. Grounding of HTN planning domain. *Int. J. Artif. Intell. Tools* 26(5):1760021:1–1760021:24.

[Schreiber et al. 2019] Schreiber, D.; Pellier, D.; Fiorino, H.; and Balyo, T. 2019. Tree-rex: SAT-based tree exploration for efficient and high-quality HTN planning. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2018, Berkeley, CA, USA, July 11-15, 2019*, 382–390.

From PCP to HTN Planning Through CFGs

Daniel Höller[1] and **Songtuan Lin**[3] and **Kutluhan Erol**[2] and **Pascal Bercher**[3]
[1] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[2] İzmir University of Economics, Turkey
[3] The Australian National University, Australia
hoeller@cs.uni-saarland.de, kutluhan.erol@ieu.edu.tr, {songtuan.lin, pascal.bercher}@anu.edu.au

Abstract

The International Planning Competition in 2020 was the first one for a long time to host tracks on HTN planning. The used benchmark set included a domain describing the undecidable *Post Correspondence Problem (PCP)*. In this paper we describe the two-step process applied to generate HTN problems based on PCP instances. It translates the PCP into a grammar intersection problem of two context-free languages, which is then encoded into an HTN problem.

Introduction

Hierarchical Task Network (HTN) planning is a widely-used planning approach with many practical applications (Bercher, Alford, and Höller 2019). It provides two means of modeling: a grammar-like decomposition structure as well as actions with preconditions and effects. The hierarchy makes it more expressive than e.g. classical planning, where only the latter is available. Erol, Hendler, and Nau (1996) showed that it enables the description of recursively enumerable, undecidable problems like the intersection problem of context-free languages. The International Planning Competition (IPC) in 2020 was the first for a long time to host tracks on HTN planning. To include provable hard domains, we included one describing the *Post Correspondence Problem (PCP)*, which is one of the standard decisions problems known to be undecidable (Hopcroft and Ullman 1979).

In this paper we give a short introduction of the applied encoding. We first translate PCP into the language intersection problem of the context-free languages. For the language intersection model, we can use the original encoding used to prove hardness of the plan existence problem in HTN planning (Erol, Hendler, and Nau 1996).

Background

We first shortly introduce PCP (for further details see e.g. (Hopcroft and Ullman 1979)). An instance of a PCP consists of two finite lists of finite strings (over an alphabet A with at least two symbols) with the same length: $P^1 = s^1_1, \dots, s^1_n$ and $P^2 = s^2_1, \dots, s^2_n$ ($n \in \mathbb{N}$). There is no restriction on the length of the individual s^1_i and s^2_i. A solution is a finite sequence of indices $j_1 \dots j_k$ ($k \in \mathbb{N}$) with $1 \leq j_r \leq n$ for each $1 \leq r \leq k$ such that the compound string $s^1_{j_1} \dots s^1_{j_k}$ is identical to $s^2_{j_1} \dots s^2_{j_k}$.

A context-free grammar is a tuple $G = (\Gamma, \Sigma, P, S)$, where Γ is a finite set of non-terminal symbols, Σ is a finite set of terminal symbols, P is a finite set of production rules mapping a single non-terminal symbol to a finite sequence of terminal and non-terminal symbols. S is the start symbol. With the language $L(G)$ of a grammar we refer to the set of words of that grammar, i.e., all terminal symbol sequences that can be obtained by refining S via adhering the rules in P. Erol, Hendler, and Nau (1994; 1996) were the first to recognize the close relationship to HTN models, which they exploited for HTN's undecidability proof. Höller et al. (2014; 2016) have taken this further and studied the close relationship between various hierarchical (and non-hierarchical) planning problems and the Chomsky Hierarchy.

From PCP to HTN Planning Problems

We first translate a given PCP instance into a grammar intersection problem. For each list of strings, we construct a grammar such that words derived from that grammar begin with newly introduced letters representing the selected string indices from the respective P^i, followed by the actual concatenation of these strings. This is done for both P^is, when these languages have an intersection, this means that there is a list of indices leading to the same overall string, which solves our PCP problem.

For each P^i from the PCP, we construct a CFG $G^i = (\{S^i\}, A \cup L, P^i, S^i)$, where $L = \{l_1, \dots, l_n\}$, n is the length of the string list and the production rules P^i are given by two rules $S^i \to l_j S^i s^i_j$ and $S^i \to l_j\, s^i_j$ for each s^i_j in the list, where j is the string's index, and l_j a terminal symbol.

Now that we have constructed the grammars G^1 and G^2 for P^1 and P^2, we check whether they may both produce the same string, relying on the encoding introduced by Erol, Hendler, and Nau (1996). The resulting problems include two tasks in the initial task network, which are not ordered with respect to each other. Each of them can be decomposed in sequences of actions representing the words of the language of one of the grammars. Preconditions and effects of the actions ensure that there is an applicable linearization if and only if the actions derived from the two grammars are applied in turns, and some letter from the second grammar follows the same letter from the first grammar. That way, the HTN problem has a solution if and only if the languages have a non-empty intersection.

We use the formalism by Geier and Bercher (2011). An HTN problem is a tuple $P = (F, N_p, N_c, M, \delta, tn_I, s_I, g)$. F is a set of propositional state features, N_p the set of primitive tasks, N_c the set of abstract (also *compound*) tasks, M the set of decomposition methods, δ a function mapping primitive tasks to their precondition and effects, tn_I the initial task network, s_I the initial state, and g the state-based goal condition.

Let $G^1 = (\{S^1\}, \Sigma, P^1, S^1)$ and $G^2 = (\{S^2\}, \Sigma, P^2, S^2)$ be the two grammars constructed in the previous step. The set of state features is defined as $F = \Sigma \cup \{turn_1, turn_2\}$, the *primitive tasks* as $N_p = \{p^i_a \,|\, a \in \Sigma, i \in \{1,2\}\}$. δ is defined as follows: if $i = 1$ then $\delta(p^i_a) = (\{turn_1\}, \{a, turn_2\}, \{turn_1\})$, otherwise, $\delta(p^i_a) = (\{turn_2, a\}, \{turn_1\}, \{turn_2, a\})$. The domain contains two compound tasks $N_c = \{S^1, S^2\}$. The set of methods M which decompose those two compound tasks is constructed according to the set of production rules $P^1 \cup P^2$. Let $p \in P^1 \cup P^2$ be an arbitrary production rule. If p is in the form $S^i \to lS^i s$ where $i \in \{1,2\}$, $l \in L$, and $s = c_1 \dots c_k$ is a string with $k \in \mathbb{N}$ and $c_j \in A$ for $1 \leq j \leq k$, we construct a method $m = (S^i, (T, \prec, \alpha))$ where

$$T = \{t_S, t_l, t_1, \dots, t_k\}$$
$$\prec = \{(t_l, t_S), (t_S, t_1), \dots, (t_{k-1}, t_k)\}$$
$$\alpha = \{(t_S, S^i), (t_l, p^i_l), (t_1, p^i_{c_1}), \dots, (t_k, p^i_{c_k})\}$$

For production rule not containing S^i, a similar method is constructed not including S^i. Initial task network, initial state, and state-based goal condition are defined as follows:

$$tn_I = (\{s^1, s^2\}, \emptyset, \{(s^1, S^1), (s^2, S^2)\})$$
$$s_I = \{turn_1\} \quad g = \{turn_1\}$$

Example

Fig. 1 shows the definitions of the tasks `t1G1` and `t1G2`, which are two actions that can be executed after each other, since one corresponds to the "creation" of the symbol *l1* by the first grammar, whereas the other, by the second grammar, deletes that symbol. The example code is provided in HDDL (Höller et al. 2020). The used predicate names in our figure differ only slightly from the respective names in the actual problem files (we adapted it slightly to match our formal definitions from before).

Benchmark Collection

For the IPC we selected problem instances where a solution exists, and we varied degree of difficulty as measured by the solution length. Meanwhile we also support the automatic generation of problem instances based on an external PCP problem generator. That generator creates random PCP instances given a minimal solution length, which is obtained by solving the respective problem. The random generator is available in the IPC benchmark repository next to the instances used in the IPC.

```
(:action t1G1                 (:action t1G2
  :parameters ()                :parameters ()
  :precondition                 :precondition
    (and                          (and
      (turn1)                       (turn2)
    )                               (l1))
  :effect                       :effect
    (and                          (and
      (not (turn1))                 (not (turn2))
      (turn2)                       (turn1)
      (l1)))                        (not (l1))))
```

Figure 1: Examples of primitive tasks.

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 (IJCAI)*, 6267–6275. IJCAI Organization.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. Semantics for hierarchical task-network planning. Technical Report CS-TR-3239, UMIACS-TR-94-31, ISR-TR-95-9, Inst. for Advanced Computer Studies, Inst. for Systems Research, Computer Science Department, University of Maryland.

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

Geier, T., and Bercher, P. 2011. On the decidability of HTN planning with task insertion. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*, 1955–1961. 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)*, 447–452. IOS Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2016. Assessing the expressivity of planning formalisms through the comparison to formal languages. In *Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS)*, 158–165. 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)*, 9883–9891. AAAI Press.

Hopcroft, J. E., and Ullman, J. D. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Hierarchical Task Networks Generated Using Invariant Graphs for IPC2020

Damir Lotinac, Filippos Kominis, Anders Jonsson[1]
[1] Universitat Pompeu Fabra, Roc Boronat 138, 08018 Barcelona, Spain.

Abstract

This paper describes HTN domains generated using PDDL description of a planning domain and a single representative instance. We also describe the algorithm used to generate the HTN domains. Two types of composite tasks that interact to achieve the goal of a planning instance are generated. One type of task achieves fluents by traversing invariants in which only one fluent can be true at a time. The other type of task applies a single action, which first involves ensuring that the precondition of the action holds. Finally we discuss differences between JSHOP2 and domains generated for IPC2020 in HDDL.

Introduction

Hierarchical Task Networks models enable defining compound parameterized tasks which can reduce the search complexity if the adequate constraints can be identified during the modeling. The hierarchical structure enables the modeler to encode domain-specific knowledge. The expressiveness of HTNs can help to impose the constraints, which can in turn lead to a reduction in the search complexity. The hierarchy ideally imposes some constraints on how tasks can be decomposed. The more constrained the task network, the less search has to be performed in order to achieve a certain task.

HTN models are more expressive than STRIPS (Erol, Hendler, and Nau 1994), which along with the ability to construct parametric compound tasks allows for capturing domain-specific knowledge.

In this paper we describe the HTN domains and instances which were submitted to IPC2020 [1] and the algorithm which was used to generate them. We used PDDL instances from IPC-2000 and IPC-2002 as input.

To generate the HTN domains we use HTNPrec algorithm. The algorithm takes as input the PDDL description of a planning domain and a single representative instance. The approach is to generate HTNs that encode invariant graphs of planning domains. An invariant graph is similar to a lifted domain transition graph, but can be subdivided on types. To traverse an invariant graph we define two types of tasks: one that reaches a certain node of an invariant graph, achieving the associated fluent, and one that traverses a single edge of an invariant graph, applying the associated action. These two types of tasks are interleaved, in that the expansion of one type of task involves tasks of the other type.

We also describe differences between HTN domains generated by HTNPrec, HTNGoal and domains generated for the IPC2020. While HTNPrec and HTNGoal (Lotinac and Jonsson 2016) use a JSHOP2 (Nau et al. 2003) representation, for the IPC the domains are given in the HDDL format (Höller et al. 2020). Further some of the optimizations are not included in the IPC version. The HTN instances generated using JSHOP2 are solved with blind search, thus those HTN domains are meant to guide the search through the underlying invariant graph structures. In contrast HDDL domains are generated with minimal additions to the original PDDL domain.

Hierarchical Task Networks

Our HTN definition is inspired by Geier and Bercher (2011). However, just as for STRIPS planning, we separate the definition into a domain part and an instance part. We also impose additional restrictions: a task network can contain at most one copy of each task, and task decomposition is limited to *progression*, always decomposing tasks with no predecessor.

An HTN domain is a tuple $\boldsymbol{h} = \langle P, A, C, M \rangle$ consisting of four sets of untyped function symbols. Specifically, P is the set of *predicates*, A is the set of *actions* (i.e. primitive tasks), C is the set of *compound tasks* and M is the set of *decomposition methods*. Predicates and actions are defined as for STRIPS domains but, unlike STRIPS domains, HTN domains are untyped and we allow negative preconditions.

Each method $m \in M$ has an associated tuple $\langle c, tn_m, \text{pre}(m) \rangle$ where $c \in C$ is a compound task with the same arity as m, tn_m is a *task network* and $\text{pre}(m)$ is a set of preconditions, defined as for actions. The task network $tn_m = (T, \prec)$ consists of a set T of pairs (t, φ), where $t \in A \cup C$ is a task and φ is an argument map from m to t, and a partial order $\prec$ on the tasks in T.

Given an HTN domain $\boldsymbol{h}$, an HTN instance is a tuple

[1]The domains and the code are available at: https://github.com/dloti/pddl-to-htn

s $= \langle \Omega, \text{init}, tn_I \rangle$, where Ω is a set of objects and init is an initial state. The instance ***s*** induces sets P_Ω and A_Ω of fluents and grounded actions, and sets C_Ω and M_Ω of grounded compound tasks and grounded methods, respectively. A grounded method $m[x] \in M_\Omega$ has associated tuple $\langle c[x], tn_m[x], \text{pre}(m[x]) \rangle$, where $c[x]$ is a grounded compound task and the precondition $\text{pre}(m[x])$ is derived as for grounded actions. The grounded task network $tn_m[x] = (T_x, \prec)$ is defined by $T_x = \{t[\varphi(x)] : (t, \varphi) \in T\}$. The initial grounded task network $tn_I = (\{t_I\}, \emptyset)$ contains a single grounded compound task $t_I \in C_\Omega$.

An HTN state (s, tn) consists of a state $s \subseteq P_\Omega$ on fluents and a grounded task network tn. We use $(s, tn) \rightarrow_D (s', tn')$ to denote that an HTN state decomposes into another HTN state, where $tn = \langle T_x, \prec \rangle$ and $tn' = \langle T_y, \prec' \rangle$. A valid *progression decomposition* consists in choosing a grounded task $t \in T_x$ such that $t' \not\prec t$ for each $t' \in T_x$, and applying one of the following rules:

1. If t is primitive, the decomposition is applicable if $\text{pre}(t) \subseteq s$, and the resulting HTN state is given by $s' = s \ltimes t$, $T_y = T_x \setminus \{t\}$ and $\prec' = \{(t_1, t_2) \in \prec \mid t_1, t_2 \in T_y\}$.
2. If t is compound, a grounded method $m = \langle t, tn, \text{pre}(m) \rangle$ with $tn = (T_m, \prec_m)$ is applicable if $\text{pre}(m) \subseteq s$, and the resulting HTN state is given by $s' = s$, $T_y = T_x \setminus \{t\} \cup T_m$ and

$$\begin{aligned} \prec' = \{&(t_1, t_2) \in \prec \mid t_1, t_2 \in T_y\} \\ \cup\ \{&(t', t_1) \in T_m \times T_y \mid (t, t_1) \in \prec\} \cup \prec_m . \end{aligned}$$

The first rule removes a grounded primitive task t from tn and applies the effects of t to the current state, while the second rule uses a grounded method m to replace a grounded compound task t with tn_m while leaving the state unchanged. If there is a finite sequence of decompositions from (s_1, tn_1) to (s_n, tn_n) we write $(s_1, tn_1) \rightarrow_D^* (s_n, tn_n)$. An HTN instance ***s*** is solvable if and only if $(\text{init}, tn_I) \rightarrow_D^* (s_n, \langle \emptyset, \emptyset \rangle)$ for some state s_n, i.e. the initial HTN state (init, tn_I) is decomposed into an empty task network. Let π be the sequence of grounded actions extracted during such a decomposition; π corresponds to a *plan* that results from solving ***s***.

Invariants

In STRIPS planning, an exactly-1 invariant is a subset of fluents $F' \subseteq P_\Omega$ such that exactly one fluent in F' is true at any moment. Formally, $|F' \cap \text{init}| = 1$ and any grounded action $a \in A_\Omega$ that adds a fluent in F' deletes another. The Fast Downward planning system (Helmert 2009) uses the domain description of a STRIPS domain to detect lifted invariant candidates. Unlike Fast Downward, which grounds lifted invariants on actual instances, our algorithm operates directly on the lifted invariants.

In LOGISTICS, Fast Downward finds a single lifted invariant candidate {(`in ?o ?v`),(`at ?o ?p`)}, i.e. a set of predicates with associated arguments. In the given invariant, variable `?o` is *bound* while variables `?v` and `?p` are *free*. To ground the lifted invariant on an instance ***p***, we should create one mutex invariant F' for each assignment of objects to the bound variables, obtaining each fluent in F' by assigning objects to the free variables. In our running example, assigning the package `p1` to `?o` results in the following grounded mutex invariant:

```
{(at p1 ap1),(at p1 ap2),(at p1 l1),(at p1 l2),
  (in p1 t1),(in p1 t2),(in p1 a1)}.
```

The meaning of the invariant is that across all LOGISTICS instances, a given object `?o` is either in a vehicle or at a location.

If a predicate $p \in P$ is not part of any invariant but there are actions that add and/or delete p, we create a new lifted invariant {(p `?o1` ⋯ `?ok`),($\neg p$ `?o1` ⋯ `?ok`)}. In this invariant, all variables `?o1`, …, `?ok` are bound and an associated fluent can either be true or false.

Given a lifted invariant, our algorithm generates one or several invariant graphs. We do so by iterating over the actions of the domain and identifying which actions add and delete predicates in the same lifted invariant. When grounded, such actions have the effect of *changing* the fluent of an exactly-1 invariant that is currently true. An invariant graph is a representation of a lifted invariant in which the nodes are the predicates of the invariant and the edges are the actions used to change the predicate that is currently true. We use invariant graphs to infer which actions to perform in order to achieve a particular fluent of an exactly-1 invariant.

The reason why a given lifted invariant can generate multiple invariant graphs is that the *type* of the bound objects may be different for different actions. For example, in the LOGISTICS domain, all actions affect the lone invariant above. However, in the actions for loading or unloading a package, the bound object `?o` is a package, in the action for driving a truck `?o` is a truck, and in the action for flying an airplane `?o` is an airplane. Moreover, we can either load a package into a truck or an airplane. We use the actions to differentiate between types, possibly generating multiple invariant graphs for each lifted invariant.

To generate the invariant graphs induced by lifted invariants we go through each action, find each transition of each invariant that it induces (by pairing add and delete effects and testing whether the bound objects are identical), and map the types of the predicates to the invariant. We then either create a new invariant graph for the bound types or add nodes to an existing graph corresponding to the mapped predicate arguments.

Figure 1 shows the invariant graphs that we generate in LOGISTICS. In the top graph (G_1), the bound object is a package `?p`, in the middle graph (G_2) it is a truck `?t`, and in the bottom graph (G_3) it is an airplane `?a`. Note that the predicate `in` is not actually part of the two bottom graphs, since trucks and planes cannot be inside other vehicles. Nevertheless, the invariant still applies: a truck or plane can only be at a single place at once.

Each edge of an invariant graph corresponds to an action that deletes one predicate of the invariant and adds another. To do so, the arguments of the action have to include the arguments of both predicates, including the bound objects. In the figure, the invariant notation is extended to actions on

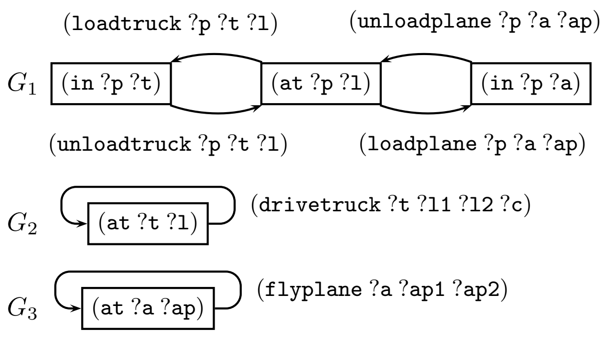

Figure 1: Invariant graphs G_1, G_2 and G_3 in LOGISTICS.

edges such that each argument of an action is either bound or free.

Even if actions preserve the invariant property, the initial state of a planning instance may violate the condition $|F' \cap \text{init}| = 1$, in which case F' is not an exactly-1 invariant. To verify that a lifted invariant candidate corresponds to actual exactly-1 invariants, our algorithm needs access to the initial state of an example planning instance ***p*** of the domain. If this verification fails, the lifted invariant is not considered by the algorithm.

Generating HTNs

In this section we describe the algorithm for generating the HTN domains. The idea is to construct a hierarchy of tasks that traverse the invariant graphs to achieve certain fluents. In doing so there are two types of interleaved tasks: one that achieves a fluent in a given invariant (which involves applying a series of actions to traverse the edges of the graph), and one that applies the action on a given edge (which involves achieving the preconditions of the action).

A planning domain is a tuple $\boldsymbol{d} = \langle \mathcal{T}, <, P, A \rangle$, where $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ is a set of types, $<$ is an inheritance relation on types, P is a set of typed function symbols called *predicates*, and A is a set of typed function symbols called *actions*. Each action $a \in A$ has a set of preconditions $\text{pre}(a)$, a set of add effects $\text{add}(a)$ and a set of delete effects $\text{del}(a)$. Each element in these three sets is a pair (p, φ) consisting of a predicate $p \in P$ and an argument map φ from a to p.

Given $\boldsymbol{d}$, a planning instance is a tuple $\boldsymbol{p} = \langle \Omega, \text{init}, \text{goal} \rangle$, where $\Omega = \Omega_1 \cup \ldots \cup \Omega_n$ is a set of objects of each type.

Formally, our algorithm takes as input a STRIPS planning domain $\boldsymbol{d} = \langle \mathcal{T}, <, P, A \rangle$ and a planning instance $\boldsymbol{p} = \langle \Omega, \text{init}, \text{goal} \rangle$ and outputs an HTN domain $\boldsymbol{h} = \langle P, A', C, M \rangle$. The HTN domain ***h*** can then be used to solve any other instance of the domain. Specifically, for each instance $\boldsymbol{p}'$ of the planning domain $\boldsymbol{d}$, we construct an HTN instance ***s***. Solving the HTN induced by ***d*** and ***s*** returns a plan that can be adapted to solve $\boldsymbol{p}'$.

The input planning instance ***p*** is used for three purposes:

1. To verify that an invariant candidate is actually an invariant by testing the condition $|F' \cap \text{init}| = 1$.
2. To extract a subset of predicates $P_G \subseteq P$ that are part of the goal.
3. To perform goal ordering as described in a subsequent section.

The algorithm first constructs the invariant graphs $G_1, \ldots, G_k$ described above. In what follows we describe the components of the HTN domain ***h***.

The set A' contains the following actions:

- Each action $a \in A$. For each element $\beta_k(a) \in \mathcal{T}$ of the type list of a, we add an additional precondition $(\beta_k(a), \varphi_k)$. where the argument map φ_k maps the argument x_k of a to the lone argument of the type predicate $\beta_k(a)$, ensuring that argument x_k has the correct type.

Note that only actions in A add or delete predicates in the original set P. The set C contains three types of compound tasks:

- For each predicate $p \in P$, a task achieve-p with arity $\alpha(p)$.
- For each invariant graph G_i and each $p \in P$ that is positive in G_i, a task achieve-p-i with arity $\alpha(p)$.
- For each invariant graph G_i, each predicate p in G_i, and each outgoing edge of p (corresponding to an action $a \in A$), a task do-p-a-i with arity $\alpha(a)$.

The task achieve-p is a wrapper task that uses a task achieve-p-i to achieve p by traversing the edges of the invariant graph G_i. To traverse each edge of G_i, achieve-p-i has to use a task of type do-p-a-i, which in turn uses tasks of type achieve-p' to achieve the preconditions of a.

Methods

The set M contains the following decomposition methods. For simplicity, we use x to denote an argument list, and define argument maps inline which are described in the text. We describe methods in pseudo-SHOP2 syntax in the following format:

```
(:method (⟨name⟩[⟨arguments⟩])
 (⟨precondition⟩)
 (⟨tasklist⟩))
```

For each method in the first line we specify a name and arguments, in the second line we give a precondition list, and finally in the third we specify the respective task list to which method decomposes. For clarity, we add an exclamation mark in front of primitive tasks.

- Methods for achieve-p

  The first type of compound task, achieve-p, has one associated method for each invariant graph G_i in which p appears. This method is defined as follows:

  ```
  (:method (achieve-$p[x]$)
    ($\neg p[x]$)
    (achieve-$p$-$i[x]$)).
  ```

  Intuitively this method delegates achieving p to the task achieve-p-i for some invariant graph G_i. The precondition $\neg p[x]$ ensures that p is not currently true.

In addition, there is one method with empty task list which is applicable when p already holds:

$(:\texttt{method}\ (\mathsf{achieve}\text{-}p[x])$
$(p[x])$
$()).$

- Methods for $\mathsf{achieve}\text{-}p\text{-}i$

The second type of compound task, $\mathsf{achieve}\text{-}p\text{-}i$, has one associated method for each predicate q in the invariant graph G_i and outgoing edge of q (corresponding to an operator o):

$(:\texttt{method}\ (\mathsf{achieve}\text{-}p\text{-}i[x])$
$(\neg p[x], q[\varphi_q(x)])$
$(\mathsf{do}\text{-}q\text{-}o\text{-}i[\varphi_o(x)], \mathsf{achieve}\text{-}p\text{-}i[x])).$

Operator o appears on an outgoing edge from q, i.e. o deletes q. Intuitively, one way to achieve p in G_i, given that we are currently at some different node q, is to traverse the edge associated with o using the compound task $\mathsf{do}\text{-}q\text{-}o\text{-}i$. After traversing the edge we recursively achieve p from the resulting node. The argument map φ_o should map the bound objects of p to o while leaving the remaining arguments of o as free variables. The argument map φ_q maps the bound objects of p to q, and shares all free variables with φ_o (since q is a delete effect of o).

We also define a decomposition method for $\mathsf{achieve}\text{-}p\text{-}i$ which is applicable when p already holds and has empty task list:

$(:\texttt{method}\ (\mathsf{achieve}\text{-}p\text{-}i[x])$
$(p[x])$
$())$

- Method for $\mathsf{do}\text{-}p\text{-}o\text{-}i$

The third type of compound task, $\mathsf{do}\text{-}p\text{-}o\text{-}i$, has a single associated method. The aim is to apply operator o to traverse an outgoing edge of p in the invariant graph G_i. To do so, the task list has to ensure that all preconditions $p_1, \dots, p_k$ of o hold (excluding p, which has to hold to apply the method, as well as any static preconditions of o). We define the method as

$(:\texttt{method}\ (\mathsf{do}\text{-}p\text{-}o\text{-}i[x])$
$(p[\varphi_p(x)])$
$(\mathsf{achieve}\text{-}p_1[\varphi_1(x)], \dots, \mathsf{achieve}\text{-}p_k[\varphi_k(x)], !o[x]))$

Here, the argument map φ_j, $1 \leq j \leq k$, maps the arguments of operator o to the precondition p_j of o. This mapping is given directly by the definition of operator o. Note that the decomposition achieves all preconditions of o except p, then applies o.

When p is the only precondition of operator o, task $\mathsf{do}\text{-}p\text{-}o\text{-}i[x]$ is not needed since operator o is always applicable as long as p holds. In this case, whenever $\mathsf{do}\text{-}p\text{-}o\text{-}i[x]$ appears in a decomposition method of a task $\mathsf{achieve}\text{-}q\text{-}j$, we replace $\mathsf{do}\text{-}p\text{-}o\text{-}i[x]$ directly with the operator $!o[x]$.

Planning Instances

Once we have generated the HTN domain $\boldsymbol{h}$ we can apply it to any instance of the domain. Given a STRIPS instance $\boldsymbol{p} = \langle \Omega, \text{init}, \text{goal} \rangle$, we construct an HTN instance $\boldsymbol{s} = \langle \Omega, \text{init}', \langle \mathsf{achieve}\text{-}p_1[x_1], \dots, \mathsf{achieve}\text{-}p_k[x_k] \rangle \rangle$, given goal $= \{p_1[x_1], ..., p_k[x_k]\}$, as follows. The set of objects $\Omega = \Omega_1 \cup \cdots \cup \Omega_n$ is identical to that of $\boldsymbol{p}$. The initial state init$'$ is defined as $\text{init}' = \text{init} \cup \{\tau_j[\omega] : \tau_j \in \mathcal{T}, \omega \in \Omega_j\} \cup \{\texttt{goal}\text{-}p[x] : p[x] \in \text{goal}\}$. We thus mark the type τ_j of each object ω using the fluent $\tau_j[\omega]$, and we mark all fluents $p[x]$ in the goal state using the fluent $\texttt{goal}\text{-}p[x]$. The initial task network contains the achieve tasks which correspond to each fluent $p[x]$ in the goal state. The ordering of achieve tasks is imposed based on the order of goal fluents in the given PDDL instance.

Optimizations

Achieving the preconditions of an action a in any order is inefficient since an algorithm solving the HTN instance may have to backtrack repeatedly. For this reason, we include the HTNPrec algorithm that uses a simple inference technique to compute a partial order in which to achieve the preconditions of a. We define a set of predicates whose value is supposed to persist, and check whether a path through an invariant graph is applicable given these persisting predicates. While doing so, only the values of bound variables are known, while free variables can take on any value. We match the bound variables of predicates and actions to determine whether an action allows a predicate to persist.

Discussion

There are several differences between the HDDL domains and instances and JSHOP2 version generated by HTNPrec and HTNGoal. The HDDL version is closest to HTNPrec, since it does not apply goal order optimization. However, there are several differences to the HTNPrec algorithm as well. In this section we give tasks, methods and predicates which are not generated by the IPC2020 version of the algorithm.

Predicates which are not generated:

- $\texttt{visited}\text{-}p$, indicating that p has already been visited during search.
- $\texttt{achieving}\text{-}p$, indicating that p or another predicate in the same invariant are already being achieved.
- $\texttt{goal}\text{-}p$, indicating that a fluent derived from p is a goal state.

Actions which are not generated:

- $\texttt{occupy}\text{-}i$, which marks each predicate in the invariant graph G_i as being achieved.
- $\texttt{clear}\text{-}i$, which deletes $\texttt{visited}\text{-}p$ and $\texttt{achieving}\text{-}p$ for each predicate p of the invariant graph G_i.
- $\texttt{test}\text{-}p$ with arity 0 and no effects, whose precondition tests if *all* goal fluents derived from p hold.

The only task left out is solve whose decomposition achieves the goal condition JSHOP2 version. In HDDL the algorithm simply creates the task list by adding the achieve tasks in the order of appearance in the PDDL instance. This is under the assumption that the original instance can be solved under such restriction.

The predicates and actions mentioned above are used to guide the search over the HTN decompositions. As such these predicates are added having blind search in mind and while they should not hinder the perofmance of a heuristic planner, they can also be left out. HTNPrec and HTNGoal also have a different structure given that the resulting HTN consists of only one task to decompose. In contrast, for the IPC2020 we generated instances with task list consisting of achieve tasks.

References

Erol, K.; Hendler, J.; and Nau, D. 1994. HTN planning: Complexity and expressivity. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'94)*, 1123–1128.

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 (IJ-CAI'11)*, 1955–1961.

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

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. Hddl: An extension to pddl for expressing hierarchical planning problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, 9883–9891.

Lotinac, D., and Jonsson, A. 2016. Constructing hierarchical task models using invariance analysis. In *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, 1274–1282.

Nau, D.; Au, T.; Ilghami, O.; Kuter, U.; Murdock, W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research* 20:379–404.

HTN IPC-2020 Domains: Blocksworld-HPDDL and Multiarm-Blocksworld

Ron Alford
The MITRE Corporation
ralford@mitre.org

Abstract

The Blocksworld-HDDL problems from the 2020 HTN International Planning Competition were adapted from the HTN translation papers by Alford et al.. The domain enforces a strategy of only picking up blocks to move, and placing them either in their final location or on the table. The Multiarm-Blocksworld domain extends the domain by allowing for multiple independent arms.

The Blocksworld-HPDDL domain was introduced by the 2009 translation of totally-ordered HTN problems to PDDL (Alford, Kuter, and Nau 2009). Blocksworld-HPDDL was supplemented by the Multiarm-Blocksworld domain in further work translating partially-ordered HTNs to PDDL (Alford et al. 2016). These domains were designed to test the effectiveness heuristic search when using control knowledge to eliminate the Sussman Anomaly from the standard Blocksworld's search space.

The IPC problem files are generated with the uniform Blocksworld state generator by Slaney and Thiébaux (2001), and converted to the HDDL problem format. Each of the problem initial states is supplemented with predicates describing the goal (e.g., `(goal_on a b)` if `(on a b)` is part of the goal state). The HTN IPC contained 30 problems from this domain with a block count ranging from five to 1,000.

The domain has a top level task `(achieve-goals)`. At a high level, it loops through blocks which aren't marked as `done` (i.e., for a block `b`, `(done b)` is part of the state). For each block `b` where `(done b)` is not part of the state:

- If `b` is on the table and has a goal of being on the table, `(done b)` is added to the state.
- If `b` is on a block `c`, `(on b c)` is part of the goal state, and `c` is marked as `done`, then mark `b` as `done`.
- If `b` is clear, on another block, and its goal location is clear (or on the table), then pick up the block, place it in its goal location, and mark it `done`.
- If `b` is clear and its goal location is not, pick up the block and set it on the table.

Preconditions to the four `achieve-goals` methods prevent them from selecting a block which isn't ready for any of the four methods. For any solution to the original Blocksworld domain, this method structure permits a solution with the same or fewer total `pickup`, `putdown`, `stack`, and `unstack` actions by eliminating unnecessary moves and replacing some stacks with table placements.

In structure, the `achieve-goals` task is tail-recursive with a max progression bound of 4, strictly limiting the size of task networks that HTN-progression oriented planners will encounter during search (Alford, Bercher, and Aha 2015). There are 12 methods and 6 operators, four of which are the original actions of Blocksworld, and two of which are bookkeeping operators.

The Multiarm-Blocksworld extends the original Blocksworld domain by adding independent arms to the environment. The HTN method structure is the same. To add the arms, each of the methods and placement operators is given an additional `arm` parameter, and the initial task network contains an `achieve-goal` task for each arm. In the HTN IPC, this domain was used in the total order track, where the `achieve-goals` tasks were serially ordered. This meant that the IPC Multiarm-Blocksworld problems have the same effective solution space as the Blocksworld-HPDDL domains.

The author's affiliation with The MITRE Corporation is provided for identification purposes only, and is not intended to convey or imply MITRE's concurrence with, or support for, thepositions, opinions, or viewpoints expressed by the author. ©2021 The MITRE Corporation. CC BY-NC-ND

References

[Alford, Bercher, and Aha 2015] Alford, R.; Bercher, P.; and Aha, D. 2015. Tight bounds for HTN planning. In *ICAPS*.

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

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

[Slaney and Thiébaux 2001] Slaney, J., and Thiébaux, S. 2001. Blocks world revisited. *Artificial Intelligence* 125(1-2):119–153.

HTN IPC-2020 Domain: Robot

Ron Alford
The MITRE Corporation
ralford@mitre.org

Abstract

The Robot problems from the 2020 HTN International Planning Competition were adapted from the HTN translation papers by Alford et al.. The domain encodes an office delivery problem, and enforces a strategy of only picking up packages that are not in their goal location, and only placing packages in their final location. Package order and navigation are left up to the choices of the planner.

The Robot domain, modeled after the Robot Navigation domain (Kabanza, Barbeau, and St-Denis 1997), was used to evaluate the translation of totally-ordered HTN problems to PDDL (Alford, Kuter, and Nau 2009) and updated in further work translating partially-ordered HTNs to PDDL (Alford et al. 2016). This domain was originally designed to showcase the effectiveness of partial HTN knowledge in guiding planner search. The domains in the above papers encoded a method for picking up objects in their initial locations and placing them in their goal location. Navigation actions were allowed to be inserted by the planner outside of the HTN structure, similar to HTN task insertion planning (Geier and Bercher).

In the IPC, the additional methods were introduce to place the navigation actions in the hierarchy, giving the domain traditional HTN semantics. The new domain contains 4 operators and 11 methods implementing 6 abstract tasks. The problems are tail recursive with a max progression bound of 2, which means any HTN progression of the initial task network results in a task network with at most two tasks (Alford, Bercher, and Aha 2015).

The top level task, `achieve-goals`, has four methods:

- `achieve-goals-move`, `achieve-goals-open`: These change rooms and open door, respectively, and then recurse back to `achieve-goals`
- `achieve-goals-pickup` has a precondition that there is an object in the current room and its not in its goal location. Its subtasks pick up the object, and recurse to the `release` task.
- `finished` has no precondition or subtasks.

The `release` task has three methods:

- `release-open` and `release-move` which call the open and move operators respectively, then recurse back to `release`.
- `release-putdown_abstract` has a precondition that the robot is holding a package while in the package's goal location. It has subtasks to release the package and recurse back to the top level task `achieve-goals`.

The problem generator[1] for the Robot domain takes the number of packages and rooms as a parameter. It generates a problem with the rooms connected in a random acyclic graph, the packages uniformly distributed among the rooms, and the doors closed or open with even chance. The problem generator outputs in PDDL format, which had to be hand adapted for the HDDL format of the HTN IPC (Höller et al. 2020). The IPC contained 30 problems with room counts between 1 and 300, and package counts between 1 and 150.

References

[Alford, Bercher, and Aha 2015] Alford, R.; Bercher, P.; and Aha, D. 2015. Tight bounds for HTN planning. In *ICAPS*.

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

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

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

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

[Kabanza, Barbeau, and St-Denis 1997] Kabanza, F.; Barbeau, M.; and St-Denis, R. 1997. Planning control rules for reactive agents. *Artificial Intelligence* 95(1):67–113.

[1] https://github.com/ronwalf/HTN-Translation/tree/master/examples/robot

HTN IPC-2020 Domains: Towers

Ron Alford
The MITRE Corporation
ralford@mitre.org

Abstract

The Towers domain from the 2020 HTN International Planning Competition was adapted from the HTN translation papers by Alford et al.. The domain encodes a solution to the Towers of Hanoi problem, admitting a unique, optimal solution and having no branches in the search space.

Towers of Hanoi is a puzzle consisting of a set of rings of decreasing size and three pegs. The rings are initially stacked in size-order on one ring, with the smallest at the top. The goal of the puzzle is to move the rings one at a time between the pegs such that the entire tower ends up on the goal peg without ever placing a larger ring on top of a smaller.

Optimal solutions for Towers of Hanoi problems have $2^{|\text{rings}|}-1$ moves. The Towers domain, introduced in works translating HTNs to PDDL (Alford, Kuter, and Nau 2009; Alford et al. 2016), encodes a tail-recursive, optimal solution without any branching in the forward search space. The solution moves the smallest disk in a rotating pattern between the ring. After each move of the smallest ring, it moves the smallest of the other two exposed rings to the larger. There is an initial phase of the methods which determines whether there are an odd or even number of rings, which determines which tower the smallest ring goes to first.

The Towers domain has one operator, five compound tasks, and eight methods implementing those tasks. Any HTN progression of the initial task network of a Towers problem has at most two tasks, giving it a progression bound of 2 (Alford, Bercher, and Aha 2015). The outline of the tasks and methods are as follows:

- The top level task is `(shiftTower t1 t2 t3)`, which shifts the tower from `t1` to `t2` using `t3`. The `m-shiftTower` method has a single subtask, `selectDirection`.
- The `selectDirection` task has two methods, `m-selectDirection` which flips its planned order for shifting the tower, and `selectedDirection`, which initiates the `rotateTower` task.
- The `(rotateTower ?t1 ?t2 ?t3` task has a single method with two subtasks: moving the smallest ring from `?t1` to `?t2`, then starting the `(exchange ?t1 ?t2 ?t3)` task.
- The `(exchange ?t1 ?t2 ?t3)` has three methods:
  - An empty one if `?t1` and `?t3` are clear.
  - `exchangeLR` which moves `?t1` to `?t3` if `?t1` is the smaller.
  - `exchangeRL` which moves `?t3` to `?t1` if `?t1` is the larger.

  The later two methods then recurse to `(rotateTower ?t2 ?t3 ?t1)` to continue the solution.

The problem generator[1] for the Towers domain takes the number of rings as a parameter, and generates problem files with a single tower of rings. The problem generator outputs in PDDL format, which had to be hand adapted for the HDDL format of the HTN IPC (Höller et al. 2020). The IPC contained 20 problems with 1 to 20 rings.

References

[Alford, Bercher, and Aha 2015] Alford, R.; Bercher, P.; and Aha, D. 2015. Tight bounds for HTN planning. In *ICAPS*, volume 25, 7–15. AAAI Press.

[Alford et al. 2016] Alford, R.; Behnke, G.; Höller, D.; Bercher, P.; Biundo, S.; and Aha, D. 2016. Bound to plan: Exploiting classical heuristics via automatic translations of tail-recursive HTN problems. In *ICAPS*, volume 26, 20–28. AAAI Press.

[Alford, Kuter, and Nau 2009] Alford, R.; Kuter, U.; and Nau, D. S. 2009. Translating HTNs to PDDL: A small amount of domain knowledge can go a long way. In *IJCAI*, 1629–1634. AAAI Press.

[Höller et al. 2020] Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. HDDL: An extension to PDDL for expressing hierarchical planning problems. In *AAAI*, volume 34, 9883–9891.

[1]https://github.com/ronwalf/HTN-Translation/blob/master/examples/towers/genTowers.hs

HTN Planning Domain for Deployment of Cloud Applications

Ilche Georgievski
Service Computing Department
Institute for Architecture of Application Systems
University of Stuttgart
firstname.lastname@iaas.uni-stuttgart.de

Abstract

Cloud providers are facing a complex problem in configuring software applications ready for deployment on their infrastructures. Hierarchical Task Network (HTN) planning can provide effective means to solve such deployment problems. We present an HTN planning domain that models deployment problems as found in realistic Cloud environments.

Introduction

The choice of enterprises to have their software applications deployed and run on Cloud infrastructures is motivated by elasticity, flexibility, scalability and high availability, which are the promised benefits of Cloud Computing (Buyya et al. 2009). Before being able to deploy an application, Cloud management teams have to find, choose and configure appropriate software components that will compose the application, thus making it ready for deployment. Solutions to such deployment problems are typically configured manually or using predefined scripts. Both approaches seem impractical for Cloud management as they are error-prone and require strenuous effort to handle a large number of components, versions of components and high interdependence between components (Binz et al. 2014). Cloud providers and Cloud Computing community are therefore in need for approaches and tools that can solve deployment problems automatically (Arshad, Heimbigner, and Wolf 2003).

Artificial Intelligence (AI) planning can provide powerful means to automatically and efficiently search for solutions to deployment problems. Hierarchical Task Network (HTN) planning appears to be particularly suitable as it can incorporate the configuration knowledge otherwise provided by the Cloud management teams. In this paper, we describe HTN planning problems that correspond to deployment problems. We first introduce the component model used for describing deployment problems, and then we describe an HTN planning domain that models such deployment problems.

Component Model

Aelous is a component model used to describe software applications as found in realistic Cloud deployments (Di Cosmo et al. 2014). A central element of Aeolus is a *component*, a manageable software resource that provides and requires functionalities. Each component has three states: *uninstalled*, *installed*, and *running*. State transitions are done using *deployment actions*. For example, we can run an installed component by invoking a *runComponent* action on it. A component may require or provide different functionality at each state. The requirement of functionality is expressed via *require ports*, and providing of functionality through *provide ports*. A component can transition from one state to another only if the functionality the new state requires can be provided by other component(s). When a component goes to a new state, its require ports are bound to appropriate provide ports of other component(s). This process is called *port binding*. Once a component is in the new state, its provide ports become active via *port activation*. Since a component transitions from some state, its ports must be deactivated and unbound via *port deactivation* and *port unbinding*.

This component model has two interesting features. First, since component represent abstract entities, they must be instantiated. The peculiarity comes from the fact that the *creation of new uninitialised component instances* happens on demand during runtime. The second feature is that a cycle may occur between states of different component instances: an instance is expected to provide a functionality, but it is not possible because the instance is required to change its state at the same time (Lascu, Mauro, and Zavattaro 2013). We can deal with such cycles by creating as many instances of the same component as needed, and deploying them in different states. This process is called *instance duplication*.

A *configuration* describes all available components, currently deployed component instances and their states, and current bindings of components via ports. A *deployment problem* consists of an initial configuration, a set of deployment actions, and a request for a new configuration, i.e., application. The solution is a *deployment run*, which is a sequence of deployment actions on components that, when deployed, produce the required configuration.

Deployment Planning Domain

We now describe the `deployment` HTN domain model that encodes deployment problems. Our description is based on the paper in which we introduced the HTN planning approach to solving deployment (Georgievski et al. 2017).

Components, States and Ports

We describe components, instances, and ports using `component instance port` as domain types. Each component type would be represented as an object of `component`. For example, a *Wordpress* component would be represented as `wordpress - component`.

Even though Aelous associates components with states, component instances are the ones to be in a specific state during planning. We encode a state of an instance in a predicate "*(state instance)*", where *state* is a string representing the instance's state, and *instance* is a variable representing the component instance. For example, `(installed w1)` represents a Wordpress instance *w1* in an installed state.

We encode the association of states with ports in a predicate "*(statePort component port)*", where *statePort* is a string describing the type of port in a specific state, and *component* is a variable referring to an abstract component that requires or provides a port represented by the *port* variable. For example, if *Wordpress* requires the *httpd* port in the installed state, we encode it as `(installed-require wordpress httpd)`. Note that such knowledge holds for all instances of the respective component. These predicates are static predicates.

Deployment Actions

We encode all deployment actions as planning actions as follows. Action's parameters correspond either to a component instance variable or to variables of a port and two instances in the case of binding actions (see below). The preconditions and effects of each action capture the semantics of the respective deployment action. Listing 1 shows the action that corresponds to the *startComponent* deployment action, which makes the state of an instance to become installed. It uses a conditional effect within a universal quantifier to activate all the ports associated with the installed state of the component which the current instance belongs to. The encoding of the actions for running, stopping and terminating component instances are similar. There are also binding actions responsible for low-level binding of ports – require ports are bound to provide ports. They are represented by two planning actions. The `bind` action creates a binding between a provide port of some instance and a require port of another one, and the `unbind` action deletes an already established binding between two instances.

The last action is for creating new uninitialised instances. The `createInstance` action shown in Figure 2 uses a domain function to get (and increase) a number that we use to uniquely represent an instance in a predicate as `(instance ?iNum - number)`. The domain function does not take arguments and serves as a counter to keep track of the current value that can be assigned for new instances. The action uses another predicate, `(type ?iNum - number ?c - component)`, to associate a new instance with a particular component.

Configuration Processes

We now describe the encoding of processes needed for configuring applications. The basic process requires satisfaction of dependencies to functionalities provided by components. Let us assume that an instance in an uninstalled state cannot have requirements to be satisfied. We may then consider two abstractions of the basic process. The first one refers to acquiring a component functionality in an installed state, while the second abstraction refers to establishing a functionality in a running state. HTNs naturally enable encoding knowledge at different levels of abstraction; we can formulate tasks and encode high-level strategies in the methods of these tasks before reasoning on low-level primitive tasks (Georgievski and Aiello 2015).

We encode each abstraction as a compound task, namely `install` and `run`. Their methods encode specific configuration processes. One such method encodes the prerequisites for port activation. If the current component instance has require ports that are not active, the method first activates each port and makes a recursive call until all necessary ports are activated. The actual process of port activation is encoded in a separate task, which not only activates a required functionality, but also finds and installs (or runs) a component instance that provides that functionality. An instance with active require ports can then use the functionalities of other components with active provide ports. This is achieved by another method that encodes the port binding. For this process, the method depends directly on the binding actions. In addition to the methods for port activation and binding, there is a method for the case when all require ports are active and bound. To address the satisfaction of all require ports, we use a universal quantifier with implication in the method for

Listing 1: HTN action for starting a component instance.

```
(:action start
  :parameters (?i - instance)
  :precondition (not (installed ?i))
  :effect (and
    (installed ?i)
    (forall (?p - port) (when
      (and (installed-provide ?c ?p)
           (type ?i ?c))
      (active ?p ?i))
    )
  )
)
```

Listing 2: HTN action for creating an uninitialised component instance.

```
(:action createInstance
  :parameters (?c - component)
  :precondition ()
  :effect (and
     (instance (instance-number))
     (type (instance-number) ?c)
     (increase (instance-number) 1)
  )
)
```

both tasks, `install` and `run`. In the case of `run`, we have to deactivate the ports that will be no longer provided by the instance in the installed state. The process of port deactivation is similar to the process of port activation and it uses port unbinding. The process of port unbinding is more complex than the binding one, and requires checking for constraint violation. That is, we have to take care of active provide ports bound to active require ports. We use a separate task to encode the port unbinding. The `unbindPorts` task does nothing when the port is bound and needed for the next transition. When all necessary constraints are satisfied, it unbinds a specific port and recursively calls itself.

There are methods in `install` and `run` that deal with the case when there are no required functionalities for an instance. This means that we need a transition which can be handled by installing the component instance directly. In the case of running an instance, we invoke the port deactivation task to ensure a valid transition to the running state.

The modelling of the transitions from a running state to an installed state and further to an uninstalled state is analogous to the encoding of the tasks we described so far.

Finally, we encode instance duplication as a separate method. The method makes sure that the current component instance is in a specific state and it has at least one provide port bound. Consequently, a new component instance is created either in an installed state or in a running state, depending on the type of configuration.

Final Remarks

Our HTN planning domain model encodes realistic Cloud deployment problems. Using this domain, one can generate a problem file by specifying components and ports as objects, component states and ports as predicates, currently deployed instances as predicates, current states of deployed instances as predicates, bindings as predicates, and initialising the domain function to some value. Listing 3 shows an example of a problem file for the deployment of Wordpress, and Listing 4 shows its plan. Finally, HTN planning problems with varying difficulty can be generate automatically by manipulating the states and ports of components, as described in (Georgievski et al. 2017).

Acknowledgments

We thank Faris Nizamic, Alexander Lazovik and Marco Aiello for the discussions on earlier versions of the domain. We also thank Gregor Behnke for the valuable insights on the domain encoding and for transforming the domain to a suitable specification for the IPC 2020 on HTN planning.

Listing 3: HTN problem file

```
(define (problem p)
 (:domain deployment)
 (:objects
  wordpress mysql apache2 - component
  httpd mysql-in mysql-up - port
 )
 (:init
  (installed-require wordpress httpd)
  (running-require wordpress httpd)
  (running-require wordpress mysql-up)
  (installed-provide apache2 httpd)
  (installed-provide mysql mysql-in)
  (running-provide mysql mysql-up)
  (= (instance-number) 0)
 )
 (:htn
  :tasks (run wordpress)
  :ordering ()
  :constraints ()
 )
)
```

Listing 4: Example plan for the problem in Listing 3

```
1.  (createInstance w0)
2.  (createInstance w1)
3.  (start a1)
4.  (bind httpd w0 a1)
5.  (start w0)
6.  (createInstance m2)
7.  (start m2)
8.  (run m2)
9.  (bind mysql-up w0 m2)
10. (run w0)
```

References

Arshad, N.; Heimbigner, D.; and Wolf, A. L. 2003. Deployment and dynamic reconfiguration planning for distributed software systems. In *IEEE International Conference on Tools with Artificial Intelligence*, 39–46.

Binz, T.; Breitenbücher, U.; Kopp, O.; and Leymann, F. 2014. *TOSCA: Portable Automated Deployment and Management of Cloud Applications*. Springer. 527–549.

Buyya, R.; Yeo, C. S.; Venugopal, S.; Broberg, J.; and Brandic, I. 2009. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Gener. Comput. Syst.* 25(6):599–616.

Di Cosmo, R.; Mauro, J.; Zacchiroli, S.; and Zavattaro, G. 2014. Aeolus: A component model for the cloud. *Information and Computation* 239:100–121.

Georgievski, I., and Aiello, M. 2015. HTN planning: Overview, comparison, and beyond. *Artificial Intelligence* 222:124–156.

Georgievski, I.; Nizamic, F.; Lazovik, A.; and Aiello, M. 2017. Cloud Ready Applications Composed via HTN Planning. In *IEEE International Conference on Service Oriented Computing and Applications*, 23–33.

Lascu, T. A.; Mauro, J.; and Zavattaro, G. 2013. A Planning Tool Supporting the Deployment of Cloud Applications. In *IEEE International Conference on Tools with Artificial Intelligence*, 213–220.

Snake Domain for HTN IPC 2020

Maurício Cecílio Magnaguagno
Independent researcher
maumagnaguagno@gmail.com

Abstract

This is a description of the Snake domain and problem generator submitted to the HTN IPC 2020 total order track. In the Snake domain the goal is to hunt mice spread over multiple locations, with one or more snakes that get longer as they strike each mouse.

Introduction

The Snake domain is based on the homonymous game genre, in which snakes move to clear locations or strike a nearby mice in a grid/graph-based scenario, the mice do not move as they are too afraid. Each snake occupies one or more adjacent locations due to their long body. The goal is to hunt all the mice or have the snakes occupying certain locations (which forces them to eat and grow). Multiple plans may exist in some scenarios due to snakes being able to strike mice with different orderings and paths. Plans contain zero or more movement actions and one strike per mouse. Differently from the game where usually only one mouse is visible at a time, all mice are visible to give more choice. The domain was motivated by the creative way in which one can describe the snake actions without updating all the snake parts and the little amount of objects required to describe a snake. This paper presents the Snake domain and problem generator[1] for PDDL (McDermott et al. 1998), HDDL (Höller et al. 2020) and (J)SHOP input language (Ilghami and Nau 2003).

Domain

The domain requires *:typing*, *:equality* and *:negative-preconditions* in PDDL, and also *:method-preconditions* and *:universal-preconditions* in HDDL. The JSHOP domain implicitly has the same HDDL requirements. Universal preconditions are used to verify that every location does not contain a mouse and the hunting task is complete.

Types

All objects are either *snake* or *location*. This removes the need to have more objects to define each mouse and snake parts. Removing such objects makes descriptions simpler and grounding faster due to fewer parameters. We use *(mouse-at ?location)* instead of *(at ?mouse ?location)* to remove the *?mouse* parameter from the *strike* action. If we had opted for snake parts we would have multiple descriptions of each long snake, causing a state-space explosion.

Predicates

The state is described by only a few predicates. Locations are *occupied* to avoid overlapping snake parts and mice during movement actions, and also used to simulate walls. Locations that are *adjacent* constrain the range of actions. A snake *head* location is used to constrain the range of actions of each snake. The sequence of locations occupied by each snake are *connected*, with the last part being the *tail*.

Actions/Operators

Three actions/operators exist in this domain. The *strike* action represents the mouse being consumed by an adjacent snake head. Two movement actions are used to describe a single or multiple location snake movement, *move-short* and *move-long*, respectively. Move was split in two to minimize the amount of ground actions without the use of disjunctions. The JSHOP version also contains explicit *visit/unvisit* operators to avoid infinite loops. The signatures of actions are shown in Listing 1.

Listing 1: Signatures of *Snake* actions with types ommited.

```
(:action strike :parameters (
  ?snake ?headpos ?foodpos))
(:action move-short :parameters (
  ?snake ?nextpos ?snakepos))
(:action move-long :parameters (
?snake ?nextpos ?headpos ?bodypos ?tailpos))
```

Tasks and Methods

Two tasks are described in the JSHOP and HDDL versions, with 5 methods in total. The first task is *hunt*, with zero parameters, used as the main task. Two methods are used for this task, a recursive one to select one snake that will strike a mouse, and a base one for no more mice. The base case is described after the recursive method as it happens only once, when all mice have been consumed.

The second task is *move*, with a snake, its head and goal location as parameters. Here we have a base method and two recursive ones to use the *move-long* and *move-short* actions.

[1] https://github.com/Maumagnaguagno/Snake

The *move-base* case is described first to avoid redundant expansions in planners that follow the description order. The *move-short* is the last case described as it is less common. The signatures of tasks and their related methods are shown in Listing 2.

Listing 2: Signatures of *Snake* tasks and related methods.

```
(:task hunt :parameters ())
(:method hunt_all :parameters (?snake
  ?foodpos ?snakepos ?pos1))
(:method hunt_done :parameters ())
(:task move :parameters (?snake
  ?snakepos ?goalpos))
(:method move-base :parameters (
  ?snake ?snakepos ?goalpos))
(:method move-long-snake :parameters (
  ?snake ?snakepos ?goalpos ?pos2
  ?bodypos ?tailpos))
(:method move-short-snake :parameters (
  ?snake ?snakepos ?goalpos ?pos2))
```

Problem

Each problem contains snakes and locations as objects. Each snake must contain at least a head and tail described in the initial state. If head and tail are on the same location, single location snake, there is no need to connect snake parts. Each mouse location must be described in the initial state. Locations that contain snake parts, mice or walls are occupied. Locations must be adjacent to one another to describe possible paths. Adjacencies are usually symmetrical, *(adjacent l1 l2) (adjacent l2 l1)*, and grid-based, but are not limited to.

For goal-based planning it may include snakes' final configuration and mice not existing anymore. For task-based planning it may include movement and hunting tasks. Due to the possibly large amount of mice, it is recommended to use a quantifier to describe a goal state without mice or tasks to hunt every mouse.

Problem generator

Currently a text representation, like the one from Sokoban[2], can be used with our problem generator. Each character in a text file represents one element of the Snake problem in a grid-based scenario:

- *Space*: clear location
- @: snake head location
- $: snake body location
- *: mouse location
- #: wall location

Currently limited to a single snake with snake parts adjacent only to previous and next locations to avoid ambiguity. Walls are converted to always occupied locations, but could also be represented as lack of adjacencies to these locations, which would be harder to manually modify later. Multiple problems in this format are already available, they were manually crafted to generate longer solutions or force certain paths for the snake to be able to strike all mice.

The current problem generator converts all **.snake* files in the current folder or the ones provided as arguments according to a *type* argument, generating **.snake.type* files. Type includes *pddl*, *hddl* and *jshop*.

[2]http://www.sokobano.de/wiki/index.php?title=Level_format

Example

The content of the input *pb2.snake* is presented in Listing 3. With the execution of the problem generator, *ruby pbgenerator.rb hddl pb2.snake*, we obtain an HDDL equivalent problem. The output *pb2.snake.hddl* is presented in Listing 4.

```
*␣␣\n
␣␣$\n
␣␣@
```

Listing 3: Snake input file example with 3x3 grid, two-parts snake and a mouse.

Listing 4: HDDL description of converted *pb2.snake*.

```
(define (problem pb2) (:domain snake)
  (:objects viper - snake
            px0y0 px1y0 px2y0
            px0y1 px1y1 px2y1
            px0y2 px1y2 px2y2 - location)
  (:init (head viper px2y2)
         (connected viper px2y2 px2y1)
         (tail viper px2y1)
         (mouse-at px0y0)
         (occupied px0y0)
         (occupied px2y1)
         (occupied px2y2)
         (adjacent px0y0 px1y0)
         ... ; Adjacencies ommited
         (adjacent px2y2 px2y1))
  (:htn :subtasks (hunt)) )
```

Conclusion

This domain presents several features to help planner testing. All planning instances can be described in the compact format used by the generator, converted to images and easily modified by hand. The planning instances can scale indefinitely, as larger grids accept more mice and longer snakes, however it requires a smart random level generator to create such larger instances with unique challenges. Heuristic planners can estimate which snake is closer to each mouse to minimize actions, while considering that long snakes create moving walls that affect such estimations. Numeric planners could take even more advantage in regular grids. In the future we expect to improve the problem generator with multiple snakes and their goal locations. Multiple snakes could also modify the domain, with the requirement of moving all snakes every time-step, like the real game.

References

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.

Ilghami, O., and Nau, D. S. 2003. A General Approach to Synthesize Problem-Specific Planners. Technical Report CS-TR-4597, Maryland University, Dept of Computer Science, College Park, Maryland.

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

The Barman-HTN Domain for IPC 2020

Max Waters, Lin Padgham, Sebastian Sardina
RMIT University, Melbourne, Australia
max.waters@rmit.edu.au, sebastian.sardina@rmit.edu.au, lin.padgham@rmit.edu.au

Abstract

The *Barman-HTN* domain is an HTN decomposition of the IPC *Barman* domain. It extends *Barman* with a task network that guides cocktail creation while retaining the primitive operators that made the original so challenging for the delete-relaxation heuristic.

The *Barman-HTN* domain is an HTN extension of the well-known IPC *Barman* domain (López, Celorrio, and Olaya 2015), in which cocktails must be prepared out of various ingredients. Solving a *Barman* instance requires careful management of limited resources. For example, shot glasses and shakers must be clean and empty before use, but filling one with an ingredient deletes both of these conditions, meaning that it must be emptied and cleaned before re-use. This makes *Barman* particularly challenging for planners that use the delete-relaxation heuristic: as action preconditions are frequently deleted and can only be restored by executing further actions, delete-relaxation tends to produce overly optimistic estimates.

This property of the domain operators, combined with recent interest in the delete-relaxation heuristic in HTN planners (Höller, Bercher, and Behnke 2020) and the fact that the cocktail construction task can be naturally decomposed into subtasks, suggest that *Barman* is a suitable basis for an HTN benchmark domain.

Barman-HTN extends *Barman* with an HDDL (Höller et al. 2020) task network that guides the pouring and mixing of ingredients and also provides careful resource management. For example, the method `MakeCocktail` (Figure 1) decomposes the task of mixing and shaking a cocktail. The first subtask, `AchieveCleanShaker`, ensures that the shaker is clean and empty, and the two instances of `AchieveContainsShakerIngredient` ensure that it contains the required ingredients. The next two steps, `AchieveHolding` and `AchieveHandEmpty`, ensure that one hand is empty and the other is holding the shaker. These subtasks satisfy the preconditions of the final step, the action `shake`, which results in the shaker containing the cocktail.

Resource management is handled by tasks and methods that bring about a required condition from any given state. For example, the task `AchieveHolding(?h, ?c)` produces the condition `holding(?h, ?c)`, and is decomposed by two methods. If hand `?h` is already holding container `?c`,

```
(:method MakeCocktail
    :parameters (?s - shaker ?c - cocktail
       ?i_1, i_2 - ingredient ?h_1, ?h_2 - hand)
    :task (AchieveContainsShakerCocktail ?s ?c)
    :precondition (and
       (cocktailPart1 ?c ?i_1) (cocktailPart2 ?c ?i_2)
       (not (= ?h_1 ?h_2)))
    :ordered-subtasks (and
       (AchieveCleanShaker ?s)
       (AchieveContainsShakerIngredient ?s ?i_1)
       (AchieveContainsShakerIngredient ?s ?i_2)
       (AchieveHolding ?h_1 ?s)
       (AchieveHandEmpty ?h_2)
       (shake ?c ?i_1 ?i_2 ?s ?h_1 ?h_2)))

(:method PickUp
    :parameters (?h - hand ?c - container)
    :task (AchieveHolding ?h ?c)
    :precondition (not (holding ?h ?c))
    :ordered-subtasks (and
       (AchieveHandEmpty ?h) (AchieveOnTable ?c)
       (grasp ?h ?c)))
```

Figure 1: The `MakeCocktail` and `Pickup` methods.

then the empty method `AchieveHoldingNull` is applicable. Otherwise, `PickUp` (Figure 1) decomposes the task into `AchieveHandEmpty(?h)` and `AchieveOnTable(?c)`, that satisfy the preconditions of the primitive action `grasp(?h, ?c)` by ensuring that `?h` is empty and `?c` can be picked up, respectively. The task network contains other such decompositions for resource management tasks such as cleaning glasses and shakers, and emptying hands.

References

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems. In *AAAI 2020*, 9883–9891. AAAI Press.

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

López, C. L.; Celorrio, S. J.; and Olaya, A. G. 2015. The deterministic part of the seventh International Planning Competition. *Artificial Intelligence* 223: 82–119.

The Hierarchical Satellite Domain

Bernd Schattenberg[1]
3B intelligent solutions, Germany, schattenberg@3b-intelligent-solutions.com
[1] The domain was created while still being at the Institute of Artificial Intelligence of Ulm University

Abstract

The *Satellite* domain is one of the classical benchmark domains in the canon of the International Planning Competition. This paper describes our hierarchical take on it.

Introduction

The hierarchical *Satellite* domain is inspired by space applications that are a first step towards the "Ambitious Spacecraft" as described by David Smith at the AIPS 2000 conference (Smith, Frank, and Jónsson 2000). It involves planning a set of stellar observation tasks for multiple autonomous satellites, each equipped with slightly different but possibly overlapping technologies. The equipment consists of observation instruments with different characteristics in terms of data productions, so-called *modes* like thermal images, x-ray, etc., for which corresponding calibration targets are defined. Satellites are motile and can be oriented towards arbitrary stellar target objects by slewing the complete platform between different attitudes/directions. A benchmark problem in this domain is consequently given by an initial state that describes the satellite configurations and stellar phenomena positions, while the goal state specifies of which observation targets an image has to be taken in which mode.

The *Satellite* domain has been introduced as a classical benchmark to the planning community in the 2002 installment of IPC. We have developed a hierarchical version of it in order to analyse planning strategy designs for hybrid planning systems (Schattenberg, Weigl, and Biundo 2005; Schattenberg, Bidot, and Biundo 2007), from which the presented, purely hierarchical version has been derived.

This document focuses on the design decisions that led to the hybrid planning domain model for the formal framework introduced in (Biundo and Schattenberg 2001; Schattenberg 2009), i.e., on adding hierarchical features to a non-hierarchical domain model (cf. (Pragst et al. 2014)).

Types and Relations

The first step in translating the original *Satellite* domain into a hierarchical formalism is to introduce a type hierarchy. While the PDDL encoding already defined the types for satellites, directions, instruments, and (image) modes, we think that this does not capture an essential feature of the problem instances: the defined directions are obviously divided into the actual observation phenomena that are of scientific interest and attitude points that are only used for calibration purposes. This aspect is incorporated in the type hierarchy by providing a general purpose `Direction` as super-type for `Calib_Direction` and `Image_Direction`.

In our formal hybrid-planning framework, we also annotated the direction super type to be *abstract*, i.e. a conceptual type for which no constant declaration is allowed. The rationale for such a language feature is to identify types that are intended to be exclusively used for structuring the application domain concepts, thus supporting modelling tools to validate problem and domain consistency.

When it comes to specifying relation symbols for expressing predicate sentences or facts about the world state, HDDL and PDDL models explicitly provide the relation symbols' signatures in its declaration header. However, they do not explicitly denote whether actions (are allowed to) manipulate the respective attributes, or, in more formal terms, they do not discriminate flexible and rigid relations. Instead, state-invariant features are typically extracted from the domain model during pre-processing.

The original model's documentation plus some common sense suggest to adhere to the following partitioning of relation symbol declarations: Flexible relations are

- $\texttt{pointing}_{\texttt{Satellite,Direction}}$
- $\texttt{power\_avail}_{\texttt{Satellite}}$
- $\texttt{have\_image}_{\texttt{Image\_Direction,Mode}}$
- $\texttt{power\_on}_{\texttt{Instrument}}$
- $\texttt{calibrated}_{\texttt{Instrument}}$

The `pointing` relation is used for expressing that a satellite platform (the first argument in any atom over this relation), and with it all on-board instruments, aim at a given direction (the second argument of such atoms). Slewing the satellite therefore implicitly controls the orientation of the desired instrument as well. `power_avail` and `power_on` reflect that energy is a limited resource on the observation platform and that therefore only one instrument can be served at a time. On-board observation systems typically have to take reference images for calibrating the sensors and if an instrument is ready for taking images, its status changes to `calibrated`. In a state in which the image of a phenomenon is finally taken a respective atom over `have_image` is supposed to hold.

The relations representing state-invariant facts are:
- `on_board`$_{\text{Instrument,Satellite}}$
- `supports`$_{\text{Instrument,Mode}}$
- `calibration_target`$_{\text{Instrument,Calib\_Direction}}$

This includes the relation for modelling which instruments which satellite carries, which kind of sensor the instrument provides, and what the reference object for calibrating a given instrument is.

Actions, Tasks, and Methods

The action specifications can be directly taken from the original non-hierarchical PDDL description:
- `turn_to`$_{\text{Satellite,Direction,Direction}}$
- `switch_on`$_{\text{Instrument,Satellite}}$
- `switch_off`$_{\text{Instrument,Satellite}}$
- `calibrate`$_{\text{Satellite,Instrument,Calib\_Direction}}$
- `take_image`$_{\text{Satellite,Image\_Direction,Instrument Mode}}$

The intended meaning of these five operator signatures be self explanatory. The corresponding action definitions basically implement conditional switching operations for the intended state feature, e.g.:

```
(:action switch_on
   :parameters (?so_i - instrument
                ?so_s - satellite)
   :precondition
      (and (on_board ?so_i ?so_s)
           (power_avail ?so_s))
   :effect
      (and (power_on ?so_i)
           (not (calibrated ?so_i))
           (not (power_avail ?so_s))))
```

It is the operator that routes energy to a given instrument on the observation platform, it switches power from available to not available. In this way, no two instruments can be used in parallel on one satellite. The second precondition for the action assures the required instrument to be on board the given satellite, which merely enforces a consistent binding of the two parameters. Please note that although this particular style of modeling introduces a considerable amount of redundancy in the *Satellite* action definitions, any processing that is aware of the underlying state-invariance will reduce the unnecessary branching at this point.

When we analyzed the IPC benchmark problems for this domain, it occurred to us that there obviously exists an *intended procedure* for taking satellite images and that all solutions follow that pattern with minor deviations: Making an observation for a given sensor mode and phenomenon firstly consists of choosing a suitable instrument, which in turn indirectly determines the satellite that performs the observation. In a second step, the instrument has to be routed energy to and properly calibrated. The satellite finally slews in the direction of the target phenomenon and takes the image.

This procedure is plausible enough to be considered not as a specification artefact that has been accidentally introduced by the competition initiators but as an underlying principle in the *Satellite* domain and consequently a clue for a well-reasoned action abstraction. An apparent structure is to build an abstraction for each of the two phases: preparing the instrument and taking the picture becomes an abstract task `do_observation` with parameters for the desired phenomenon to observe and the mode to support. The preparation phase seems to require an abstraction hierarchy on its own, in order to encapsulate the different ways of getting the sensory system on-line (the instrument is already on and calibrated, some other instrument has to be turned off first in order to raise the energy level properly, etc.). We therefore introduced an abstract action for activating the instrument and for dealing with the calibration. The resulting (complex) tasks are consequently the following three:
- `do_observation`$_{\text{Image\_Direction,Mode}}$
- `activate_instrument`$_{\text{Satellite,Instrument}}$
- `auto_calibrate`$_{\text{Satellite,Instrument}}$

We intended to define the complex task schemata in the fashion of ABSTRIPS operator reductions (Sacerdoti 1974). That means, we do not employ state abstraction axioms (cf. (Biundo and Schattenberg 2001)) but simply generalize the preconditions and effects of the primitive implementations, like in the following example:

```
(:task activate_instrument
   :parameters (?ai_s - satellite
                ?ai_i - instrument)
   :precondition
      (and (on_board ?ai_i ?ai_s))
   :effect (and (power_on ?ai_i)))
```

Given these complex and primitive tasks, the methods of the domain model set up a decomposition hierarchy that implements the different observation procedures as described above. Please note that this decomposition hierarchy does not impose semantic restrictions on the solution space.

The following method `method0` implements an observation by sequentially activating the instrument, turning the satellite, and taking the image:

```
(:method method0
 :parameters
  (?d_prev - direction  ?sat - satellite
   ?d_im - image_direction
   ?i - instrument      ?mode - mode)
 :task
  (do_observation ?d_im ?mode)
 :subtasks (and
  (task0 (activate_instrument ?sat ?i))
  (task1 (turn_to ?sat ?d_im ?d_prev))
  (task2 (take_image ?sat ?d_im ?i ?mode)))
 :ordering (and (task0 < task1)
                (task1 < task2))
 :constraints (and
  (sortof ?d_im - image_direction)
  (not (= ?d_im ?d_prev))))
```

The parameters section introduces all variable names used for task node parameters and variable constraints. By binding the same variable consistently to different sub-task expressions, the corresponding task schema parameters are explicitly co-designated. For example, the target image direction for the observation task is the same for the slewing task `turn_to` and the actual image taking.

The abstract activation task `task0` can further be decomposed into two variants, one dealing with another instrument having to be turned off first, the other for situations in which the satellite has energy already available.

The implementation of the calibration process `auto_calibrate` is either atomic in the context of other observation tasks or it has to perform a preparatory slew into the calibration direction first.

The main combinatorial problem in this domain boils down to the question of how to establish the `pointing` state features. Along the decomposition hierarchy, any observation is self-contained such that the turning operation after a calibration step is properly instantiated and eventually will be fully causally supported from within the surrounding network actions. If a plan, however, contains multiple observation operations that are only developed to an intermediate level, there is typically some confusion about causal support with respect to the orientation of the satellite. Let us therefore briefly investigate the issues of implementing an observation in the presence of other observations.

A problem in the *Satellite* domain is typically given by a number of abstract observation tasks. The domain model offers four implementation variants, the applicability of which depends on the observation contexts:

1. First, the instrument is activated, then the satellite turns to the direction of scientific interest and finally takes the image. This is the base case for isolated (sub-) problems as explained above.
2. The instrument might be properly calibrated from previous observations. In this case, it suffices to slew the satellite and take the image.
3. Problems with many jobs may take advantage of decompositions that provide an activation-imaging skeleton for which the slews can be filled in later by task insertion modifications. This method therefore provides the causal information that connects activation and usage of the sensor. Please note that this requires planners that perform task insertion as well.[1]
4. If the configuration supports task insertion (like in the previous case) or if we have to deal with exceptional situations in which more than one image is required of a phenomenon, the fourth variant solely consists of a direct translation into taking the image.

Concluding Remarks

Satellite induces refinement spaces that contain many isomorphic plans, constructed around exactly one complex task: observation. The different methods do thereby not provide *alternative* ways of performing that task but rather define the configuration or situational environment of the observation process: with or without calibration, with or without preparation slews, and the like.

But the *Satellite* domain has several nice properties due to which it qualifies as an interesting demonstration and benchmark domain for hierarchical planning systems.

The main advantage is the intuitive simplicity of the application domain and the underlying principles. This also holds for future model extensions like incorporating temporal information, addressing energy consumption, etc. Any modification can easily be explained and its effects on the solution generation process investigated. It is also a relatively simple task to algorithmically generate problems, to validate solutions, and to judge problem complexity as well as solution quality. In contrast to other simple benchmark scenarios, planning for satellite observations exhibits satisfying variety and extension options.

Another observation on *Satellite* benchmark problems is that this domain allows to control problem complexity, for example in order to determine the scaling behaviour of a strategy, in several dimensions.

For example, the mildly sophisticated scaling by cloning observation jobs and scientific equipment, leads to an increasing number of self-similar sub-problems. Although this may be the intended scientific focus, it has to be taken into account that this kind of complexity may not favour a generally well performing search strategy. It makes solving ten times more observations “more difficult” in a similar way than stacking ten times more blocks did for non-hierarchical planning at the time.

In contrast, we can define benchmark problems with an increasing number of observation jobs that require an increasing number of modes and instruments on a constant number of satellites. This induces an increasing number of interacting sub-goals and causal interferences. As a consequence, solution density in the search space will decline.

A last aspect is the amount of overlapping target requirements, respectively instrument capabilities: if a set of observations can be performed by single platforms sequentially as well as by multiple platforms in parallel, optimality of the solutions becomes more and more an issue.

[1]Like PDDL, HDDL requires types to be disjunct and therefore no calibration target can, at the same time, be an object of scientific interest, which in turn allows no solutions for this method.

References

Biundo, S., and Schattenberg, B. 2001. From abstract crisis to concrete relief – a preliminary report on combining state abstraction and HTN planning. In *ECP 2001*, 157–168. AAAI Press.

Pragst, L.; Richter, F.; Bercher, P.; Schattenberg, B.; and Biundo, S. 2014. Introducing hierarchy to non-hierarchical planning models - a case study for behavioral adversary models. In *PuK 2014*.

Sacerdoti, E. D. 1974. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence* 5(2):115–135.

Schattenberg, B.; Bidot, J.; and Biundo, S. 2007. On the construction and evaluation of flexible plan-refinement strategies. In *KI 2007*, 367–381. Springer.

Schattenberg, B.; Weigl, A.; and Biundo, S. 2005. Hybrid planning using flexible strategies. In *KI 2005*, 249–263. Springer.

Schattenberg, B. 2009. *Hybrid Planning & Scheduling*. Ph.D. Dissertation, Ulm University, Germany.

Smith, D. E.; Frank, J.; and Jónsson, A. K. 2000. Bridging the gap between planning and scheduling. *The Knowledge Engineering Review* 15(1):47–83.

The Hierarchical Woodworking Domain

Bernd Schattenberg[1,*] and **Pascal Bercher**[2,*]
[1] 3B intelligent solutions, Germany, schattenberg@3b-intelligent-solutions.com
[2] The Australian National University, Australia, pascal.bercher@anu.edu.au
[*] The domain was created while still being at the Institute of Artificial Intelligence of Ulm University, Germany

Abstract

The *Woodworking* domain is one of the classical benchmark domains in the canon of the International Planning Competition. This paper describes our hierarchical take on it.

Introduction

The hierarchical *Woodworking* domain models workflows in a workshop setting. Wooden boards are cut into parts of required sizes, which are planed, smoothened, and finally painted in specified colours and qualities. The various spray and varnish paints thereby require different preparation treatments of the respective wooden surface. Combinations of these process steps into proper workflows are provided by the decomposition methods.

The main causal interactions on a task level occur when some of the heavier workshop tools abrade the surface of wooden items, thereby undoing previous treatment steps. Other minor planning-sub-problems emerge when some of the machinery involved only allows for processing one item at a time. In its current version, this merely imposes limitations on possible plan linearisations but may become subject to plan optimization for resource-aware planners.

The *Woodworking* domain has been introduced as a benchmark to the planning community in 2008 for IPC 6. We have developed a hierarchical version of it in order to analyse planning strategy designs for hybrid planning systems using landmarks (Elkawkagy et al. 2012; Bercher, Keen, and Biundo 2014) and this domain model has finally been translated into the current, purely hierarchical version.

This short description focuses on the design decisions that led to the hybrid planning domain model for the formal framework introduced by Biundo and Schattenberg (2001) and Schattenberg (2009), that means, on the specifics of adding hierarchical features to a non-hierarchical domain model (cf. the work by Pragst et al. (2014)).

Mechanics of the Model

The type hierarchy of *Woodworking* establishes three main categories of objects: the wooden targets of creative handicraft (`woodobj`), workshop machines (`machine`), and a general object type. Wooden objects can be either boards or parts with the latter being obtained from the former by cutting them out. The type `object` is the most general type and an entry point for declaring constants representing specific wood materials, colours, and the like.

Regarding the state-variant features, most of the predicates describe the processing states of the processed wooden objects. This includes the following:

- $\texttt{unused}_{\texttt{part}}$
- $\texttt{boardsize}_{\texttt{board,aboardsize}}$
- $\texttt{treatment}_{\texttt{part,treatmentstatus}}$
- $\texttt{surface\_condition}_{\texttt{woodobj,surface}}$
- $\texttt{colour}_{\texttt{part,acolour}}$
- $\texttt{wood}_{\texttt{woodobj,awood}}$
- $\texttt{available}_{\texttt{woodobj}}$

State features built from `unused` and `available` serve as semaphore, respectively book-keeping implementations for the pre-defined pool of part and wooden object constants. E.g., once a wooden object is used in a process step as a `part`, i.e., if a process step objective is assigned the respective constant, that very `part` constant is taken from the pool of available constants by negating its `unused` property. As a side effect of this technique, the `wood` property has to be passed on from the raw board (for which it's technically unchangeable) to its processed part artefact.

While the rest of the above state features is straight forward with more intuitive semantics, the following state-invariant relations require some examination:

- $\texttt{machine\_present}_{\texttt{machine}}$
- $\texttt{has\_colour}_{\texttt{machine,acolour}}$
- $\texttt{goalsize}_{\texttt{part,apartsize}}$
- $\texttt{boardsize\_successor}_{\texttt{aboardsize,aboardsize}}$
- $\texttt{grind\_treatment\_change}_{\texttt{treatmentstatus,treatmentstatus}}$
- $\texttt{is\_smooth}_{\texttt{surface}}$
- $\texttt{contains\_part}_{\texttt{board,part}}$

`Machine_present` denotes the availability of mobile workshop machines and corresponds to availability of wooden objects – of course, machines are not used up in the process. Similarly, `colour` is used to describe painted parts, while `has_colour` represents the colour a respective workshop machine has at its disposal.

Sizes and treatment states are modeled by explicitly stating the available constants in a problem description. On these values, a simple symbolical computation is axiomatised implicitly in the task specifications: Parts, the process target objects, can have a `goalsize` of small, medium or large. On the other hand, boards, i.e. the process source materials, are described via the state-variant `boardsize`, which is supposed to have at least three discrete values. The board size symbols are arranged according to the facts over the `boardsize_successor` predicate. The rationale is now a very high-level abstraction of a wood-cutting process, in which small sized parts reduce the board size by one `boardsize_successor`, medium sized ones by two, and large sized ones by three.

Please note that the current version of the hierarchical `Woodworking` domain, as does its non-hierarchical origin, does not yet support a re-use of the remaining boards after a part has been cut from them. Furthermore, the intention behind `contains_part` was unfortunately not documented, as it had not been used in any domain element.

The remaining physics of surface treatment by grinding (levels of removing varnish from wooden surfaces) is represented by the `grind_treatment_change` in a similar way like what we have shown for board sizes.

Regarding action specifications, the following operation for varnishing a part by means of immersion can serve as a common example for the domain:

```
(:action do_immersion_varnish
  :parameters (?p - part
               ?m - immersion_varnisher
               ?c - acolour
               ?s - surface)
  :precondition
   (and (available ?p)
        (has_colour ?m ?c)
        (surface_condition ?p ?s)
        (is_smooth ?s)
        (treatment ?p untreated))
  :effect
   (and (not (treatment ?p untreated))
        (treatment ?p varnished)
        (colour ?p ?c)))
        (not (colour ?p natural))
```

Similar actions specifications are used for other means of applying colour to the wood object, basically depending on surface condition and treatment status. Grinding and planing are implemented analogously, emphasising the change of surface condition, colour stripping, and the like.

When analysing the original IPC benchmark problems for this domain, it becomes apparent that the *intended procedure* for *processing* wooden parts follow a general pattern: cutting and sawing a board in order to obtain a suitably sized part, grinding or planing that part to achieve the desired surface condition, and finally applying a specific paint to realize a specific colour and treatment. The hierarchical *Woodworking* domain captures this pattern of sub-processes by defining corresponding abstract tasks like `cut_and_saw`, `grindNplane` and the like. The resulting decomposition hierarchy is relatively flat with one major intermediate level of abstract tasks that allow for alternative decompositions into the process options as described above.

We defined the complex task schemata in the fashion of ABSTRIPS operator reductions (Sacerdoti 1974). That means, we do not employ state abstraction axioms as described by Biundo and Schattenberg (2001) but simply generalise the preconditions and effects of the primitive implementations. The most abstract task, the processing objective, is thus defined as follows:

```
(:task process
  :parameters (?p - part ?c - acolour
               ?oldS - surface
               ?newS - surface)
  :effect (and (colour ?p ?c)))
```

On this level of abstraction, processing consists of colouring a wooden part with all causal interactions delegated to the expansion methods. The methods themselves implement the different process variants by combining related tasks into modular subroutines. Please note that the decomposition hierarchy does not impose semantic restrictions on the solution space.

Properties of the Model

The domain is partially ordered and acyclic. It contains six abstract tasks, 13 primitive tasks, and 14 methods, where each task has between two and four methods. The IPC set contains 30 problem instances of various degrees of hardness. The first eleven instances were modeled by hand by the authors and are relatively easy with maximal shortest solution lengths of 15 steps. The remaining problem instances were created by a random generator, written by Gregor Behnke (based on an existing one for the original domain). The hardest instance has a shortest solution with 178 steps. Using the grounder by Behnke et al. (2020), we can report that the number of ground primitive and abstract tasks as well as decomposition methods ranges from a few dozen for the lower first (smaller) problem instances until 87.680 primitive tasks, 120.819 abstract tasks, and 592.235 decomposition methods for the largest problem instance.

References

Behnke, G.; Höller, D.; Schmid, A.; Bercher, P.; and Biundo, S. 2020. On succinct groundings of HTN planning problems. In *Proceedings of AAAI 2020*, 9775–9784. AAAI.

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid planning heuristics based on task decomposition graphs. In *Proceedings of SoCS 2014*, 35–43. AAAI.

Biundo, S., and Schattenberg, B. 2001. From abstract crisis to concrete relief – a preliminary report on combining state abstraction and HTN planning. In *Proceedings of ECP 2001*, 157–168. AAAI Press.

Elkawkagy, M.; Bercher, P.; Schattenberg, B.; and Biundo, S. 2012. Improving hierarchical planning performance by the use of landmarks. In *Proceedings of AAAI 2012*, 1763–1769. AAAI Press.

Pragst, L.; Richter, F.; Bercher, P.; Schattenberg, B.; and Biundo, S. 2014. Introducing hierarchy to non-hierarchical planning models - a case study for behavioral adversary models. In *Proceedings of PuK 2014*.

Sacerdoti, E. D. 1974. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence* 5(2):115–135.

Schattenberg, B. 2009. *Hybrid Planning & Scheduling*. Ph.D. Dissertation, Ulm University, Germany.

The HTN Domain “Factories”

Malte Sönnichsen,[1] Dominik Schreiber[2]
Karlsruhe Institute of Technology
[1]malte@soennichsen.xyz [2]dominik.schreiber@kit.edu

Introduction

In this paper we present the benchmark domain “Factories” for Hierarchical Task Network (HTN) planning written in the HDDL format (Höller et al. 2020).

In this benchmark domain, the planning objective is to construct a *factory* by satisfying a dependency graph on certain *resources* while using *trucks* to deliver the needed resources from one place to another. As such, our domain is a crossover of traditional logistics and transport domains with a producer-consumer problem.

Overview

An input problem consists of f factories, d dependencies, and t trucks. Each factory *requires* certain resources in order to be built, which are the factory’s dependencies, and is able to *produce* certain resources when built. As such, the problem’s factories depend on each another: The construction of some factory F may require the construction of other factories which produce the resources necessary to construct F. This renders the problem recursive and makes the choice of a hierarchical planning model natural. The set of dependencies can be seen as a Directed Acyclic Graph (DAG) induced by the required and produced resources of each factory.

Transporting resources from a producer to a consumer requires trucks. There can be multiple trucks in the problem, so the tactical choice of which truck should do the delivery can lead to varying plan lengths.

In order to avoid complex conditions, we introduced composite resources which are fused from two other resources. This way, a factory has at most one resource demand in the HDDL while the number of needed actual resources can be arbitrarily high. Also, for the sake of simplicity, the resources a certain factory requires for its construction and for producing a resource itself are identical.

Generating Problems

Our problem generator script features three inputs which are (i) the number of factories, (ii) the maximum number of resource dependencies *per factory*, and (iii) the number of trucks. The graph of locations is generated as a simple undirected circle of n locations with up to n additional random edges, where n is set to twice the number of factories and

```
(:method m_factory_already_constructed
    :parameters (?f - factory
                 ?l - location)
    :task (construct_factory ?f ?l)
    :precondition (factory-at ?f ?l)
    :subtasks ()
)
(:method m_construct_factory
    :parameters (?f - factory
                 ?r - resource
                 ?l - location)
    :task (construct_factory ?f ?l)
    :precondition (and
        (demands ?f ?r)
        (location-free ?l)
        (not (factory-constructed ?f))
    )
    :ordered-subtasks (and
        (get_resource ?r ?l)
        (construct ?f ?r ?l)
    )
)
```

Figure 1: The two methods which decompose the initial task `(construct_factory ?f ?l)`.

trucks. Trucks are placed randomly in the graph. Dependencies are introduced to the problem by iterating over the factories “from left to right” and repeatedly adding dependencies of the current factory to a resource that some factory to the left produces. This ensures that the dependency graph is in fact acyclic and can be ordered. The first factory has no resources and the final factory is the destination object to be constructed.

Properties

The Factories domain is recursive, i.e., it may feature task T as a possible subtask in a method for task T. The domain is totally ordered.

The domain makes use of positive and negative method preconditions. All constraints are stated conjunctively (“STRIPS-style”), i.e., as a list of literals.

We think the Factories domain is an appealing bench-

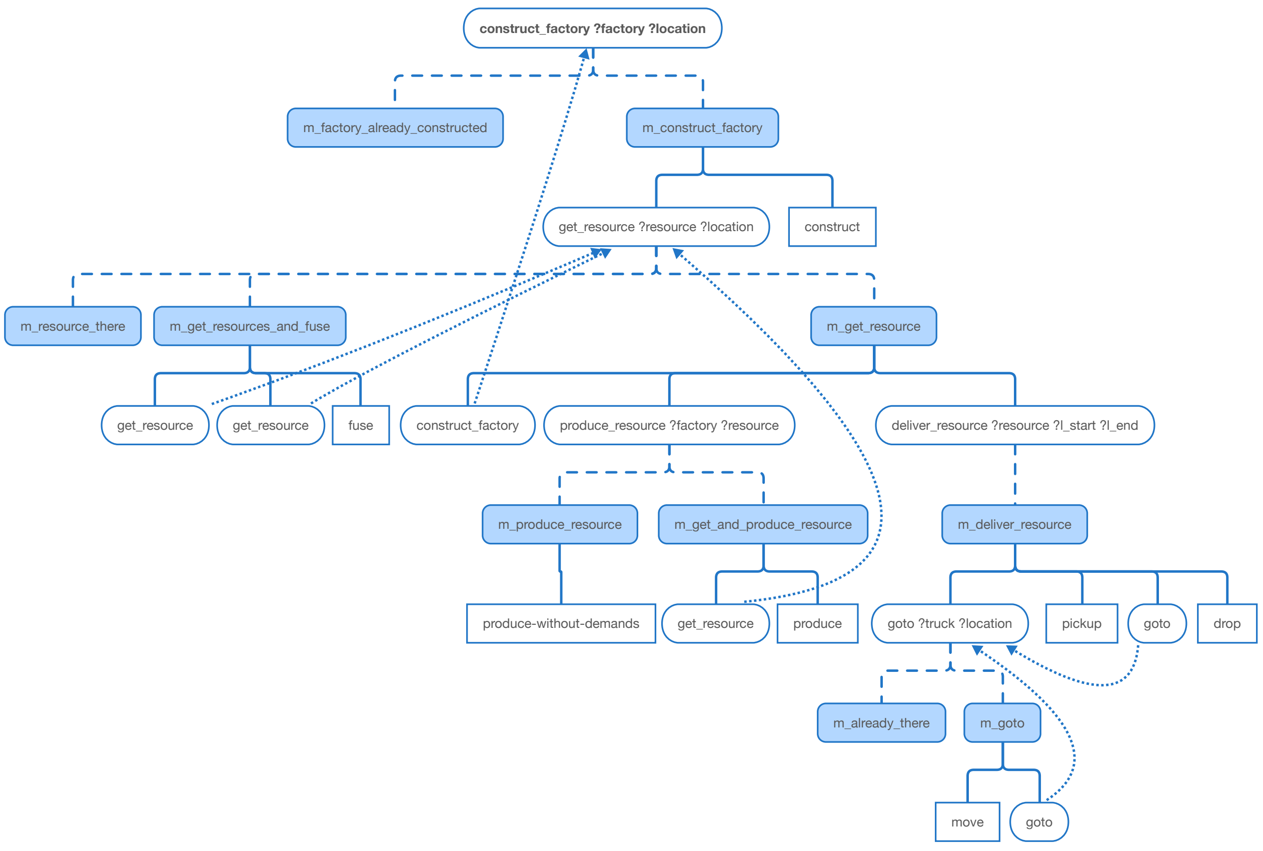

Figure 2: Illustration of the hierarchy of the Factories domain. Rectangles represent primitive tasks, circular containers represent composite tasks, and blue rectangles with rounded corners represent methods. Dashed lines denote the choice (OR) of a method, straight lines denote sequences (AND) of subtasks. Dotted arrows hint to recursive subtask relationships.

mark domain because it is a very natural application of hierarchical planning and because the problem size is easily scaleable up to dimensions which are very difficult for common HTN planners while the problem description stays relatively compact. A problem's properties can be steered using the three input parameters: Increasing the number of trucks makes grounding more difficult and produces higher numbers of methods per task. Conversely, increasing the number of factories increases the overall size of the task network, and increasing the number of dependencies makes the problem more "logically dense" while increasing the size of the task network to a moderate degree.

With our simple problem generation, the non-trivial decisions a planner has to make "only" involves the choice of which trucks should deliver which resources and which factory to place at which position; still, it is an interesting domain to benchmark planners on because it tests their ability to ground (if applicable) and search this quite straight-forward hierarchical structure as efficiently as possible. Moreover, the resulting plan length may vary considerably based on a planner's strategic decisions regarding truck assignments and factory placements, so we believe that the domain lends itself to evaluations concerning plan quality and plan optimization techniques.

Although heavily idealized, we imagine that our domain's structure can be adapted to a plethora of similar tasks that feature DAG-like dependencies.

Acknowledgments

Many thanks to Gregor Behnke for finding, and proposing fixes to, several bugs.

References

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. HDDL: An extension to PDDL for expressing hierarchical planning problems. In *AAAI*, 9883–9891.

The Smartphone Domain

Pascal Bercher[1,*] and **Susanne Biundo**[2] and **Bernd Schattenberg**[3,*]
[1] The Australian National University, Australia, pascal.bercher@anu.edu.au
[2] Institute of Artificial Intelligence, Ulm University, Germany, susanne.biundo@uni-ulm.de
[3] 3B intelligent solutions, Germany, schattenberg@3b-intelligent-solutions.com
[*] The domain was created while still being at the Institute of Artificial Intelligence of Ulm University, Germany
The domain was written by Bastian Seegebarth, formerly at Ulm University. Authors are ordered alphabetically.

Abstract

This extended abstract is about the Smartphone domain, submitted as a benchmark domain to the IPC 2020.

The Smartphone Model

Companion Technology (Biundo and Wendemuth 2016; 2017) enables every-day technical systems to become really user-friendly – those companion systems adapt their functionality to the individual user's current situation, emotional state, and needs. Companion Technology involves various scientific disciplines (Biundo et al. 2016), and AI planning plays a key role as it allows a goal-directed behavior of systems and provides many further user-centered technologies, such as plan explanations (Bercher et al. 2017).

In earlier work we described how AI planning can be used to enhance the operation of a Smartphone (Biundo et al. 2011). Fig. 1 illustrates some of the menus of the (actual) smartphone that was modeled. There, we already described excerpts of the model, such as parts of the sort and task hierarchy. The actual working model was, however, created later on by our former colleague Bastian Seegebarth under the supervision of Bernd Schattenberg. The model allows to carry out various standard tasks done regularly, such as sending messages (via various means like SMS or email), attaching pictures, creating new and deleting contacts, etc.

The original model (also available in the respective repository) was written for a *hybrid* planning formalism (Biundo and Schattenberg 2001; Schattenberg 2009; Bercher et al. 2016), which fuses Hierarchical Task Network (HTN) planning with Partial Order Causal Link (POCL) planning. That model also used state abstraction axioms that define a hierarchy on state features, to be exploited for preconditions and effects of abstract tasks. For the submission to the IPC, all these "hybrid" features were stripped away, resulting into a pure HTN model. The model is cyclic and partially ordered.

The domain ended up not being selected for the IPC, because only seven problem instances were modeled, and no random generator for further instances was provided. All domains for the IPC feature 30 problem instances, but modeling further instances by hand turned out to be too hard due to the complex structure of the model.

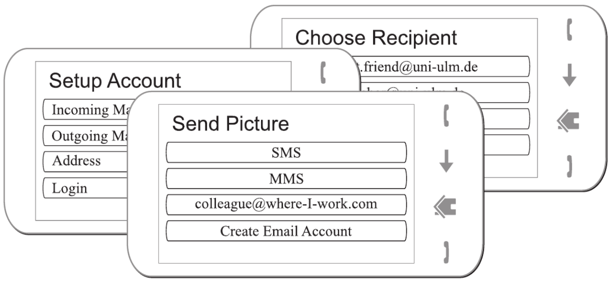

Figure 1: Illustration of the Smartphone that was modeled.

References

Bercher, P.; Höller, D.; Behnke, G.; and Biundo, S. 2016. More than a name? On implications of preconditions and effects of compound HTN planning tasks. In *ECAI*, 225–233. IOS Press.

Bercher, P.; Höller, D.; Behnke, G.; and Biundo, S. 2017. User-centered planning. In Biundo, S., and Wendemuth, A., eds., *Companion Technology – A Paradigm Shift in Human-Technology Interaction*, Cognitive Technologies. Springer. chapter 5, 79–100.

Biundo, S., and Schattenberg, B. 2001. From abstract crisis to concrete relief – a preliminary report on combining state abstraction and HTN planning. In *ECP*, 157–168. AAAI.

Biundo, S., and Wendemuth, A. 2016. *Companion-technology for cognitive technical systems. Künstliche Intelligenz* 30(1):71–75. doi: 10.1007/s13218-015-0414-8.

Biundo, S., and Wendemuth, A., eds. 2017. *Companion Technology – A Paradigm Shift in Human-Technology Interaction*. Cognitive Technologies. Springer.

Biundo, S.; Bercher, P.; Geier, T.; Müller, F.; and Schattenberg, B. 2011. Advanced user assistance based on AI planning. *Cognitive Systems Research* 12(3-4):219–236.

Biundo, S.; Höller, D.; Schattenberg, B.; and Bercher, P. 2016. Companion-technology: An overview. *Künstliche Intelligenz* 30(1):11–20.

Schattenberg, B. 2009. *Hybrid Planning & Scheduling*. Ph.D. Dissertation, Ulm University, Germany.