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^{max}_{add}; 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 ⟵ {top};
2 while OPEN not empty do
3     n ⟵ OPEN.pop();
4     if n.flaws ≠ ∅ then
5         return n.plan;      // solution found
6     f ⟵ n.flaws.pop();
7     for r ∈ resolvers(f, n.plan) do
8         OPEN ⟵ 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^{max}_{add} 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*.