Loop Detection in the PANDA Planning System

Daniel Höller,[1] **Gregor Behnke**[2]
[1]Saarland University, Saarland Informatics Campus, Saarbrücken, Germany,
[2]University of Freiburg, Freiburg, Germany,
hoeller@cs.uni-saarland.de, behnkeg@cs.uni-freiburg.de

Abstract

The International Planning Competition (IPC) in 2020 was the first one for a long time to host tracks on Hierarchical Task Network (HTN) planning. HYPERTENSION, the winner of the tack on totally-ordered problems, comes with an interesting technique: it stores parts of the decomposition path in the state to mark expanded tasks and forces its depth first search to leave recursive structures in the hierarchy. This can be seen as a form of loop detection (LD) – a technique that is not very common in HTN planning. This might be due to the spirit of encoding enough advice in the model to find plans (so that loop detection is simply not necessary), or because it becomes a computationally hard task in the general (i.e. partially-ordered) setting. We integrated several approximate and exact techniques for LD into the progression search of the HTN planner PANDA. We test our techniques on the benchmark set of the IPC 2020. Both in the partial ordered and total ordered track, PANDA with LD performs better than the respective winner of the competition.

Introduction

The International Planning Competition (IPC) in 2020 was the first for a long time to host tracks on Hierarchical Task Network (HTN) planning (Erol, Hendler, and Nau 1996; Bercher, Alford, and Höller 2019): one for the full formalism, and one where decomposition methods need to be totally-ordered (TO) HTN planning. HYPERTENSION (Magnaguagno, Meneguzzi, and de Silva 2021), the winner of the TO track, comes with an interesting technique: it memorizes some of the applied decompositions in newly added state features. Based on this book-keeping, HYPERTENSION forces its depth first search to leave recursive decomposition structures that otherwise may lead to infinite loops by forbidding decompositions of tasks that occur a second time during the decomposition process. This can be seen as a limited form of loop detection (LD). It is limited since it can only detect loops in a single decomposition path. A detection is impossible if two different paths lead to isomorphic task networks. Further, the specific technique used by HYPERTENSION renders its search incomplete (though this seems not to be a problem on the IPC benchmark set) as it does not consider changes in the original state variables.

Copyright © 2021, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

LD is not commonly used in HTN planning. This might be due to the spirit of encoding enough advice into the model to find plans (making LD unnecessary), or because it becomes a hard to solve task in the general, i.e. partially-ordered (PO), setting. For loop detection, it is not sufficient to only consider states, but one also has to take the tasks in the current task network and their ordering relations into account. This makes LD in general as hard as Graph Isomorphism (Behnke, Höller, and Biundo 2015).

We investigate the effect of loop detection on the heuristic progression search of the HTN planner PANDA (Höller et al. 2018, 2019, 2020; Höller, Bercher, and Behnke 2020). We propose several (exact and approximate) LD techniques and evaluate them on the benchmark sets of the IPC 2020. Our techniques increase the performance of PANDA in both tracks. In the PO setting, the base PANDA already solves more instances than the IPC's winner, but loop detection further increases coverage and IPC score. In the TO setting, the base PANDA performs worse than the runner-up of the IPC and better than the winner when using loop detection.

Formal Framework

We use the HTN formalism introduced by Geier and Bercher (2011). In HTN planning, we have two types of tasks. Let A be the set of actions (primitive tasks) and C the set of abstract (also called compound tasks). The environment is described using a set of propositional state symbols F. A state s is defined by a subset of F that holds in it (i.e. $s \in 2^F$). Propositions not in s are supposed to be false. The functions $prec$, add, and del map actions to their preconditions, add-, and delete-effects. All are defined as $f : A \to 2^F$. An action a is applicable in a state s if and only if $prec(a) \subseteq s$. The state s' resulting from the application of a is defined as $s' = \gamma(a, s) = (s \setminus del(a)) \cup add(a)$. A sequence of actions $a_1 a_2 \dots a_n$ is applicable in a state s_0 if for $1 \leq i \leq n$, a_i is applicable in s_{i-1} with $s_i = \gamma(a_i, s_{i-1})$.

Tasks are maintained in *Task Networks* (TNs), which are partially-ordered multi-sets of tasks. A TN is defined by a triple $(T, \alpha, \prec)$, where T is a set of task identifiers, α a function mapping the task ids to tasks $\alpha : T \to A \cup C$, and $\prec$ a partial order on T. Abstract tasks are decomposed using a set of *decomposition methods* M. A method is a pair (c, tn) where c is an abstract task that defines which task can be decomposed by the method, and tn a TN defining

its subtasks. When a method is applied to an abstract task c in a TN, c is deleted from the network, the subtasks of the method are added, and inherit the ordering relations of c with respect to other task in the network. A method $m = (c, tn)$ decomposes a TN $tn_1 = (T_1, \prec_1, \alpha_1)$ including a task $t \in T_1$ with $\alpha_1(t) = c$ into a TN tn_2 defined as follows. Let $tn' = (T', \prec', \alpha')$ be a TN that is equal to tn but using ids not contained in the decomposed network (i.e. $T_1 \cap T' = \emptyset$).

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

When tn_1 can be decomposed into tn_2 by using 0 or more (sequential) method applications, we write $tn_1 \rightarrow^* tn_2$.

An HTN planning problem is defined as $P = (F, C, A, M, s_0, tn_I, prec, add, del)$. The two elements s_0 and tn_I define the initial state (i.e. $s_0 \in 2^F$) and the initial TN. A solution to the problem is a TN $tn = (T, \alpha, \prec)$ with:

- $tn_I \rightarrow^* tn$, i.e. it can be obtained by decomposing the initial task network.
- $\forall t \in T : \alpha(t) \in A$, i.e. all tasks are primitive.
- There is a sequence $t_{i_1} t_{i_2} \dots t_{i_n}$ of all task identifiers in T that satisfies the ordering relation $\prec$ such that $\alpha(t_{i_1})\alpha(t_{i_2}) \dots \alpha(t_{i_n})$ is applicable in s_0.

An HTN planning problem is called *totally-ordered* if and only if the ordering relations of the subtasks of all methods and in tn_I are total, i.e. the orderings are linear paths.

There are mainly two search techniques in HTN planning. In progression search, only tasks of a TN without predecessors in the ordering relation are processed, i.e., decomposed in case of abstract tasks, or applied, in case of primitive tasks. Other systems are closer to the definition given above: search nodes contain a TN and a set of causal links and (additional) ordering relations. These systems search flaw-based like POCL planners in classical planning.

Loop Detection in HTN Planning

As given before, our aim is to combine LD with heuristic progression search. Here, search nodes are stored in a fringe during search. We additionally use a *visited list* $\mathcal{V}$ to keep track of already seen nodes. Before adding a node to the fringe, we check whether it has been seen before, i.e. is in $\mathcal{V}$, and if so discard it. In progression search, a search node is a pair (s, tn) of a state s and a task network tn. For testing whether $(s, tn) \in \mathcal{V}$, we have to compare states and TNs. We divide $\mathcal{V}$ into buckets $\mathcal{V}[s]$ s.t. all visited nodes with state s are in $\mathcal{V}[s]$. Access to the buckets is done via bit-wise `XOR` hashing on 64-bit unsigned integers followed by an exact comparison. We then have to decide for a given TN tn whether $tn \in V[s]$. Since TNs differ in their ids, we have to check whether there is a TN in $\mathcal{V}[s]$ that is isomorphic to tn.

Definition 1. Two TNs $tn_1 = (T_1, \prec_1, \alpha_1)$ and $tn_2 = (T_2, \prec_2, \alpha_2)$ are isomorphic if and only if there is a bijective function $f : T_1 \rightarrow T_2$ such that $\forall t, t' \in T_1 : (t, t') \in \prec_1 \Leftrightarrow (f(t), f(t')) \in \prec_2$ and $\alpha_1(t) = \alpha_2(f(t))$.

The *Task Network Isomorphism Problem* is to decide whether two given TNs are isomorphic.

This is only the simplest and weakest definition. It could be extend to *dominance*, where the solutions that can be reached from one TNs are a strict superset of the other TNs. Consider two TNs that (1) are equal despite the ordering relations and (2) the ordering relations of one TN is a subset of that of the others. We can safely prune the more constrained one as any solution is still reachable through the other one. Determining even this simple type of ordering dominance is NP-complete (Behnke, Höller, and Biundo 2015). We have thus not looked into dominance pruning. Even the strict isomorphism problem is already as hard as Graph Isomorphism (GI) (Behnke, Höller, and Biundo 2015). Furthermore, we do not only have to do the comparison between two TNs, but check for a given TN tn_1 whether there *exists* a TN tn_2 in $\mathcal{V}[s]$ such that tn_1 and tn_2 are isomorphic.

Given these difficulties, it is worthwhile to investigate (a) common special cases and (b) approximations for TN isomorphism. We consider only *over*approximations, i.e., techniques that always identify isomorphism, but may also state that two TNs are isomorphic when they in fact are not. Using such a LD leads to an incomplete search, but can pay off in practice – provided the problem remains solvable.

We first discuss techniques for TOHTN planning and come to PO afterwards. Each section starts with the *exact* isomorphism test and gives approximations afterwards. All presented approximations render the search incomplete, i.e. only the exact tests result in a complete overall planner. However, in our evaluation we also include combinations of the presented techniques. They first use the approximate test and perform the exact one when nodes are identified as identical by the approximation. This works well in practice (especially in the PO setting, where the exact test is costly) and does not render the search incomplete.

Total Order HTN Planning

Exact Isomorphism For TOHTN planning, the TN of every search node is a sequence of tasks $S = \langle t_1, \dots, t_n \rangle$. Given such a sequence S, we have to check whether $\mathcal{V}[s]$ contains S. Currently, we use the `set` from the C++ Standard Template Library. This way, exactly testing whether $S \in V[s]$ takes $\mathcal{O}(|S| \cdot \log |\mathcal{V}[s]|)$ time. By using a trie we could obtain $\mathcal{O}(|A \cup C| \cdot |S|)$, which can be reduced by hashing and/or a balanced tree per node of the trie to $\mathcal{O}(\log |A \cup C| \cdot \log |S|)$.

Task Hash With a growing number of search nodes, even this logarithmic test can become too expensive. To mitigate the issue, we have added a hashing step before testing whether $S \in \mathcal{V}[s]$, i.e. we again divide $\mathcal{V}[s]$ into hash buckets and test only the bucket into which S falls. Such a hashing function can also be used as an (over-)approximate isomorphism test, if we consider two TNs isomorphic if their hash is equal. Our first hashing method is called *taskhash*. It discards all ordering information from the TNs and computes a hash value that only incorporates information on which task is how often in a given TN. Let $N = C \cup A$. Further, let $t2n : N \rightarrow \{0, \dots, |N| - 1\}$ be a function

that maps each task to a number between 0 and the total number of tasks in the particular HTN problem minus one. Let $count : TN \times N \to \mathbb{N}_0$ with $count((T, \prec, \alpha), n) = |\{t \in T \mid \alpha(t) = n\}|$ be a function counting the occurrences of the task n in a given TN and $pnr : \mathbb{N}_0 \to \mathbb{P}$ a function that maps a number n to the n^{th} prime number. Lastly, let pl be a large prime number small enough to prevent overflows. We compute *taskhash* as follows:

function taskhash(tn)
$hash = 0$
for $n \in N$ *and* $0 \leq j < count(tn, n)$ **do**
  $hash = (hash + pnr(j|N| + t2n(n))) \mod pl$
return $hash$

Relaxation and soundness. The only information left is how many instances of which task are in the network. Any ordering information is discarded. This method returns the same hash for two isomorphic networks as they contain the same multiset of tasks. It might return the same hash for two nodes that are not isomorphic, but differ in their ordering relations.
Runtime. Our implementation incrementally tracks which tasks are in a TN and uses a list of predefined prime numbers. This makes it computable in linear time (with respect to the size of the TN). Once calculated, we only need to compare numbers, which can be done in constant time.

Partial Order HTN Planning

Exact Isomorphism For general HTNs, we eventually have to check the isomorphism of directed vertex-labeled graphs, which is GI-complete (Behnke, Höller, and Biundo 2015). We check this using exhaustive search for a bijection. We only try to match nodes in both TNs that do not have any unmatched predecessors. We group the next nodes to be matched into buckets of nodes labeled with the same task and process all currently matchable nodes at once.

There is an often occurring structure in HTN planning, which we can exploit. Often only the initial task network tn_I is partially-ordered while the task networks in all methods are totally-ordered. Further, tn_I is often totally *un*ordered, i.e. it does not contain any ordering constraints. We call such problems *parallel-sequences problems*. Although this structure is rather restrictive, solving parallel-sequences problems is already undecidable (Erol, Hendler, and Nau 1996), i.e. it already carries the core difficulty of HTN planning.

In parallel-sequences problems, any derivable task network tn_1, i.e. $tn_I \to^* tn_1$, has a very specific structure: it consists of a set of sequences of tasks which are fully parallel, i.e. there are no ordering constraints between them. We can represent such TNs as a set of sequences of tasks $S = \{\langle t_1^1 \dots t_1^{n_1} \rangle, \dots, \langle t_m^1 \dots t_m^{n_m} \rangle\}$. We can obtain a normal form of S by sorting these sequences lexicographically. We then use the same techniques as for totally-ordered problems – with the sole difference that we consider a sequence of sequences of tasks instead of *one* sequence.
Relaxation and soundness. Both variants of exact isomorphism checking do not make any relaxation and are sound.
Runtime. Checking isomorphism of general TNs can have up to an exponential runtime in the size of the TNs. Also, we have to test every TN under consideration separately as there is no criterion on which e.g. a binary search could be performed. For parallel-sequences problems we achieve the same runtime as for totally-ordered problems.

Task Hash As for TO problems, we use *taskhash* to speed-up or to approximate the exact isomorphism test.

Task Layers Since *taskhash* performs a very strong approximation (it ignores all ordering constraints), we present an approximation that takes ordering at least partially into account. We divide the tasks in a given TN into layers according to their ordering relations. The first layer contains tasks without predecessors in the ordering relations, the second layer those tasks that are ordered behind the tasks in the first layer, and so on. Let $tn = (T, \prec, \alpha)$ be a TN. Starting with $i = 0$ the layers are defined as follows:

$$T_i = \{t \in T \mid \neg\exists t' \in (T \setminus \bigcup_{j<i} T_j) \text{ with } (t', t) \in \prec\}$$
$$L_i = \{\alpha(t) \mid t \in T_i\}$$

We calculate the L_i sets for each TN and return a match when all layers are identical. L_is are stored as balanced tree.
Relaxation and soundness. We lose information on the exact ordering between the tasks in a given layer and the ones in the preceding and succeeding layers. We might also lose information on the exact number of instances of a task (when they are in the same layer). Since isomorphic TNs only differ in their ids, but not in α and the ordering relations used here, it will correctly return equality for isomorphic TNs, but might also return it for TNs that are not isomorphic. Consider e.g. tn_1 where a is ordered before b and c before d, and tn_2 where a is ordered before d and c before b.
Runtime. PANDA stores TNs as directed graph, so layers can be calculated in linear time. For comparing two TNs, all layers are compared, which can also be done in linear time (with respect to the size of the TNs). Overall access takes $\mathcal{O}(|tn| + |L| \cdot \log |\mathcal{V}[s]|)$.

Ordering Relations The last approximation considers ordering relations locally. Let $tn = (T, \prec, \alpha)$ be a TN. We compute the following:

$$OR = \{(\alpha(t), \alpha(t')) \mid (t, t') \in \prec\}$$

We return a match when it is equal for two TNs.
Relaxation and soundness. This technique cannot distinguish multiple instances of the same task. For isomorphic TNs, it returns the same result since they only differ in their id set. It may, however, also return the same result when the TNs differ: consider a TN with a chain a before b and b before c, and a single extra task b. The result will be the same as for a before b, and the second instance of b before c.
Runtime. Given the TN representation of PANDA, the set can be computed in linear time in $|tn|$. Using balanced trees we get $\mathcal{O}(|tn| + |OR| \cdot \log |\mathcal{V}[s]|)$ overall.

	GBFS Add th	GBFS Add s	GBFS Add th+s	GBFS FF th	GBFS FF th+s	GBFS FF s	PANDASAT	TOAD	HYPERTENSION	GBFS LM-Cut s	GBFS LM-Cut th+s	GBFS LM-Cut th	LILOTANE	GBFS Add	GBFS FF	GBFS LM-Cut	PDDL4J	SIADEX	PYHIPOP	false positives GBFS Add th	% duplicates GBFS Add	% time for LD GBFS Add th
Assembly Hier. (30)	**30**	**30**	**30**	**30**	**30**	**30**	6	**30**	3	7	7	7	5	**30**	11	6	2	0	1	0.00	30.62	2.91
Barman-BDI (20)	17	16	16	**20**	**20**	**20**	19	16	**20**	15	15	15	17	13	**20**	13	11	**20**	0	0.01	82.44	1.16
BW GTOHP (30)	28	**29**	**29**	28	**29**	**29**	25	23	16	25	25	25	23	**29**	**29**	24	16	13	1	0.00	2.94	2.34
BW HPDDL (30)	28	28	28	25	25	25	6	21	**30**	14	14	14	1	0	0	0	0	0	0	0.00	99.99	1.65
Childsnack (30)	23	23	23	21	21	21	23	24	**30**	20	20	20	28	23	21	19	21	22	0	0.01	32.39	0.44
Depots (30)	22	22	22	27	27	27	**28**	24	24	24	24	24	24	22	27	24	23	22	0	0.06	28.50	4.79
Elevator L (147)	146	**147**	**147**	146	**147**	**147**	**147**	**147**	**147**	**147**	**147**	146	**147**	2	2	2	2	11	2	0.00	98.62	0.56
Entertainment (12)	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**	8	**12**	**12**	**12**	4	**12**	**12**	**12**	4	0	1	0.00	14.42	18.50
Factories (20)	**9**	**9**	8	7	7	7	8	5	3	5	5	5	4	1	0	0	0	0	1	0.00	95.13	1.54
Freecell L (60)	18	16	15	**19**	**19**	16	10	0	0	0	0	0	12	0	0	0	0	0	0	0.25	63.00	0.05
Hiking (30)	**25**	**25**	**25**	**25**	**25**	**25**	22	23	**25**	20	20	20	23	24	23	7	17	0	0	0.00	87.32	0.25
Logistics L (80)	48	48	48	49	49	48	**80**	49	22	79	79	75	45	0	1	1	0	0	0	0.00	100.00	1.40
Minecraft Pl. (20)	4	4	4	4	4	4	4	1	**5**	1	1	4	1	4	4	1	1	3	0	0.00	0.00	0.04
Minecraft Reg. (59)	43	43	43	43	43	44	40	41	**58**	41	41	41	37	44	43	41	23	35	0	29.33	4.88	2.88
Monroe FO (20)	18	**20**	**20**	18	**20**	**20**	**20**	0	**20**	14	13	12	**20**	**20**	**20**	13	**20**	7	0	1.19	9.32	0.08
Monroe PO (20)	8	10	10	12	12	12	**20**	0	0	9	9	8	**20**	11	12	9	1	0	0	0.44	25.68	0.13
Multiarm BW (74)	72	**74**	**74**	27	27	27	19	**74**	8	17	17	17	4	29	4	3	0	1	0	0.00	62.63	1.02
Robot (20)	**20**	**20**	**20**	**20**	**20**	**20**	11	**20**	**20**	19	19	19	11	1	2	1	6	0	1	0.00	95.00	15.15
Rover (30)	26	26	26	22	21	21	24	9	**30**	18	18	18	23	27	22	18	**30**	**30**	6	0.01	3.11	0.74
Satellite (20)	**20**	**20**	**20**	17	17	17	**20**	10	**20**	12	12	12	15	**20**	17	12	**20**	0	7	0.00	0.00	0.62
Snake (20)	**20**	**20**	**20**	**20**	**20**	**20**	**20**	15	**20**	**20**	**20**	**20**	**20**	**20**	18	19	**20**	7	2	0.00	3.10	0.23
Towers (20)	13	13	13	13	13	13	8	10	**16**	13	13	13	9	13	13	13	15	11	2	0.00	0.00	5.47
Transport (40)	32	25	25	30	21	21	**40**	34	**40**	19	19	24	34	24	20	19	34	1	18	37.70	36.13	5.39
Woodworking (30)	28	27	27	28	29	29	28	**30**	7	19	19	19	**30**	17	19	17	6	3	4	0.91	61.66	17.46
Instances: 892	**710**	707	705	663	658	655	640	618	572	570	569	567	560	386	340	274	272	186	46	3.35	57.33	2.63
Normal. Coverage	**18.8**	**18.8**	**18.8**	18.2	18.1	18.1	17.0	15.1	15.7	14.3	15.7	14.6	14.3	14.9	12.7	10.1	10.1	6.1	1.6			
IPC Score	**15.0**	**15.0**	**15.0**	14.4	14.4	14.4	13.7	11.9	**15.0**	10.3	10.3	10.3	12.5	11.0	10.1	7.4	7.6	4.9	0.9			

Table 1: Coverage Table – Total Order Track.

	GA* FF th+gi	GA* FF th	GA* FF oh	GA* FF oh+gi	GA* Add th+gi	GA* FF	GA* Add th	GA* Add oh+gi	GA* Add oh	GA* LM-Cut th	GA* Add	GA* LM-Cut th+gi	PANDASAT	GA* LM-Cut oh+gi	GA* LM-Cut	GA* LM-Cut oh	SIADEX	PYHIPOP	false positives GA* FF th	% duplicates GA* FF	% time for LD GA* FF th
Barman (20)	3	4	2	2	3	1	3	2	2	2	1	2	8	1	1	1	**20**	0	50.47	40.43	2.51
Monroe FO (25)	24	24	24	24	**25**	24	24	**25**	**25**	17	**25**	15	0	15	16	15	8	0	2.77	6.64	0.10
Monroe PO (25)	19	**20**	19	18	18	18	19	18	18	14	18	14	0	14	13	13	2	0	0.47	19.86	0.11
PCP (17)	**14**	6	**14**	**14**	**14**	**14**	6	**14**	**14**	7	**14**	**14**	13	**14**	**14**	**14**	0	0	54.02	0.09	20.26
Rover (20)	11	12	11	11	7	10	10	7	7	9	6	7	**16**	6	7	6	14	2	19.13	56.04	4.58
Satellite (25)	**25**	**25**	**25**	**25**	24	**25**	24	24	24	**25**	24	23	**25**	23	23	23	**25**	6	6.17	11.57	49.03
Transport (40)	12	16	12	12	8	11	15	6	7	17	4	15	**23**	15	13	15	1	3	46.12	47.46	7.54
UM-Translog (22)	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	21	0.00	0.56	86.73
Woodworking (30)	12	13	12	12	14	9	11	14	12	12	11	12	**17**	11	10	10	3	6	25.88	27.51	15.47
Instances: 224	**142**	**142**	141	140	135	134	134	132	131	125	125	124	124	121	119	119	95	38	23.07	25.08	19.60
Normal. Coverage	**5.9**	5.7	**5.9**	**5.9**	5.7	5.6	5.4	5.6	5.5	5.0	5.3	5.1	5.1	5.0	5.0	4.9	4.2	1.6			
IPC Score	**4.9**	4.5	**4.9**	4.8	4.6	4.6	4.3	4.5	4.5	3.8	4.3	4.1	3.1	4.1	4.0	4.0	4.0	1.2			

Table 2: Coverage Table – Partial Order Track.

Empirical Evaluation

We implemented our techniques in the PANDA system (Höller et al. 2021)[1] and combined it with PANDA's progression search and *Relaxed Composition* (RC) heuristics (Höller et al. 2018) that internally use classical heuristics to guide the HTN search. We use the Add (Bonet and Geffner 2001), FF (Hoffmann and Nebel 2001) and LM-Cut (Helmert and Domshlak 2009) heuristics. The search is based on a ground model, which is different to several systems from the IPC that do not ground the model. We use the grounding procedure presented by Behnke et al. (2020).

We tested against PANDA without our techniques and against the participants of the IPC 2020, i.e., HYPERTENSION (Magnaguagno, Meneguzzi, and de Silva 2021), LILOTANE (Schreiber 2021a,b), SIADEX (Fernandez-Olivares, Vellido, and Castillo 2021), PDDL4J (Pellier and Fiorino 2021), PYHIPOP (Lesire and Albore 2021). We further included the TOAD system (Höller 2021), and PANDASAT in its current TO (Behnke, Höller, and Biundo 2018; Behnke 2021) and PO versions (Behnke, Höller, and Biundo 2019). We have updated the IPC planners to their newest versions, some of which fixed bugs in the original planners. We used the IPC 2020 benchmark set.

Experiments ran on Xeon Gold 6242 CPUs using 1 core, 8 GB of memory, and 30 minutes time limit.

In the following, we compare coverage and give certain metrics for our techniques like time needed for loop detection or percentage of duplicate search nodes. A more extensive evaluation can be found in a technical report (Höller and Behnke 2021). For the TO track, greedy best first (GBFS) worked best for PANDA, while in the PO track, weighted A* had the highest coverage. We only report these for PANDA.

Table 1 shows the results for the totally-ordered track. It first gives the absolute coverage of the systems. Configurations marked with a *th* use the *taskhash* test, those marked with *s* use only the exact test for totally-ordered problems, those marked with *th+s* first hash with *taskhash* and then use the exact test on all TNs in the hash bucket.

All GBFS configurations except for those using LM-Cut have a higher coverage than the other systems, while the GBFS LM-Cut configurations are placed between the IPC 2020 winner HYPERTENSION and the runner-up LILOTANE. The best configurations use the RC-Add heuristic, followed by RC-FF and RC-LM-Cut. The configurations not using LD have lower coverage than the IPC systems. The best of our configurations beats the respective PANDA baseline by 84% for RC-Add, 95% for RC-FF, and 108% for RC-LM-Cut. For the best configuration, the table gives on the right the number of false hits of the *taskhash* function, i.e., the number of nodes identified as visited by *taskhash* that in fact where no duplicates. There are mainly two domains where this happens. Averaged over all algorithms and heuristics we have 3.33% false hits. This is also relatively stable over search algorithms and heuristics. The second additional column shows the following: based on search without duplicate pruning we calculated the percentage of visited nodes that would have been pruned when using it. Here the results differ widely: between 0% (in the Satellite and Towers domains) and 100% (after rounding; in the Logistics domain) of the search nodes are duplicates. In the median instance 87.56% of search nodes are duplicates. The last column gives the percentage of time spend for LD using GBFS RC-Add with only *taskhash*.

Table 2 summarizes the results for the PO track. Configurations marked with *th* only use the *taskhash* function, those marked *oh* use the *taskhash* followed by *Task Layers* and *Ordering Relations* (denoted ordering hashes). Computing both of the latter two functions together does not significantly increase the computational effort compared to computing one of them, so we decided to always use their combination. When *+gi* follows, the exact test is performed in case of a hash collision. We applied the special handling for parallel-sequences instances whenever possible. These are all instances except for the 50 instances of the two Monroe domains and one instance of UM-Translog.

In the PO track, PANDA has a higher coverage than the IPC systems also without our techniques. However, they increase the coverage (though less significant than in the TO setting, between 5% and 8%). The configurations using the FF heuristic perform best. The number of false positive hits of the *taskhash* function is much higher than in the TO setting. When we look at the specific domains, we see that the approximate *taskhash* function loses coverage in a single domain (both compared to the exact calculation and to the basic PANDA configuration): PCP. It encodes *Post Correspondence Problem*, where the relaxation of the ordering relations seems to be especially harmful. In the other domains, it performs at least as good as the other methods.

Conclusion

We have introduced several techniques for loop detection in HTN planning. The problem boils down to deciding whether two TNs are isomorphic. We introduced exact and approximate techniques that overapproximate isomorphism, which might lead to an incomplete search but works very well in practice. We integrated our techniques into the heuristic progression search of the PANDA framework and evaluated them on the benchmark sets of the IPC 2020. They pay off in both the partially-ordered and the totally-ordered setting (though the gain in performance is much higher in the latter).

Our techniques are directly applicable in systems using a ground progression search. For POCL-like algorithms, the basic task in LD is still the comparison of task networks, so our techniques might be helpful. However, search nodes additionally include causal links. When these links shall be taken into account, this imposes further complexity and the adaptation of our techniques is not a straightforward task.

Acknowledgments

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

[1] The source code is available at *panda.hierarchical-task.net*

References

Behnke, G. 2021. Block Compression and Invariant Pruning for SAT-based Totally-Ordered HTN Planning. In *Proceedings of the 31st International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI Press.

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

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

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

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

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

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

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

Fernandez-Olivares, J.; Vellido, I.; and Castillo, L. 2021. Addressing HTN Planning with Blind Depth First Search. In *10th International Planning Competition: Planner and Domain Abstracts (IPC)*.

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. IJCAI/AAAI.

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

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

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

Höller, D.; and Behnke, G. 2021. Loop Detection in the PANDA Planning System: Extended Data. Technical Report 296, University of Freiburg, Department of Computer Science.

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

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

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

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2019. On Guiding Search in HTN Planning with Classical Planning Heuristics. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI)*, 6171–6175. IJCAI.

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

Lesire, C.; and Albore, A. 2021. PYHIPOP – Hierarchical Partial-Order Planner. In *10th International Planning Competition: Planner and Domain Abstracts (IPC)*.

Magnaguagno, M. C.; Meneguzzi, F.; and de Silva, L. 2021. HyperTensioN – A three-stage compiler for planning. In *10th International Planning Competition: Planner and Domain Abstracts (IPC)*.

Pellier, D.; and Fiorino, H. 2021. Totally and Partially Ordered Hierarchical Planners in PDDL4J Library. In *10th International Planning Competition: Planner and Domain Abstracts (IPC)*.

Schreiber, D. 2021a. Lifted Logic for Task Networks: TOHTN Planner Lilotane in the IPC 2020. In *10th International Planning Competition: Planner and Domain Abstracts (IPC)*.

Schreiber, D. 2021b. Lilotane: A Lifted SAT-based Approach to Hierarchical Planning. *Journal of Artificial Intelligence Research* 70: 1117–1181.