On Succinct Groundings of HTN Planning Problems

Gregor Behnke, Daniel Höller, Alexander Schmid, Pascal Bercher,[*] and **Susanne Biundo**
Institute of Artificial Intelligence, Ulm University, D-89069 Ulm, Germany
{gregor.behnke, daniel.hoeller, alexander-1.schmid,susanne.biundo}@uni-ulm.de, pascal.bercher@alumni.uni-ulm.de

Abstract

Both search-based and translation-based planning systems usually operate on grounded representations of the problem. Planning models, however, are commonly defined using lifted description languages. Thus, planning systems usually generate a grounded representation of the lifted model as a preprocessing step. For HTN planning models, only one method to ground lifted models has been published so far. In this paper we present a new approach for grounding HTN planning problems that produces smaller groundings in a shorter timespan than the previously published method.

Introduction

Most modelling languages for planning problems (such as PDDL (McDermott 2000) and HDDL (Höller et al. 2020)) allow for specifying planning problems in a *lifted* fashion, e.g. by allowing the modeller to specify the model using a first-order logical language. Actions are specified with parameters and the action's preconditions and effects are specified using literals referring to these parameters. Using such a lifted representation, a modeller can easily write models with a large number of action instantiations without the need to enumerate them explicitly. More importantly, a lifted representation enables the modeller to specify a single planning domain that can be used in multiple planning problems without any change to the domain.

Unfortunately, to plan directly using only the lifted model is quite difficult. Most planners transform the lifted input-model into a grounded model before planning. Planning is then performed on the grounded representation, either via search (Höller et al. 2018b; Bercher et al. 2017; Helmert 2006) or via a translation into other problems, e.g. SAT (Behnke, Höller, and Biundo 2019b; 2019a; 2018b; Behnke and Biundo 2018; Behnke, Höller, and Biundo 2018a; Rintanen 2014; Rintanen, Heljanko, and Niemelä 2006). Naively grounding the lifted representation by simply instantiating all its elements is seldom feasible due to the huge size of the naively grounded model. Instead, the grounding procedure aims to remove as many unnecessary instantiations as possible. Unnecessary here means that grounding of e.g. actions can be removed if they cannot be part of a solution to the planning problem. Smaller groundings are generally advantageous to planners, as their per-search-node effort or the size of any translation decreases. Further, even the quality of computed heuristics can improve if "distractor actions" are removed. Even a small decrease in the size of the grounding can significantly impact the efficiency of the planner. As such, grounding is a critical step in the process of planning.

For Hierarchical Task Network (HTN) planning (Erol, Hendler, and Nau 1996), there is – as far as we know – only a single paper concerned with grounding HTN planning domains (Ramoul et al. 2017), which is used in the planner GTOHP. However, several other HTN planners plan using a grounded model (e.g. FAPE (Dvorak et al. 2014) and PANDA (Bercher, Keen, and Biundo 2014; Bercher et al. 2017)), but there are no publications on their grounding procedures. Lastly, Tree-Rex (Schreiber et al. 2019) uses a yet unpublished improvement of GTOHP's grounder.

In this paper we describe the grounding procedure that as so-far been used by the HTN planner PANDA and a recently developed improved version of the grounder called pandaPI. We start by describing the lifted HTN planning formalism, then give an overview of grounding in planning in general and in HTN planning in particular. We describe the grounding procedure used by PANDA. Thereafter we present the new and more systematic view on the HTN grounding process, which allowed us to create the new and highly efficient grounder pandaPI. Lastly, we compare the performance of pandaPI with that of PANDA, GTOHP, and Tree-Rex.

Lifted HTN Planning Formalism

Before explaining the HTN grounding procedure, we start by briefly describing the formalism of lifted HTN planning. We have based our formalism on the lifted one by Alford, Bercher, and Aha (2015), which in turn is based on the formalism by Geier and Bercher (2011).

Assume that $\mathcal{L} = (P, T, V, C)$ is a quantifier- and function-free first-order predicate logic with the following elements. P is a finite set of *predicate symbols*. A predi-

[*]Pascal Bercher is now at the College of Engineering and Computer Science, the Australian National University

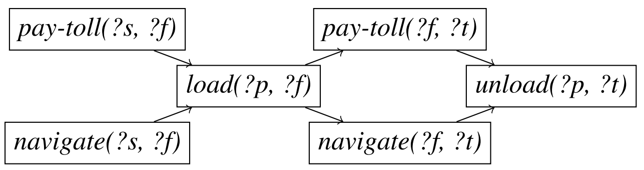


Figure 1: A task network in a transportation domain. If performed, it will transport a package from its initial location to its target location. The variables *?s* (start location), *?f* (initial package location), and *?t* (target package location) are of type location. The variable *?p* is of type package. Parallelism between the pay-toll and navigate tasks models that the toll can be paid at any time while the transporter is on its way from the one location to another.

cate's arity defines its number of parameter variables (taken from V), each having a certain type (defined in T). T is a finite set of *type symbols*. V is a finite set of typed variable symbols to be used by the parameters of the predicates in P. C is a finite set of typed constants.

HTN planning problems specify two types of tasks: *Primitive tasks* are identical to actions in classical planning and are identified via a first-order atom $action(?v_1, \ldots, ?v_n)$[1] (e.g. *drive(?f, ?t)*). Their semantics is given via their preconditions *pre* – a conjunction of positive first-order literals over $\mathcal{L}$ – and effects *eff* – a conjunction of (positive and negative) first-order literals[2]. The variables occurring in *pre* and *eff* must be parameters, i.e. members of $\{?v_1, \ldots, ?v_n\}$. Applicability and the state transition semantics of actions is defined as in classical planning.

Abstract tasks are also described via first-order atoms $task(?v_1, \ldots, ?v_n)$. Their semantics is given in terms of pre-defined means for performing them, which are described by *decomposition methods* M. A decomposition method $m \in M$ is a tuple (c, tn) consisting of an abstract task c and a task network tn. A task network is a partially ordered multi-set of actions and tasks.

Definition 1. *A task network* tn *over a set of primitive and abstract* tasks X *(first-order atoms) is a tuple* $(I, \prec, \alpha)$ *with:*
1. I *is a finite set of* task IDs.
2. $\prec$ *is a strict partial order over* I*.*
3. $\alpha : I \to X \times \overline{V}$ *maps task IDs to task and parameters*

Note that pandaPI does also allow for specifying *variable constraints*, i.e. equals and not-equals constraints on variables and constants. We handle them naively by checking them and do not discuss them further in the paper for the sake of brevity. As an example for a task network consider the one shown in Fig. 1.

An HTN planning problem is then given by the problem's *initial state* s_I (a set of ground positive literals or facts), a set of available decomposition methods M, and the *initial abstract task* c_I which specifies the goal.

[1]We adopt the convention of PDDL (McDermott 2000) to denote variables with a prefixed question mark. I.e. $?a$ is a variable, while a denotes a constant.

[2]More complex representations are common in domain specification languages, but can be compiled into this simplistic format.

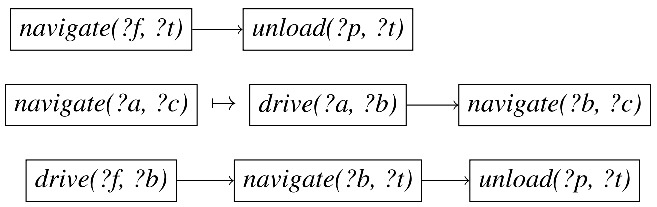


Figure 2: The first row shows a task network tn_1. The second row shows a method for the navigate task. The result of applying this method to the navigate task in tn_1 results in the task network shown in the third row.

The aim in an HTN planning problem is to refine a given initial abstract task c_I into an executable, ground, primitive task network. A task network is primitive if all tasks in it are primitive. It is ground if all variables are assigned to constants. It is further executable if there is a linearisation of its tasks that is executable in the initial state. Refining the initial task network is performed via applying decomposition methods to the abstract tasks contained in it and the resulting task networks. Applying a decomposition method (c, tn_c) to a task network tn means to replace an occurrence of the task c in tn by the contents of the task network tn_c.

Definition 2. *Let* $m = (c(?x_1, \ldots, ?x_n), tn_m)$ *with* $tn_m = (I_m, \prec_m, \alpha_m))$ *be a decomposition method, and* $tn_1 = (I_1, \prec_1, \alpha_1)$ *a task network. We assume that* $I_m \cap I_1 = \emptyset$ *and that the sets of variables occurring in* tn_1 *and* tn_m *are disjunct, which can be achieved by renaming. Then,* m *decomposes a task identifier* $i \in I_1$ *into a task network* $tn_2 = (I_2, \prec_2, \alpha_2)$ *iff* $\alpha_1(i) = c(?y_1, \ldots, ?y_n)$ *and*

$$\begin{aligned}
I_2 &= (I_1 \setminus \{i\}) \cup I_m \\
\prec_2 &= (\prec_1 \cup \prec_m \cup \\
&\quad \{(i_1, i_2) \in I_1 \times I_m \mid (i_1, i) \in \prec_1\} \cup \\
&\quad \{(i_1, i_2) \in I_m \times I_1 \mid (i, i_2) \in \prec_1\}) \\
&\quad \setminus \{(i', i'') \in I_1 \times I_1 \mid i' = i \text{ or } i'' = i\} \\
\alpha_2 &= \alpha_1 \cup \setminus \{(i, c(?y_1, \ldots, ?y_n))\} \cup \\
&\quad \{(i, c(?z_1^*, \ldots, ?z_m^*)) \mid (i, c(?z_1, \ldots, ?z_m)) \in \alpha_m, \\
&\quad\quad \forall j : ?z_j^* = ?y_k \text{ iff } ?z_j = ?x_k, \text{ and } ?z_j^* = ?z_j \text{ else}\}
\end{aligned}$$

As an example for applying a decomposition method, consider the task networks and the method shown in Fig. 2.

Grounding Planning Problems

Both theoretical research (Alford et al. 2016; Behnke et al. 2016; Bercher et al. 2016; Höller et al. 2016; Behnke, Höller, and Biundo 2015; Höller et al. 2014) and practical research (Schreiber et al. 2019; Höller et al. 2018b; Behnke et al. 2019) on hierarchical planning is usually done on *grounded*, i.e. variable-free models, instead of lifted models. Especially newer search-based HTN planners like FAPE (Dvorak et al. 2014), GTOHP (Ramoul et al. 2017), or PANDA (Bercher, Keen, and Biundo 2014; Bercher et al. 2017) ground a given lifted planning problem prior to search. A grounded model allows for both a more efficient implementation and for easier to compute and more concise heuristics. In contrast, the translation technique by Alford et al. (2016) is executed on the lifted model – grounding is only performed on the resulting classical model.

In theory, computing a grounded model based on a given lifted model is easy. One has to compute all possible instantiations of lifted predicates, primitive actions, abstract tasks, and methods and replace their lifted versions appropriately by them. For details regarding this full grounding process we refer to Alford, Bercher, and Aha (2015). Such a grounding will be exponential in size. Thus, a fully grounded model is not useful in many practical cases, as handling it within given memory and time limits is hard or even impossible.

In many planning problems, computing all instantiations of all predicates, tasks, and methods is not necessary. For example, it is not necessary to create a grounding $drive(l_1, l_2)$ of the drive action if there is no road between the locations l_1 and l_2. For such an instantiation $drive(l_1, l_2)$, we know a priori that its precondition can never be fulfilled[3]. Thus this action cannot be part of any plan. Ideally, we would like to compute only those groundings of predicates, tasks, and methods that occur in some solution to the planning problem. Determining whether this is the case is undecidable.

Theorem 1. *The problem of deciding whether an action a is part of some plan for a given HTN planning problem is undecidable.*

Proof. We can show the claim via a reduction from the plan existence problem for HTN planning problems, which is known to be undecidable (Erol, Hendler, and Nau 1996; Geier and Bercher 2011). Let $\mathcal{P}$ be an HTN planning problem. We add a primitive action a without any precondition and effects. Further, we add a new initial abstract task c_I^*, which has only a single decomposition method yielding one instance of a and the old initial abstract task c_I of $\mathcal{P}$ without any ordering constraint. Now, $\mathcal{P}$ has a solution if and only if a is part of some plan for $\mathcal{P}$. □

Instead, we aim at computing an approximation of this property. We are looking for a subset of all ground instances of predicates, tasks, and methods such that all ground instances not included in that set are not contained in any solution. I.e., we disregard a ground instance if we can prove that it cannot be contained in a solution. This approximation entails a trade-off: with higher computation time, a better approximation might be found.

This technique of approximate grounding is widely used in classical planning. In general, an action is not included in the grounding if it cannot be part of any executable plan in the *delete-relaxation* of the problem. The delete-relaxation of a planning problem is a copy of the problem in which all negative effect literals are removed. Any action that is not part of a plan in a delete-relaxed problem cannot be part of a plan in the original problem. For a given action one can determine in polynomial time whether it is part of any delete-relaxed classical plan (Bylander 1994). The set of these actions is usually computed via a *planning graph* (Blum and Furst 1997). Often, this reduction leads to a significant decrease in the size of the grounded problem. Some planning systems, like FF (Hoffmann and Nebel 2001), first compute the full grounding and subsequently prune actions[4]. This, however, does not eliminate the bottle-neck of grounding, but makes the grounding smaller for the planning process itself. An efficient implementation based on DATALOG was proposed by Helmert (2009), which does not have this bottle-neck of a full instantiation.

To the best of our knowledge there is currently only one publication in the field of HTN planning devoted to grounding, which is the grounder of GTOHP (Ramoul et al. 2017). It uses a grounding procedure similar to that of FF – i.e. it computes a full instantiaton and prunes subsequently – and similarly uses the concept of inertia to prune tasks early during instantiation (Koehler and Hoffmann 2000). Inertia of a predicate describe the ways its truth value can change while a plan is executed – not at all, only from negative to positive (or vice versa), or in both directions. In inertia-based simplification, a primitive task whose precondition evaluates to false under the computed inertia values is removed from the planning problem, as it can never become executable. Subsequently, all methods it is contained in are removed as well. If an abstract task has no applicable method remaining it is likewise removed.

The recently published planner Tree-Rex (Schreiber et al. 2019) uses GTOHP's grounding procedure but also prunes abstract tasks that cannot be refined into primitive actions any more – even though they have applicable decomposition methods. This leads to generally smaller groundings.

Note that GTOHP removes effectless actions from the methods they are contained in (Ramoul et al. 2017). The respective methods are not pruned afterwards, but considered part of the correct grounding without the removed effectless actions. According to the formalisation of HTN planning, these actions can however be contained in plans – and pose constraints in them. As such it makes any found solution (potentially) invalid as it may not adhere to the solution criteria of HTN planning. Secondly, the implementation of GTOHP does not allow for two parameters of an action or method to be instantiated with the same constant. Consider as an example a method that paints two wooden boards $?b_1$ and $?b_2$ in colours $?c_1$ and $?c_2$. GTOHP enforces that $?c_1$ and $?c_2$ are different without this constraint being a part of the domain. This leads to an incomplete grounding, this time when the (only) solution uses the method where both colours are, e.g. red, as we only have red paint. For our evaluation, we have fixed both issues in the code of GTOHP.

PANDA's Grounding

The HTN planning system PANDA has used a grounded representation for planning since 2014. All techniques that were published and evaluated using PANDA (e.g. (Behnke, Höller, and Biundo 2019a; 2019b; Höller et al. 2018b; Höller et al. 2018a; 2018c; Bercher et al. 2017; Behnke, Höller, and Biundo 2017; Bercher, Keen, and Biundo 2014)) based upon a grounded model, while the benchmark in-

[3]Assuming that there is no means to build new roads.

[4]Note that FF uses the concept of inertia (Koehler and Hoffmann 2000) to simplify the preconditions and effects before full grounding.

stances used in evaluations of PANDA are defined using the lifted language HDDL (Höller et al. 2020). Most of the techniques developed for PANDA (and pandaPI) originate in the need to solve specific planning domains. Notably, the need to handle the Monroe domain (Blaylock and Allen 2005), which is plan recognition domain, lead to the development of most of PANDA's grounding algorithm.

So far, we have not published any description of PANDA's grounding process. We correct this shortcoming with this paper and describe how the grounding procedure that has been used in the previous evaluations of PANDA works. In this section, we will give a short overview of the grounding techniques used by PANDA. The grounding procedure used by PANDA still had its shortcomings and problems. Thus, we have developed a new grounder: pandaPI. It is based on a new and more generic view on PANDA's grounding procedure. This enables us to significantly improve the performance compared PANDA's grounder.

PANDA's grounding procedure comprises three steps: a lifted domain simplification, a delete-relaxed reachability analysis, and a hierarchical reachability analysis based on the Task Decomposition Graph (TDG) (Elkawkagy et al. 2012; Bercher et al. 2017). The second and third step are executed once on the lifted input domain in order to ground the domain. Thereafter they are repeated in order to reduce the size of the grounding even further. Algorithm 1 shows the overall mechanism of PANDA's grounder. Iterating the reachability analysis is one of the major features of PANDA's grounder, as it enables finding smaller and more concise groundings. Note that this is also the main difference between the grounding results found by PANDA and Tree-Rex's improved grounding (Schreiber et al. 2019). It is based on the following observation: The TDG-based analysis can remove groundings of primitive actions, as they are not reachable through the hierarchy any more. Such an action in turn might have been the only one that enables the execution of other – yet unremoved – actions in the planning graph. Removing them might cause grounded decomposition methods to be removed. This again, might remove other primitive actions.

As an example, consider the methods and primitive tasks shown in Fig. 3. If we assume that the initial state is empty, then only the actions a, c and d are delete-relaxed reachable. Thus, the second method $B \mapsto a, b$ is not reachable as it contains a task – b – that can never become executable. It will thus be pruned from the grounded model. The remaining methods however still remain in the model after the TDG has been computed. One however notices that removing the method $B \mapsto a, b$ also removes the only possibility to obtain the action a via decomposition. It can thus be removed from consideration, i.e. only c and d remain in the grounding. This is where GTOHP and Tree-Rex stop their analysis. PANDA runs an additional delete-relaxed reachability analysis after a and b are pruned. Now, d is not reachable any more, since it is not possible to achieve its precondition x any more. Thus d can be pruned as well and consequently all remaining methods, as well as c. This pruning might again remove other actions from consideration and has thus to be repeated until convergence to achieve a minimal model – which PANDA does.

	pre	add	del
$A \mapsto B, C$	$a : \{\}$	$\{x\}$	$\{\}$
$B \mapsto a, b$	$b : \{z\}$	$\{\}$	$\{\}$
$B \mapsto c$	$c : \{\}$	$\{y\}$	$\{\}$
$C \mapsto d$	$d : \{x\}$	$\{\}$	$\{\}$

Figure 3: Sample Decomposition Methods on the left, and four actions with their preconditions and effects on the right.

```
Data: lifted HTN planning problem $\mathcal{P}$
$\mathcal{P}$ = lifted_domain_simplifications($\mathcal{P}$);
$P_g$ = planning_graph($\mathcal{P}$);
$(A_g, M_g, P_g)$ = tdg($\mathcal{P}$,$P_g$);
$\mathcal{P}_g$ = $(P_g, A_g, M_g)$;
while true do
    $P_g$ = planning_graph($\mathcal{P}_g$);
    $(A_g, M_g, P_g)$ = tdg($\mathcal{P}_g$,$P_g$);
    $\mathcal{P}^*_g$ = $(P_g, A_g, M_g)$;
    if $\mathcal{P}_g = \mathcal{P}^*_g$ then
        break
    end
end
```

Algorithm 1: PANDA's grounding procedure – Overview. Variables are: P_g – grounded primitive actions, A_g – grounded abstract tasks, M_g – grounded methods

Next we will describe the individual steps of PANDA's grounder.

Parameter Splitting

As a first step, PANDA performs simplification operations on the lifted model. For example, we compile disjunctions in preconditions and conditional effects into additional actions, and compile away negative preconditions. Similarly, we compile away variables occurring in preconditions and effects (i.e. those that are contained in quantified expressions) into additional parameters.

Besides these common simplifications, PANDA performs an HTN-specific simplification operation on the lifted model with the aim of reducing the size of the grounding. In some HTN planning domains, lifted decomposition methods contain variables that are (1) used only as parameters of a single subtask and (2) which are not parameters of the abstract task. As an example, consider an abstract task $A(?x)$ with a method decomposing it into the tasks $B(?x, ?y)$ and $C(?x, ?z)$. Further assume that all variables have the same type t which contains the constants $\{c_1, \ldots, c_n\}$. If we ground this method, it has up to n^3 ground instances. Notably, we have to ground every possible combination of the otherwise independent parameters $?y$ and $?z$.

We can equivalently represent this method by three new methods while introducing two new abstract tasks. Let these abstract tasks be $B^*(?x)$ and $C^*(?x)$. The three decomposition methods are $A(?x) \mapsto B^*(?x), C^*(?x)$[5], $B^*(?x) \mapsto B(?x, ?y)$, and $C^*(?x) \mapsto C(?x, ?z)$. For these three methods, there are at most $2n^2 + n$ groundings plus an additional $2n$ new groundings of abstract tasks (B^* and C^*), which is a significant improvement over the original model.

This splitting technique is related to the splitting techniques for action schemas presented by Areces et al. (2014). Their splitting technique also splits lifted actions into multiple ones in order to reduce the number of necessary groundings. They however needed to introduce a mechanism to "coordinate" common parameters between the split actions – i.e. additional state variables (PROCNONE, DO, and PAR(.)). Contrary to their work, pandaPI's splitting has an automatic way to "coordinate" the values of common parameters of split actions: the methods. But in a sense, their technique is a generalisation of ours as we only allow the split if no coordination between the "hidden" variables is needed. Thus adapting their technique might enable more aggressive splits, which is however future work.

Delete-Relaxed Reachability

After the initial simplification of the domain, PANDA performs a delete-relaxed reachability analysis to determine which groundings of primitive tasks can possibly occur in any executable plan. PANDA's implementation is based on the efficient planning graph implementation of STAN (Long and Fox 1999). It is succinct in the sense that it never considers groundings that are not delete-relaxed reachable, similar to the DATALOG-based implementation by Helmert (2009).

TDG-based Hierarchical Reachability

PANDA's hierarchical reachability analysis is based on a data-structure called the Task Decomposition Graph (TDG). It was first introduced by Bercher, Keen, and Biundo (2014) and later refined (Bercher et al. 2017). On the one hand, the TDG is designed to compute all groundings of abstract tasks and methods that are reachable from the initial abstract task. Only those groundings can ever occur during the planning process, as any other grounding can never be obtained via decomposition as required by the solution criteria of HTN planning. On the other hand, the TDG also removes abstract tasks (and connected methods) which cannot be decomposed into a primitive plan. If this is not possible, a task network containing such a task can never be refined into a solution, as this solution must only contain primitive actions. A TDG is a directed graph. Nodes represent either ground tasks or ground methods. A task node has outgoing edges to each applicable ground method, and each method has outgoing edges to its ground subtasks. This means the graph is a representation of hierarchical reachability, i.e. which ground tasks and methods can possibly be reached via decomposition.

Definition 3. *Let $\mathcal{P}$ be an HTN planning problem. The bipartite graph $\mathcal{G} = \langle V_T, V_M, E_{T\to M}, E_{M\to T}\rangle$, consisting of a set of task vertices V_T, method vertices V_M, and edges $E_{T\to M}$ and $E_{M\to T}$ is called the TDG of $\mathcal{P}$ if it holds:*

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

Note that the TDG can represent HTN planning problems that contain cyclic methods. A cyclic decomposition is a sequence of decompositions of a grounded task c that results in a task network containing c again. If the planning problem contains such a cycle, the edge representing the method that produces the recursive occurrence of c simply points back to the vertex created for the first occurrence of c.

TDGs constructed based on the definition contain only those groundings reachable from the initial task by decomposition. As proposed by Elkawkagy, Schattenberg, and Biundo (2010) one can delete those method nodes that contain a primitive task not reachable in a state-based reachability analysis like the planning graph. As a consequence of removing those methods, there may be abstract tasks in the TDG that cannot be decomposed into a task network containing only primitive actions any more. For example, removing a method containing a not delete-relaxed reachable action might remove the only option to exit a recursive method structure. If such an abstract task occurs in a task network during decomposition, we know that it is impossible to refine that task network into a solution. We can thus prune the abstract task – and consequently all methods it is contained it. Removing these methods may again allow us to remove other abstract tasks, thus one can repeat this process until convergence (Def. 4, Nr. 2b). These tasks can be identified in polynomial time by relying on a bottom-up reachability analysis (Alford et al. 2014, proof of Thm. 3.1).

To capture this pruning, we parametrize the previous definition of a TDG by specifying an additional set of primitive ground tasks: these are the actions that are potentially executable, i.e. those tasks that resulted from the delete-relaxed reachability analysis.

Definition 4. *Let $\mathcal{P}$ be an HTN planning problem and $\mathcal{G} = \langle V_T, V_M, E_{T\to M}, E_{M\to T}\rangle$ the respective TDG according to Def. 3. Let X be the set of executable ground actions.*

Then, the pruned TDG $\mathcal{G}_X = \langle V'_T, V'_M, E'_{T\to M}, E'_{M\to T}\rangle$ *is given as the minimal connected subgraph containing c_I such that:*

1. ***Remove Useless Method Vertices***
   A method vertex $v_m = (c, tn) \in V_M$ with $tn = (I, \prec, \alpha)$ is in V'_M if and only if I does not contain a task i with $\alpha(i) \notin X$ in case $\alpha(i)$ is primitive or with $\alpha(i)$ being useless, in case it is abstract (see below).
2. ***Identify Useless Abstract Task Vertices***
   An abstract task vertex $v_t \in V'_T$ is called useless if one of the following holds:

[5]To remain correct, B^* and C^* have the order of B and C.

$$\begin{tikzcd}
A \arrow[dd] & & b \\
& m_3 \arrow[ul] \arrow[ur] & \\
m_1 \arrow[dd] & & C \arrow[ul] \arrow[r] & m_4 \arrow[r] & a \\
& m_2 \arrow[ur] \arrow[dr] & \\
B \arrow[ur] & & c
\end{tikzcd}$$

Figure 4: Sample Task Decomposition Tree.

(a) the pruned TDG $\mathcal{G}_X$ does not contain children for v_t (i.e., all successors of v_t were pruned)

(b) there is no acyclic connected subgraph of the pruned TDG $\mathcal{G}_X$ with root v_t, in which every abstract method vertex has exactly one outgoing edge and no vertex is useless (i.e., the task v_t cannot be decomposed into a set of primitive tasks)

As an example for pruning the grounded domain using the TDG, consider the TDG depicted in Fig. 4. It contains three abstract tasks A, B, and C, three primitive ones a, b, and c, as well as four methods. If all primitives are reachable, no task or method will be pruned. Now consider, as the first case, that the primitive task c is not executable, i.e. that the set of executable ground actions is $X = \{a, b\}$. Then the method m_2 will be removed via condition 1. of Def. 4. Since m_2 has been pruned, B will be pruned as well, as it has no children (condition 2. (a)). This removal cascades and also removes m_1, A, and m_3. Note that b is not removed by Def. 4, but can be removed as it is not contained in any method any more. Thus only C, m_4 and a remain.

As a second example, consider the case where a is not executable, i.e. $X = \{b, c\}$. Here the method m_4 is immediately removed via condition 1. Now, each of the remaining tasks has still applicable methods. It is however not possible to "escape" the cycle formed by the abstract tasks A, B, and C. Thus all three can be pruned from the TDG according to condition 2. (b).

pandaPI's Grounding Procedure

Recent work (Wichlacz, Torralba, and Hoffmann 2019) has shown that PANDA's grounding procedure can be extremely slow on some larger, practical planning instances. Further the analysis has shown that the runtime performance of PANDA's grounder scales extremely badly with increasing size of the input problem.

We have thus developed a new grounder, pandaPI, which we designed to fully replace PANDA's current grounder. The code of pandaPI is publicly available at https://github.com/galvusdamor/pandaPIgrounder. pandaPI and PANDA's grounder differ in three important aspects:

1. pandaPI treats delete-relaxed reachability and TDG-based reachability uniformly
2. pandaPI uses more efficient code for pruning once the domain has been grounded, while PANDA always used the clunky code necessary for a (potentially) lifted domain
3. pandaPI is written in C++, while PANDA is written in Scala

As we will describe in this section, the way in which the planning graph computes the delete-relaxed reachability of a classical planning problem and the computation of a pruned TDG bear striking similarities. We will generalise both procedures into a common framework, the Generalised Planning Graph (GPG). Using it, pandaPI computes both the delete-relaxed reachability and the TDG-based reachability with exactly the same code. Thus both parts of the grounding procedure benefit from improvements to the GPG.

Planning Graph vs TDG

The objective of the planning graph is to determine, based on a given initial state s_I, which actions can possibly be applied in any state that is delete-relaxed reachable from s_I. The planning graph maintains a set of facts F that can potentially become true, which is initialised with s_I. For that, it repeatedly checks whether the preconditions of any lifted action can be fulfilled with the facts in F. If so, that action is instantiated accordingly, and its instantiated effects are added to F. If no new action becomes applicable, the computation stops.

For constructing the TDG, there are two opposing methods. One option is to start grounding with the initial abstract task and ground the planning problem with a depth-first-search-like procedure. In it, the planner essentially applies rules 2. and 3. of Def. 3 until convergence. If this TDG has been constructed, we can use a marker algorithm to perform the pruning according to Def. 4. We start by marking all primitive actions that were reached in the planning graph. Then we recursively mark all methods for which all subtasks are marked and the abstract tasks of a all marked methods. After convergence, we remove all non-marked tasks and methods. This construction, however, can be problematic as it will frequently construct groundings of abstract tasks and methods that will be pruned afterwards – based on the delete-relaxed reachability analysis which was performed beforehand. The second way to construct the pruned TDG tries to mitigate this problem by only constructing grounded instances of methods and abstract tasks if they will not be pruned due to Def. 4. For that, one instantiates methods and abstract tasks only if the marker algorithm to compute decomposable abstract tasks would mark them as well. The algorithm maintains a set A of tasks (both primitive and abstract) that will not be removed by the pruning according to Def. 4. We start with setting A to the set of all actions that are delete-relaxed reachable, i.e. to the result of the planning graph. Then, one repeatedly checks whether there is a (lifted) decomposition method for which all subtasks have ground instances in A. If so, the method is instantiated and the grounding of its abstract task is added to A. This process is repeated until no new methods can be instantiated. This instantiation process might – on the down side – generate instantiations that cannot be reached from the initial abstract task. We therefore perform a depth-first search to discard all such instantiations. As this search is performed on an already grounded model, it is quite fast.

Like the first method to compute the TDG, the second can compute unnecessary groundings, namely those that will be removed by the depth-first search. In order to keep these

	PG	TDG
Instances	facts	tasks
Operators	actions	methods
Antecedants	preconditions	subtasks
Consequents	add effects	abstract task

Table 1: PG vs TDG

tasks to a minimum, we perform a relaxed depth-first search-style propagation of types and constraints through the hierarchy, prior to the actual instantiation process. We determine for every parameter variable of every method, task, and action which constants can be assigned to it – which is often far less than the type of the variables allows. During the instantiation phase, we then disregard all instances which do not comply with the computed possible set of parameters.

Generalised Planning Graph

The computations performed by the planning graph (PG) and the TDG described in the previous section are extremely similar in nature. Both maintain a current set of *instances* with which the preconditions (or *antecedants*) of lifted *operators* are fulfilled. If so, the *operator* is added to the set of reachable groundings and its effects (or *consequents*) are added to the set of instances. This process is repeated until no new ground instantiation of a lifted *operator* can be generated. We show a comparison of the two algorithms in Tab. 1. The only difference between the two algorithms is that the TDG will perform a depth-first search for pruning after grounding. Note that such a pruning step could also be performed for the PG. Here we would remove all ground actions that only produce effects that have no connection to any goal fact, i.e. which can never contribute to reaching the goal. This step is often not performed, as actions are seldom pruned based on this criterion.

These similarities between the PG and TDG have caused us to develop a common algorithm – the Generalised Planning Graph (GPG) – which pandaPI uses to perform both the PG- and the TDG-based grounding. It is essentially an implementation of the standard planning graph algorithm which uses an generic interface to also operate on the input data structures of the TDG. We have opted not to use a DATALOG-based implementation, which is used by Fast Downward (Helmert 2009), as this enables us to add optimisations into the planning graph algorithm that are specific to planning problems. We process new instances (facts and ground tasks, respectively) in a manner s.t. any ground instantiation is only considered once. We split the set of instances I into two: a set of already processed instances I_p and a set of unprocessed instances I_u. We repeatedly remove instances i from I_u and add them to I_p. pandaPI then considers all lifted operators which have an antecedant which can be fulfilled with i. We match the remaining antecedants recursively with instances from I_u. This way, we never consider the same ground instances of a lifted operator twice.

We use a caching strategy to find matching antecedants quickly. For that we process antecedants always in the same fixed order. pandaPI constructs for every type of operator a table with is indexed with the antecedants that is to be matched. The table contains the possible instances that can be matched to the antecedant. For every antecedant, we know which parameter variables have been set by matching the prior antecedants. And thus, we know which of the variable will already have constants assigned to them. pandaPI's table maps the possible instantiations of these variables to the instances that they are compatible with. By construction, all these lists are disjunct. Whenever pandaPI needs to find all matching instances for an antecedant, it can access this table and obtain only the instances that agree on the previously set variables. pandaPI actually does not have to check whether the instance is compatible with the already assigned variables, as it will be by construction. We can maintain this table with little overhead and update it whenever a new instance is added to I_p.

As an example, consider the operator $a(?u, ?v, ?w, ?x)$ with the following antecedants:
1. $p(?v, ?w)$
2. $p(?w, ?x)$
3. $q(?v, ?x, ?u)$

If we are looking to instances for the third antecedant, we will have already matched – by construction of pandaPI's algorithm – the two prior antecedants. Thus values for $?v$, $?w$, and $?x$ will already be known. For the third antecedant, only two variables $?v$ and $?x$ are relevant. Thus, we have a table mapping assignments of these variables to constants. For example, the entry for $[?v = c_1, ?x = c_2]$ might contain the instances $q(c_1, c_2, c_3)$ and $q(c_1, c_2, c_2)$, but not e.g. $q(c_1, c_1, c_1)$ – as this instance does not agree with the values of $?v$ and $?x$.

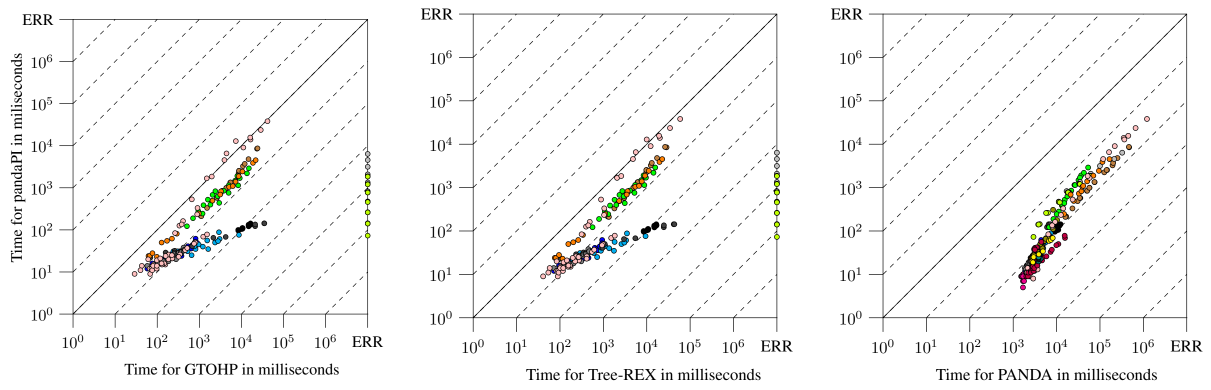
Lastly, we also use a caching procedure to determine whether yet unmatched, i.e. future, antecedants will have no matching instance based on the currently assigned variables. Thus, pandaPI might be able to determine after fixing an instance for the first antecedant, that the third antecedant does not have a matching instance any more. pandaPI will immediately backtrack in this case and thus save the computation time needed for a potential exhaustive search until antecedant three.

pandaPI uses the same general algorithm (Alg. 1) for grounding as PANDA. I.e. we start with lifted domain simplifications and parameter splitting[6], then continue with computing the planning graph and the TDG using the Generalised Planning Graph. The resulting domain is ground. We continue checking delete-relaxed reachability and TDG pruning until no further actions, tasks, and methods can be removed using a specialised, efficient implementation which operates on the grounded problem.

Evaluation

In order to ascertain the quality of the grounding found by pandaPI and its runtime performance when doing so, we have conducted an empirical evaluation. We have compared pandaPI against three other grounders for HTN planning

[6]Note that PANDA's parameter splitting sometimes did not split parameters, due to a bug in the implementation. Hence the size of the computed groundings can differ slightly.



(a) pandaPI vs. GTOHP (b) pandaPI vs. Tree-REX (c) pandaPI vs. PANDA

Figure 5: Runtime of pandaPI vs three other grounders. Time is measured in milliseconds. Scales are logarithmic. Domains are colour-coded, the legend is contained in table 3.

	Time							
	better	equal	worse	min%	max%	min	max	avg.%
PANDA	330	0	0	94.56	99.72	1517	1172525	98.55
GTOHP	198	0	4	-74.53	99.59	-5488	34963	80.17
Tree-Rex	200	0	2	-28.81	99.67	-2874	43119	83.39
	Primitive Tasks							
	better	equal	worse	min%	max%	min	max	avg.%
PANDA	0	330	0	0.00	0.00	0	0	0.00
GTOHP	202	0	0	0.01	99.97	1	676891	50.62
Tree-Rex	96	101	5	-25.00	90.64	-31	205913	26.06
	Abstract Tasks							
	better	equal	worse	min%	max%	min	max	avg.%
PANDA	132	97	101	-47.43	93.48	-490	251	6.86
GTOHP	142	0	60	-68.14	99.11	-635	145016	43.46
Tree-Rex	108	3	91	-129.30	98.17	-871	11945	16.40
	Methods							
	better	equal	worse	min%	max%	min	max	avg.%
PANDA	132	97	101	-73.25	92.86	-17745	251	6.01
GTOHP	182	0	20	-0.01	99.91	-1	1049451	65.21
Tree-Rex	148	3	51	-5.00	98.15	-1	232082	36.19

Table 2: Comparison of time and grounding sizes. The better, equal, and worse columns indicate in how many instances pandaPI was better, equal, or worse than the denoted planner. Min and Max refer to the minimum and maximum improvement that pandaPI makes over all instances.

BARMAN	●	BLOCKSWORLD	●
CHILDSNACK	●	DEPOTS	●
ENTERTAINMENT	●	GRIPPER	●
HIKING	●	MINECRAFT-AREA	●
MINECRAFT-NORMAL	●	PCP	●
ROVER-PANDA	●	ROVER	●
SATELLITE	●	SMARTPHONE	●
TRANSPORT	●	UM-TRANSLOG	●
WOODWORKING	●		

Table 3: Colour-codes for the individual domains.

problems: PANDA's grounder, GTOHP's grounder (Ramoul et al. 2017), and Tree-Rex's grounder (Schreiber et al. 2019). We have not compared against FAPE (Dvorak et al. 2014) as it cannot handle HTN planning problems containing recursion – while most benchmark instances contain recursion.

The benchmark set comprises 330 problem instances from 17 domains. 8 domains (BARMAN, BLOCKSWORLD, CHILDSNACK, DEPOTS, GRIPPER, HIKING, ROVER-GTOHP, SATELLITE-GTOHP) with 160 instances stem from the evaluations of GTOHP and Tree-Rex. 2 domains (MINECRAFT-AREA, MINECRAFT-NORMAL) with 26 instances stem from Wichlacz, Torralba, and Hoffmann (2019). The remaining 7 domains (UM-TRANSLOG, SATELLITE-PANDA, WOORWORKING, SMARTPHONE, PCP, ENTERTAINMENT, ROVER-PANDA, TRANSPORT) with 144 instances stem from the evaluations of PANDA (Höller et al. 2018b; Behnke, Höller, and Biundo 2019a). All instance are available for download at pandapi.hierarchical-task.net/domains.

Since GTOHP and Tree-Rex can only handle totally-ordered HTN planning problems, both were executed only on those 228 instances of the benchmark set that are totally-ordered, i.e. their own 8 domains plus both MINECRAFT domains, TRANSPORT, and ENTERTAINMENT. All experiments were conducted on an Intel Xeon E5-2660 with a 20 GB RAM limit for each instance. We set no time-limit.

pandaPI was able to ground all 330 problem instances with a maximum runtime of 38.166 seconds. The same is true for PANDA, whose runtime reached up to 20 minutes and 10 seconds. GTOHP was able to ground 202 of the 228 instances given to it with a maximum runtime of 41.609 seconds. It failed to ground all 26 MINECRAFT instances by exceeding the memory limit after at least 11 minutes of computation time per instance. Tree-Rex – as it uses GTOHP as its first step – also grounds 202 instances and fails on MINECRAFT with a maximum runtime of 58.935 seconds.

A per-instance comparison of the runtime between pandaPI, PANDA, GTOHP, and Tree-Rex is shown in Fig. 5. A statistical summary of timings and grounding sizes is shown in Tab. 2. For PANDA and pandaPI we do not count the actions representing grounded method preconditions. GTOHP and Tree-Rex consider them as part of grounded methods, thus counting them separately for PANDA and pandaPI would skew the results significantly – as this is simply a question of where to store these the method precondition. Notably, there are only two instances where pandaPI is slower than GTOHP, and four where it is slower than Tree-Rex, and it is always faster than PANDA. The instances where pandaPI is slower than GTOHP and Tree-Rex all stem from the SATELLITE-GTOHP domain and the difference is at most 5.488 seconds.

Further, pandaPI creates a smaller grounding than GTOHP in all instances, and in 96 instances compared to Tree-Rex. The five instances in which it produces more primitive tasks stem from the ENTERTAINMENT domain. Here, pandaPI compiles conditional effects into multiple actions, while GTOHP and Tree-Rex can handle conditional effects natively. When considering abstract tasks and methods, pandaPI often generates smaller groundings, but sometimes also generates larger groundings with respect to the number of abstract tasks. This is due to our parameter splitting which may introduce additional abstract tasks. If it is turned off (see supplemental material), our grounding is never worse than that of GTOHP. Note that in those 20 / 51 instances in which pandaPI produces more methods, it generates only one additional method. This is due to a compilation performed by pandaPI in case the domain specifies multiple initial abstract tasks. If so, pandaPI added a new artificial initial abstract task and a method decomposing it into the specified tasks.

pandaPI generates on average a significantly smaller grounding in a significantly smaller timeframe than the previous grounders of PANDA, GTOHP, and Tree-Rex. Especially, we think that we have overcome with pandaPI the current problems of grounding in HTN planning as shown by Wichlacz, Torralba, and Hoffmann (2019).

We've lastly compared pandaPI's implementation of the GPG with the currently state-of-the-art DATALOG-based implementation in Fast Downward's grounder (Helmert 2009). We have considered those instances that were identified as the most time-consuming to ground for Fast Downward (Helmert 2009) as reported by Helmert. These instances stem either from the domains SATELLITE or PSR. Since pandaPI does not support derived predicates – which are an essential part of the modelling of PSR – we have compared our implementation of the planning graph against Fast Downward's DATALOG-based implementation on those SATELLITE instances which they reported as problematic. The results are shown in Tab. 4 and show that pandaPI is significantly faster than Fast Downward on these instances.

Conclusion

Most recent systems in HTN planning realise the planning process in a fully grounded way. A smaller grounding usually improves the performance of the planner. For example, a smaller grounding allows for heuristics to be computed faster – and for them to be more precise. Further, the search mechanics of the planner is faster the smaller the grounding is, as fewer actions and methods have to be considered. Lastly, a smaller grounding also reduces the size of encodings, e.g. into propositional logic, which makes the translated problem (potentially) easier to solve. Despite these advantages, little work has been published on grounding techniques especially for HTN planning. We presented grounding procedure of PANDA and pandaPI and discuss how to compute them efficiently. Our empirical evaluation shows that it leads to smaller groundings than other grounder and uses less time to compute it.

	FD	pandaPI		FD	pandaPI
#31	15.410s	0.588s	#32	25.840s	1.178s
#33	38.765s	1.339s	#34	9.000s	0.338s
#35	13.590s	0.511s	#36	17.200s	0.620s

Table 4: Time to ground instances of the classical SATELLITE domain for Fast Downward (FD) vs pandaPI

Acknowledgements

This work was partially funded by the technology transfer project "Do it yourself, but not alone: Companion-Technology for DIY support" of the SFB/TRR 62 funded by the German Research Foundation (DFG). The industrial project partner is the Corporate Research Sector of the Robert Bosch GmbH.

References

Alford, R.; Bercher, P.; and Aha, D. W. 2015. Tight bounds for HTN planning. In *Proc. of the 25th Int. Conf. on Autom. Plan. and Sched. (ICAPS 2015)*.

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. 2014. On the feasibility of planning graph style heuristics for HTN planning. In *Proc. of the 24th Int. Conf. on Autom. Plan. and Sched. (ICAPS 2014)*, 2–10. AAAI Press.

Alford, R.; Shivashankar, V.; Roberts, M.; Frank, J.; and Aha, D. W. 2016. Hierarchical planning: Relating task and goal decomposition with task sharing. In *Proc. of the 25th Int. Joint Conf. on AI (IJCAI 2016)*.

Areces, C. E.; Bustos, F.; Dominguez, M.; and Hoffmann, J. 2014. Optimizing planning domains by automatic action schema splitting. In *Proc. of the 24th Int. Conf. on Autom. Plan. and Sched. (ICAPS 2014)*, 11–19.

Behnke, G., and Biundo, S. 2018. X and more parallelism: Integrating LTL-next into SAT-based planning with trajectory constraints while allowing for even more parallelism. *Inteligencia Artificial* 21(62):75–90.

Behnke, G.; Höller, D.; Bercher, P.; and Biundo, S. 2016. Change the plan – How hard can that be? In *Proc. of the 26th Int. Conf. on Autom. Plan. and Sched. (ICAPS 2016)*.

Behnke, G.; Schiller, M.; Kraus, M.; Bercher, P.; Schmautz, M.; Dorna, M.; Dambier, M.; Minker, W.; Glimm, B.; and Biundo, S. 2019. Alice in DIY-wonderland or: Instructing novice users on how to use tools in DIY projects. *AI Communications* 32(1):31–57.

Behnke, G.; Höller, D.; and Biundo, S. 2015. On the complexity of HTN plan verification and its implications for plan recognition. In *Proc. of the 25th Int. Conf. on Autom. Plan. and Sched. (ICAPS 2015).*

Behnke, G.; Höller, D.; and Biundo, S. 2017. This is a solution! (... but is it though?) – Verifying solutions of hierarchical planning problems. In *Proc. of the 27th Int. Conf. on Autom. Plan. and Sched. (ICAPS 2017).*

Behnke, G.; Höller, D.; and Biundo, S. 2018a. totSAT – Totally-ordered hierarchical planning through SAT. In *Proc. of the 32nd AAAI Conf. on AI (AAAI 2018).*

Behnke, G.; Höller, D.; and Biundo, S. 2018b. Tracking branches in trees – A propositional encoding for solving partially-ordered HTN planning problems. In *Proc. of the 30th Int. Conf. on Tools with Art. Int. (ICTAI 2018).*

Behnke, G.; Höller, D.; and Biundo, S. 2019a. Bringing order to chaos – A compact representation of partial order in SAT-based HTN planning. In *Proc. of the 33rd AAAI Conf. on AI (AAAI 2019).*

Behnke, G.; Höller, D.; and Biundo, S. 2019b. Finding optimal solutions in HTN planning – A SAT-based approach. In *Proc. of the 28th Int. Joint Conf. on AI (IJCAI 2019).*

Bercher, P.; Höller, D.; Behnke, G.; and Biundo, S. 2016. More than a name? On implications of preconditions and effects of compound HTN planning tasks. In *Proc. of the 22nd Europ. Conf. on AI (ECAI 2016).*

Bercher, P.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An admissible HTN planning heuristic. In *Proc. of the 26th Int. Joint Conf. on AI (IJCAI 2017).*

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid planning heuristics based on task decomposition graphs. In *Proc. of the 7th Ann. Symp. on Combinatorial Search (SoCS 2014).*

Blaylock, N., and Allen, J. 2005. Generating artificial corpora for plan recognition. In *Proc. of the 10th Int. Conf. on User Modeling (UM 2005).*

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1-2):281–300.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence* 69(1-2):165–204.

Dvorak, F.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014. A flexible ANML actor and planner in robotics. In *Proc. of the 4th Work. on Plan. and Rob. (PlanRob 2014).*

Elkawkagy, M.; Bercher, P.; Schattenberg, B.; and Biundo, S. 2012. Improving hierarchical planning performance by the use of landmarks. In *Proc. of the 26th AAAI Conf. on AI (AAAI 2012).*

Elkawkagy, M.; Schattenberg, B.; and Biundo, S. 2010. Landmarks in hierarchical planning. In *Proc. of the 20nd Europ. Conf. on AI (ECAI 2010)*, 229–234. IOS Press.

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

Geier, T., and Bercher, P. 2011. On the decidability of HTN planning with task insertion. In *Proc. of the 22nd Int. Joint Conf. on AI (IJCAI 2011).*

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research (JAIR)* 26.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* 173(5-6):503–535.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)* 14:2531–302.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language classification of hierarchical planning problems. In *Proc. of the 21st Europ. Conf. on AI (ECAI 2014).*

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2016. Assessing the expressivity of planning formalisms through the comparison to formal languages. In *Proc. of the 26th Int. Conf. on Autom. Plan. and Sched., (ICAPS 2016).*

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2018a. Plan and goal recognition as HTN planning. In *Proc. of the 30th Int. Conf. on Tools with Art. Int. (ICTAI 2018).*

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, B. 2018b. A generic method to guide HTN progression search with classical heuristics. In *Proc. of the 28th Int. Conf. on Autom. Plan. and Sched. (ICAPS 2018).*

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2018c. Plan and goal recognition as HTN planning. In *Proc. of the AAAI 2018 Workshop on Plan, Activity, and Intent Recognition (PAIR 2018).*

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).*

Koehler, J., and Hoffmann, J. 2000. On the instantiation of ADL operators involving arbitrary first-order formulas. In *Proc. of the 14th Work. on New Results in Plan., Sched. and Design (PuK 2000)*, 74–82.

Long, D., and Fox, M. 1999. Efficient implementation of the plan graph in STAN. *Journal of Artificial Intelligence Research* 10:87–115.

McDermott, D. 2000. The 1998 AI planning systems competition. *AI Magazine* 21(2):35–55.

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

Rintanen, J.; Heljanko, K.; and Niemelä, I. 2006. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence* 170(12-13):1031–1080.

Rintanen, J. 2014. Madagascar: Scalable planning with SAT. In *The 2014 Int. Plann. Comp.: Descr. of Part. Plann., Determ. Track.*

Schreiber, D.; Balyo, T.; Pellier, D.; and Fiorino, H. 2019. Tree-REX: SAT-based tree exploration for efficient and high-quality HTN planning. In *Proc. of the 29th Int. Conf. on Autom. Plan. and Sched. (ICAPS 2019)*, 382–390.

Wichlacz, J.; Torralba, A.; and Hoffmann, H. 2019. Construction-planning models in minecraft. In *Proc. of the 2nd ICAPS Workshop on Hierarchical Planning.*