More Succinct Grounding of HTN Planning Problems – Preliminary Results

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

Abstract

Planning systems usually operate on grounded representations of the planning problems during search. Further, planners that use translations into other combinatorial problems also often perform their translations based on a grounded model. Planning models, however, are commonly defined in a lifted formalism. As such, one of the first preprocessing steps a planner performs is to generate a grounded representation. In this paper we present a new approach for grounding HTN planning problems that produces smaller groundings than the previously published method. We expect this decrease in size to lead to more efficient planners.

1 Introduction

Most modelling languages for planning problems (such as PDDL (McDermott 2000)) allow for specifying planning problems in a *lifted* fashion, e.g. by allowing the modeller to specify actions with parameters whose preconditions and effects are specified using literals referring to these parameters. Using a lifted representation, a modeller can easily write models with a large number of instantiated actions without the need to enumerate them explicitly. More importantly, a lifted representation of the planning problem enables the modeller to specify a single planning domain that can be used in multiple planning problems without any change to the model. In a grounded formalism, the domain (e.g. the set of actions) changes depending on the planning problem at hand, while it does not in a lifted representation.

Unfortunately, to plan directly using only the lifted model is rather difficult, which is witnessed by the absence of a large body of work e.g. on lifted heuristics. Most planners transform the lifted representation of the planning problem they receive as an input into a grounded representation before planning. Planning is then performed on the grounded representation, for which heuristics are readily available. 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. Smaller groundings are generally advantageous to planners, as their per-search-node effort decreases and the quality of heuristics can improve. Even small decreases in the size of the grounding can have a huge impact on the efficiency of the planner. As such, grounding is a critical step in the process of planning.

For Hierarchical Task Network (HTN) planning (Bercher, Alford, and Höller 2019), there is – as far as we know – only a single paper explicitly concerned with grounding HTN planning domains (Ramoul et al. 2017). Several other HTN planners plan in a grounded way (e.g. FAPE (Dvorak et al. 2014) and PANDA (Bercher, Keen, and Biundo 2014; Bercher et al. 2017)), but there is no published work about their grounding procedures.

In this paper we report on the grounding procedure used in a variety of systems based on the PANDA framework, e.g. the plan-space-based system (Bercher, Keen, and Biundo 2014; Bercher et al. 2017), the progression-based system (Höller et al. 2018; Höller et al. 2019b), the SAT-based system for totally and partially ordered HTN planning (Behnke, Höller, and Biundo 2018a; 2018b; 2019a; 2019b), and the SAT-based HTN plan verifier (Behnke, Höller, and Biundo 2017). PANDA and its grounder have also already been used in practical applications where a fast grounding procedure is necessary. Notably, we have used it to create plans instructing novice users on how to use electronic tools for DIY home-improvement projects (Behnke et al. 2018; 2019). We start by describing the lifted HTN planning formalism, then give an overview of grounding in planning, describe the grounding procedure used by PANDA, and lastly compare our grounding against the grounding found by GTOHP (Ramoul et al. 2017).

2 Lifted HTN Planning Formalism

Before explaining our 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 predicate'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. Based on the predicate logic $\mathcal{L}$, we denote with S the power set of all ground facts

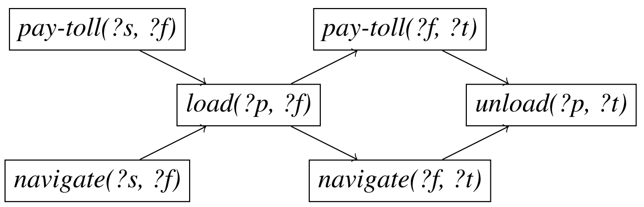

Figure 1: A task network in a simple transportation domain. If performed, it will transport a package from its initial location to its target location. We assume that there is just one transporter. The variables *?s* (start location), *?f* (initial package location), and *?t* (target package location) 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.

over $\mathcal{L}$.

The most basic data structure in HTN planning is a *task network*. It represents a partially ordered multi-set of tasks. HTN planning distinguishes two types of tasks: primitive and abstract ones. Task networks can contain both primitive and abstract tasks. Each task is identified by its task name and a parameter sequence. For instance, a (primitive) task for driving from a source location $?ls$[1] to a destination location $?ld$ is denoted by the first-order atom *drive(?ls, ?ld)*. We do not differentiate between the expressions *task* and *task names* – both are used synonymously.

Definition 1 (Task Network)**.** *A task network tn over a set of* task names *X (first-order atoms) is a tuple $(I, \prec, \alpha, VC)$ with the following elements:*

1. *I is a finite (possibly empty) set of* task identifiers*.*
2. *$\prec$ is a strict partial order over I.*
3. *$\alpha : I \to X \times \overline{V}$ maps task symbols to task names and their parameter variables*
4. *VC is a set of variable constraints. Each constraint can bind two task parameters to be (non-)equal or it can constrain a task parameter to be (non-)equal to a constant.*

For simplicity, we only allow variables as arguments for tasks. If it is desired that an argument of a task should be a constant, one can simply introduce a new variable and bind it to the value of the constant in VC. As an example for a task network consider the one shown in Fig. 1.

Task networks can contain primitive and abstract tasks. *Primitive tasks* are identical to actions in classical planning. They are identified via first-order atoms like *drive(?f, ?t)* and are specified via their preconditions *pre* and effects *eff*. For the purposes of this paper, we assume that *pre* is a conjunction of positive first-order literals over $\mathcal{L}$'s predicates and *eff* is a conjunction of (positive and negative) first-order literals. The variables occurring in *pre* and *eff* must be parameters of *name*. Note that for more complex preconditions and effects, this normal form can be achieved via compilation into (potentially) multiple new actions. However a native approach for handling them is usually more efficient.

Abstract tasks are identified by their name and arguments, e.g. *navigate(?f, ?t)*. 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, VC) consisting of an abstract task name c, a task network tn, and a set of variable constraints VC. The variable constraints VC allow to specify (co)designations between the parameters of c and either the variables in the task network tn or constants.

An HTN planning problem consists of the problem's primitive and abstract tasks, all available decomposition methods, the initial state and the initial task network.

Definition 2 (Planning Problem)**.** *A lifted HTN planning problem $\mathcal{P}$ is a tuple $(\mathcal{L}, T_P, T_C, M, s_I, tn_I)$, where:*

- *$\mathcal{L}$ is a quantifier- and function-free first-order predicate logic.*
- *T_P and T_C are finite sets of primitive and abstract tasks.*
- *M is a finite set of decomposition methods with abstract tasks from T_C and task networks over the names $T_P \cup T_C$.*
- *$s_I \in S$ is the initial state, i.e., a ground conjunction of positive literals over the predicates assuming closed world assumption.*
- *tn_I is the initial task network, not necessarily ground.*

An HTN planning problem is called ground if all predicates of its predicate logic have arity zero (i.e. they have no parameters).

The aim in an HTN planning problem is to refine a given initial abstract task s_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 via variable constraints. It is further executable if there is a linearisation of its tasks that is executable in the initial state. The refinement of the initial task network is performed via repeatedly applying decomposition methods to the abstract tasks contained in it and the resulting task networks. Applying a decomposition method (c, tn_c, VC) 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 and to add the variable constraints VC to the resulting task network. In addition we have to add variable constraints that co-designate the parameter variables of the abstract task in the method with the actual parameters of the task c that is decomposed inside tn.

Definition 3 (Decomposition)**.** *Let $m = (c(?x_1, \ldots, ?x_n), tn_m)$ with $tn_m = (I_m, \prec_m, \alpha_m, VC_m))$ be a decomposition method, $tn_1 = (I_1, \prec_1, \alpha_1, VC_1)$ 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 $tn_2 = (I_2, \prec_2, \alpha_2, VC_2)$ if and only if $\alpha_1(i) = c(?y_1, \ldots, ?y_n)$ and*

$$\begin{aligned}
I_2 &= (I_1 \setminus \{i\}) \cup I_m \\
\prec_2 &= (\prec_1 \cup \prec_m \cup \\
&\quad \{(i_1, i_2) \in I_1 \times I_m \mid (i_1, i) \in \prec_1\} \cup \\
&\quad \{(i_1, i_2) \in I_m \times I_1 \mid (i, i_2) \in \prec_1\})
\end{aligned}$$

[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.

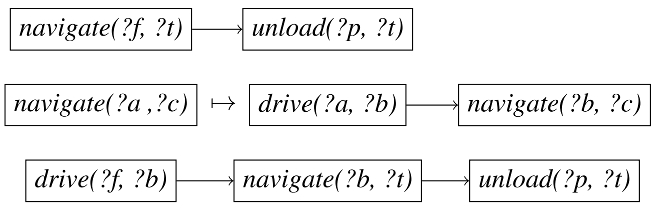

Figure 2: The first row shows a task network tn_1. The second row shows a method for the navigate task. The result of applying this method to the navigate task in tn_1 results in the task network shown in the third row.

$$\begin{aligned} &\setminus \{(i', i'') \in I_1 \times I_1 \mid i' = i \text{ or } i'' = i\} \\ \alpha_2 &= (\alpha_1 \cup \alpha_m) \setminus \{(i, c(?y_1, \ldots, ?y_n))\} \\ VC_2 &= VC_1 \cup VC_2 \cup \{?x_i = ?y_i \mid 1 \leq i \leq n\} \end{aligned}$$

Instead of introducing additional equality constraints over the variables, we could also replace all occurrences of $?x_i$ in tn_1 with $?y_1$, which is more useful in practice as it introduces variables only if necessary. Further, variable constraints are simply added and not propagated. This eliminates the necessity for handling constraints between propagated variables. Of course, an implementation would always propagate variable constraints as far as possible.

In Def. 1, we allowed a task network to contain no tasks at all, i.e. we allowed for I to be the empty set. Thus methods may decompose abstract tasks into such task networks. This is sensible and occurs (somewhat) frequently in practice. Consider the abstract task $navigate(?f, ?t)$ where $?f = ?t$. Such a task can be achieved without doing anything, i.e. by the empty task network. All ordering constraints relating to it are not lost, as their transitive implications are kept during decomposition. Similarly, the parameter variables of the decomposed abstract task remain variables of the decomposed task network, i.e. no constraints can be lost.

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

3 Grounding Planning Problems

As in classical planning, both theoretical reserach (see e.g. (Höller et al. 2014; Behnke, Höller, and Biundo 2015; Höller et al. 2016; Behnke et al. 2016; Bercher et al. 2016; Alford et al. 2016)) and practical research (Behnke et al. 2018; 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 of the search itself 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 and abstract tasks, and methods and replace their lifted versions appropriately by them. For details regarding the full grounding process we refer to the work of Alford, Bercher, and Aha (2015). Naturally such a grounding will be exponential in size with respect to the original domain. As such, 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[2]. 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 (exactly) whether this is the case is unfortunately undecidable. This is caused by the fact that deciding whether a given HTN planning problem has a solution or not is undecidable (Erol, Hendler, and Nau 1996). If determining that a task occurs in no solution would (in general) be decidable, we would have a finite-time procedure for testing whether a given HTN planning problem is solvable: simply run the test on all its primitive tasks. A solution exists if and only if at least one of them is contained in any solution (excluding the detectable case of a possible empty solution, which can be tested in advance in polynomial time).

Instead, we aim at computing an approximation of this property. I.e. 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. As such, we do not include a grounding if we can prove that it cannot be contained in a solution.

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. For a given action one can determine in polynomial time whether it is part of any delete-relaxed 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[3]. 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

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

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

to grounding in more detail, which is the grounder of GTOHP (Ramoul et al. 2017). It uses a grounding procedure similar to that of FF (Hoffmann and Nebel 2001) and similarly uses the concept of inertia (Koehler and Hoffmann 2000). Inertia of a predicate describes 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.

Note that the procedure used by GTOHP removes effectless actions from the methods they are contained in (Ramoul et al. 2017). The respective methods are not pruned afterwards (which would be incorrect), 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 as it makes any found solution (potentially) invalid as it may not adhere to the solution criteria of HTN planning. As a notable example, a state-based goal description (like used in classical planning) can be encoded in an HTN planning problem as an additional effectless action, which would be pruned by GTOHP. As such, the planner would not be obliged to reach a goal state. Also "moving" the preconditions of these effectless actions to other actions within the same method is not correct (i.e. equivalent transformation). Moving the precondition would require it to hold in conjunction with the precondition of another action, which is not required in the original problem, as another action could be ordered in between. Secondly, the implementation of GTOHP does not allow for two parameters of one 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 invalid 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 (Sec. 5), we have fixed both issues in the code of GTOHP.

4 Grounding HTNs

Our grounding procedure includes three steps: a lifted domain simplification, computing delete-relaxed reachability, and a hierarchical reachability analysis based on a graph called the Task Decomposition Graph (TDG) (Elkawkagy et al. 2012; Bercher et al. 2017).

4.1 Parameter Splitting

As a first step, we perform simplification operations on the lifted model. For example, we compile disjunctions in preconditions 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.

Beside these common simplifications known from classical planning, our grounder 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 or very few subtasks 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 $C = \{c_1, \dots, c_n\}$. If we ground this method, it has 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 $A(?x) \mapsto B^*(?x), C^*(?x)$[4], $B^*(?x) \mapsto B(?x, ?y)$, and $C^*(?x) \mapsto C(?x, ?z)$. For these three methods, there are $2n^2 + n$ groundings plus an additional $2n$ new groundings of abstract tasks (B^* and C^*), which is a significant improvement over the original model.

In general, we can perform this operation whenever there is a variable $?x$ in a method that is only a parameter of one of the subtasks A and its variable constraints connect it only to the other parameters of A. We can sometimes also perform this splitting of parameters into additional methods if the variable occurs in multiple subtasks $A_1, \dots, A_k$. Since we have to equivalently transform the model, we have to assure that by applying multiple decompositions, the original method is still correctly represented in the model. As such, we only split away a group of actions $A_1, \dots, A_k$ if all of them have the same relative ordering against the other tasks in the method. We then replace them by a new single abstract task A^* with this relative order and a method for A^* decomposing it into $A_1, \dots, A_k$ with their internal order. By applying the method for A^* to the instance of A^* in the changed main decomposition method, we obtain the original method. Note that A^* can have more parameter variables than the individual actions, i.e. it can increase the number of abstract tasks significantly. Their number is however limited by the number of ground methods. Thus we assume that this compilation is not an issue in practice.

4.2 Delete-Relaxed Reachability

After the initial simplification of the domain, we perform a delete-relaxed reachability analysis to determine which groundings of primitive tasks can possibly occur in any executable task network. Our implementation 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). We have opted for a native implementation of the planning graph algorithm.

4.3 TDG-based Hierarchical Reachability

Our hierarchical reachability analysis is based on a data-structure called the Task Decomposition Graph (TDG).

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

We first introduce the definition as given by Bercher et al. (2017). After introducing the declarative definition, we describe how it is built algorithmically (in the next section).

Definition 4 (Task Decomposition Graph (TDG))**.** *Let $\mathcal{P} = \langle \mathcal{L}, T_P, T_C, M, s_I, tn_I \rangle$ be an HTN planning problem. Without loss of generality, we assume that tn_I contains just a single ground abstract task* TOP *for which there is exactly one method in M.*[5]

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)*
   TOP $\in V_T$*, the TDG's root.*
2. ***Method Vertices*** *(derived from task vertices)*
   Let $c \in V_T$ and there is a method $(c, tn, VC) \in M$. Then, for all groundings v_m that satisfy the variable constraints in VC 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 $v_m = (c, tn, VC)$ and $tn = (I, \prec, \alpha, VC)$. 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.

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. As can be seen from the definition, it is bound linearly in the number of ground methods. It can be constructed in linear time in case the planning problem $\mathcal{P}$ is ground and in exponential time in case $\mathcal{P}$ is lifted.

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[6], 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. 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).

We parametrize the previous definition of a TDG by specifying an additional set of primitive ground tasks: these are the actions that are (supposed to be) reachable (like, e.g. the actions reachable in the planning graph).

Definition 5 (Pruned TDG)**.** *Let $\mathcal{P} = \langle \mathcal{L}, T_P, T_C, M, s_I, tn_I \rangle$ 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. 4. Let X be the set of actions as given above.*

Then, the pruned TDG *$\mathcal{G}_X = \langle V'_T, V'_M, E'_{T\to M}, E'_{M\to T}\rangle$ that exploits the reachability information for the actions in X is given as the minimal connected subgraph containing* TOP *such that:*

1. ***Remove Useless Method Vertices***
   A method vertex $v_m = (c, tn, VC) \in V_M$ with $tn = (I, \prec, \alpha, VC)$ 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:
   (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)

Whereas the parameter splitting described before is proposed in this paper the very first time, the TDG-based grounding procedure in contrast has a rather long history. Initial ideas were first proposed by Elkawkagy, Schattenberg, and Biundo (2010) showing how to compute a pruned decomposition *tree* (TDT), which was exploited during search. They described the key ideas of deleting actions that cannot be reached by a delete-relaxed reachability analysis, triggering further deletions of methods and possibly abstract tasks. The TDT was subsequently extended to a graph (Elkawkagy et al. 2012), but without altering the deployed reachability analysis. Later, we extended the reachability analysis to also prune those abstract tasks from the TDG that cannot be refined into a primitive task network (Bercher et al. 2017). However, we did not yet provide a formal, declarative definition of the resulting pruned TDG there. Furthermore, similar to the work by Elkawkagy, Schattenberg, and Biundo (2010; 2012) we only explained that this pruned TDG may be used

[5]If the problem specifies an initial partial plan tn_I we can obtain the required form by adding a new artificial (parameter-free) abstract task TOP that decomposes exactly into tn_I.

[6]Or in rare cases by a mistake of the modeller.

as a basis for heuristics. Here we explain how it also serves the purpose of obtaining a ground model. The following section is another yet unpublished contribution that is essential for the efficiency of the grounding/TDG construction procedure.

4.4 Avoiding the Bottleneck

Beside the final size of the grounding, it is – in practice – crucial to avoid large sets of intermediate groundings during the computation process. A naive idea would be to compute the full TDG and prune it afterwards. This corresponds to computing a full instantiation of all actions in classical planning and performing a reachability analysis on them. Both the full TDG and the full instantiation of actions usually contain unnecessary groundings that will be pruned afterwards. In this section we describe how the pruned TDG can be computed (somewhat) efficiently, without the need to compute the full TDG first.

When building the TDG in a top-down manner, it will initially include the initial task and is iteratively extended by adding nodes for each applicable method and its subtasks. To handle cyclic decompositions, we keep a set of created task groundings. Whenever we add a new decomposition method, we check for all its subtasks whether they are contained in the set of already created task groundings. If so, we use that already existing node to add the respective edge implied by the method to the graph and don't recurse through that grounding – as it has already been (or is in the process of being) fully expanded. This way, the procedure terminates also on cyclic HTN planning problems and it is ensured that tasks that are not reachable via the hierarchy are never included. However, when primitive tasks are added to the graph, it has to be checked whether these are reachable via state transition, and given that they are not, the graph has to be pruned as given in Def. 5.

Alternatively, one could construct a superset of the pruned TDG in a bottom-up manner. We start with nodes for the primitive tasks that are reachable under delete-relaxation. Then, for each method where all subtasks are included in the graph, nodes for the method and for the task the method decomposes are included. Methods and tasks are added until convergence. We again use a set of created task groundings to handle cyclic decompositions in the planning problem. That way, the graph never includes tasks that would need to be pruned based on state-based reachability information. However, it might include tasks and methods that are not reachable from the initial task. These can be removed by a depth-first search afterwards.

When using both the top-down and the bottom-up computation, state-based and hierarchical reachability analysis influence each other. The hierarchy might e.g. exclude actions that are necessary to fulfil other actions' preconditions. An action a_1 pruned due to state-based reachability may exclude a method that has been the only source of reachability of another action a_2. This means that the two analyses should be iterated until the grounding converged. PANDA's grounding does so.

The question is now if a system should rely on the top-down or the bottom-up building process. There is no

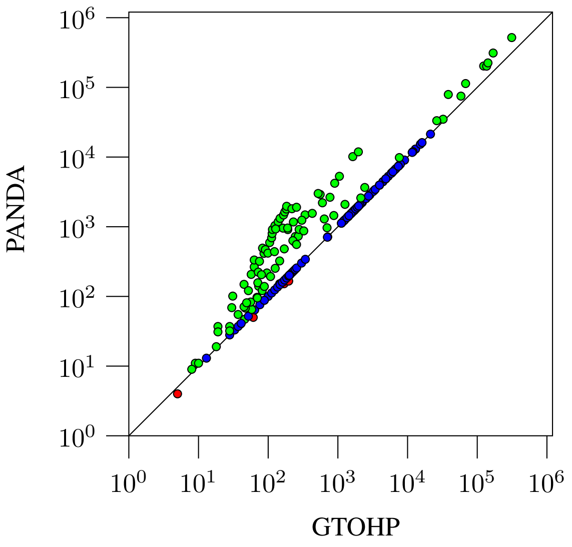

Figure 3: Number of primitive tasks in groundings computed by GTOHP and our grounder. Green indicates that PANDA finds the smaller grounding, red that GTOHP does and blue indicates that both groundings have equal size.

(domain-independent) answer to this question. It is easy to construct planning problems that result in large intermediate graphs for both ways of construction. Which one works better depends on the specific structure of the problem at hand. We do not yet know how to determine which algorithm will perform better a priori. Therefore we developed a way of construction that combines the benefits of both procedures.

We start with a top-down construction, but instead of creating the grounded nodes directly, it maintains for each parameter of each task and method a list of all constants that might be assigned to the parameter. This avoids the creation of all combinations of constants, to the cost of losing the information which parameter *combinations* are valid. When primitive tasks are reached, the constant set is further reduced via state-based reachability, and this reduction is propagated through the graph. In a second step, top-down grounding is performed using the reduced constant sets.

5 Evaluation

The implementation of the described grounding procedure is included in the PANDA planner. It is primarily implemented in Scala. The grounder accepts HTN planning problems formulated in HDDL as its input (Höller et al. 2019a).

In this preliminary evaluation we do not compare the runtime of the different approaches (top-down, bottom-up, two-way), but compare the size of the resulting grounding with a system from related work. We are currently re-implementing the grounder and will present runtime results in a follow-up paper.

We have compared our grounding procedure against GTOHP (Ramoul et al. 2017), which is the so-far only published grounding procedure for HTN planning. In the evaluation, we have included all 202 instances used in the recent

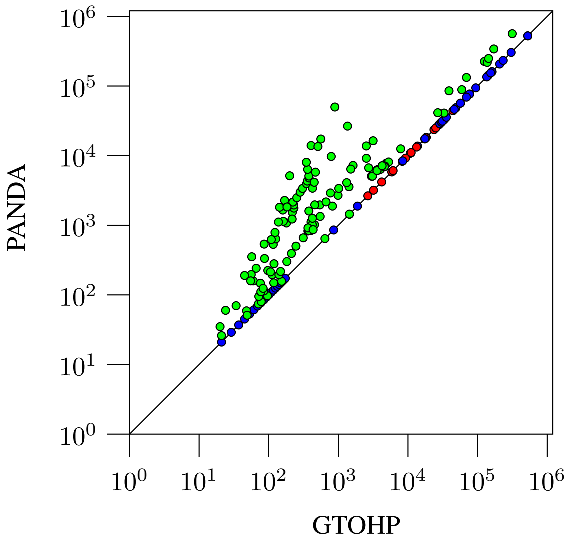

Figure 4: Number of decomposition methods in groundings computed by GTOHP and our grounder. Green indicates that PANDA finds the smaller grounding, red that GTOHP does and blue indicates that both groundings have equal size.

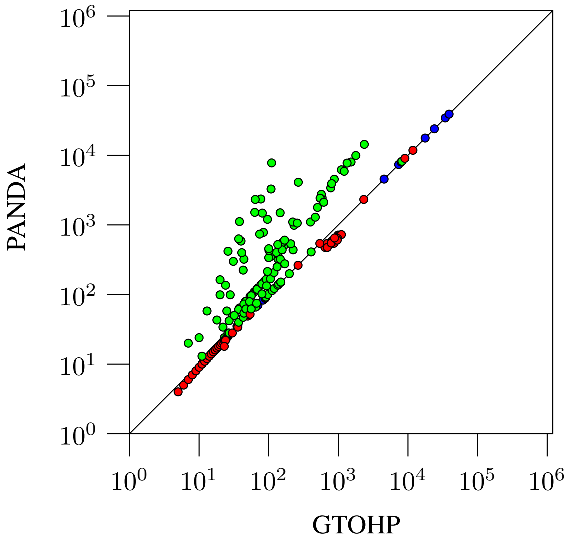

Figure 5: Number of abstract tasks in groundings computed by GTOHP and our grounder. Green indicates that PANDA finds the smaller grounding, red that GTOHP does and blue indicates that both groundings have equal size.

evaluation of Tree-Rex (Schreiber et al. 2019), which uses the GTOHP grounder.

Except for one domain (TRANSPORT), we always found at most as many primitive tasks in the grounding as GTOHP. In TRANSPORT, we usually needed a few more ground instances, as its actions contain disjunctive preconditions which we compile away while GTOHP handles them natively. A scatter plot of the results is shown in Fig. 3. We find smaller groundings in 96 instances, larger ones in 5, and 101 groundings of equal size. On average our groundings are 26.19% smaller with a maximum of 90.64% or 205.913 tasks reduction.

For the number of methods, results are shown in Fig. 4. We find smaller groundings in 132 instances, larger ones in 18, and 52 groundings of equal size. On average our groundings are 37.38% smaller with a maximum of 98.22% or 249.828 methods reduction.

For the number of abstract tasks, results are shown in Fig. 5. We find smaller groundings in 113 instances, larger ones in 80, and 8 groundings of equal size. On average our groundings are 24.02% smaller with a maximum of 98.59% or 11.980 abstract tasks reduction. As we can see from the scatter plot, if we produce a larger grounding, it is usually not significantly larger. At the maximum, our grounding contains 381 more abstract tasks due to our parameter splitting, which produces significantly fewer methods in several instances.

6 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 present our grounding procedure and discuss how to compute it efficiently. Our empirical evaluation shows that it leads to smaller groundings than related work.

Acknowledgments

This work was done within the technology transfer project “Do it yourself, but not alone: Companion-Technology for DIY support” of the Transregional Collaborative Research Centre SFB/TRR 62 “Companion-Technology for Cognitive Technical Systems” 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 *Proceedings of the 25th International*

Conference on Automated Planning and Scheduling (ICAPS 2015), 7–15. AAAI Press.

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

Alford, R.; Shivashankar, V.; Roberts, M.; Frank, J.; and Aha, D. W. 2016. Hierarchical planning: relating task and goal decomposition with task sharing. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*. AAAI Press.

Behnke, G.; Höller, D.; Bercher, P.; and Biundo, S. 2016. Change the plan – How hard can that be? In *Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS 2016)*, 38–46. AAAI Press.

Behnke, G.; Schiller, M.; Kraus, M.; Bercher, P.; Schmautz, M.; Dorna, M.; Minker, W.; Glimm, B.; and Biundo, S. 2018. Instructing novice users on how to use tools in DIY projects. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence and the 23rd European Conference on Artificial Intelligence (IJCAI-ECAI 2018)*, 5805–5807. IJCAI.

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 *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS 2015)*, 25–33. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2017. This is a solution! (... but is it though?) – Verifying solutions of hierarchical planning problems. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS 2017)*, 20–28. AAAI Press.

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

Behnke, G.; Höller, D.; and Biundo, S. 2018b. Tracking branches in trees – A propositional encoding for solving partially-ordered HTN planning problems. In *Proceedings of the 30th International Conference on Tools with Artificial Intelligence (ICTAI 2018)*, 73–80. IEEE Computer Society.

Behnke, G.; Höller, D.; and Biundo, S. 2019a. 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 2019)*. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2019b. Finding optimal solutions in HTN planning – A SAT-based approach. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*. IJCAI.

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 2019)*. IJCAI.

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 *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI 2016)*, 225–233. IOS Press.

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

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid planning heuristics based on task decomposition graphs. In *Proceedings of the 7th Annual Symposium on Combinatorial Search (SoCS 2014)*, 35–43. AAAI Press.

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 *Proceedings of the 4th Workshop on Planning and Robotics (PlanRob)*, 12–19.

Elkawkagy, M.; Bercher, P.; Schattenberg, B.; and Biundo, S. 2012. Improving hierarchical planning performance by the use of landmarks. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI 2012)*, 1763–1769. AAAI Press.

Elkawkagy, M.; Schattenberg, B.; and Biundo, S. 2010. Landmarks in hierarchical planning. In *Proceedings of the 20nd European Conference on Artificial Intelligence (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):69–93.

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

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 *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI 2014)*, volume 263, 447–452. IOS Press.

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

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, B. 2018. A generic method to guide HTN progression search with classical heuristics. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS 2018)*, 114–122. AAAI Press.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2019a. HDDL – A language to describe hierarchical planning problems. In *Proceedings of the Second ICAPS Workshop on Hierarchical Planning*.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2019b. On guiding search in HTN planning with classical planning heuristics. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*. IJCAI.

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

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.

Schreiber, D.; Balyo, T.; Pellier, D.; and Fiorino, H. 2019. Tree-REX: SAT-based tree exploration for efficient and high-quality HTN planning. In *Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS 2019)*. AAAI Press.