Journal of Artificial Intelligence Research 67 (2020) 835-880 Submitted 01/2019; published 04/2020

HTN Planning as Heuristic Progression Search

Daniel Höller DANIEL.HOELLER@ALUMNI.UNI-ULM.DE
Pascal Bercher PASCAL.BERCHER@ALUMNI.UNI-ULM.DE
Gregor Behnke GREGOR.BEHNKE@ALUMNI.UNI-ULM.DE
Susanne Biundo SUSANNE.BIUNDO@UNI-ULM.DE
Institute of Artificial Intelligence,
Ulm University, Ulm, Germany

Abstract

The majority of search-based HTN planning systems can be divided into those searching a space of partial plans (a plan space) and those performing progression search, i.e., that build the solution in a forward manner. So far, all HTN planners that guide the search by using heuristic functions are based on plan space search. Those systems represent the set of search nodes more effectively by maintaining a partial ordering between tasks, but they have only limited information about the current state during search. In this article, we propose the use of progression search as basis for heuristic HTN planning systems. Such systems can calculate their heuristics incorporating the current state, because it is tracked during search. Our contribution is the following: We introduce two novel progression algorithms that avoid unnecessary branching when the problem at hand is partially ordered and show that both are sound and complete. We show that defining systematicity is problematic for search in HTN planning, propose a definition, and show that it is fulfilled by one of our algorithms. Then, we introduce a method to apply arbitrary classical planning heuristics to guide the search in HTN planning. It relaxes the HTN planning model to a classical model that is only used for calculating heuristics. It is updated during search and used to create heuristic values that are used to guide the HTN search. We show that it can be used to create HTN heuristics with interesting theoretical properties like safety, goal-awareness, and admissibility. Our empirical evaluation shows that the resulting system outperforms the state of the art in search-based HTN planning.

1. Introduction

Hierarchical structures in planning have often been seen as *advice* that is added to a non-hierarchical model, which encodes the *physics* of the modeled system (McDermott, 2000). However, as shown by Erol, Hendler, and Nau (1996), the hierarchy enables the definition of much more complex problems and thus introduces an entire new class of planning problems. When using the representation to model such more complex behavior, the hierarchy necessarily encodes *physics* and may not encode any advice (we will discuss this in more detail in Section 2). Creating models that include not only physics but also advice increases the effort for the domain designer. But for a planning system it is harder to find solutions in a model without advice. As a consequence, a number of hierarchical planners have been introduced during the past years that use search techniques, heuristics and/or reachabil-

©2020 AI Access Foundation. All rights reserved.

ity analysis as known from classical planning or even re-use heuristic search systems from classical planning[1].

HTN planning systems can be divided into two main categories: there are *search*-based systems and *compilation*-based systems that compile the planning problem to other problem classes (Bercher, Alford, & Höller, 2019).

There are *compilations* to Answer Set Programming (Dix, Kuter, & Nau, 2003), to classical planning (Alford et al., 2009, 2016), to ConGolog and the Situation Calculus (Gabaldon, 2002; Fritz, Baier, & McIlraith, 2008), and to propositional logic (Mali & Kambhampati, 1998; Behnke, Höller, & Biundo, 2018, 2019a; Schreiber, Pellier, Fiorino, & Balyo, 2019). Translations from the undecidable HTN planning problem to a decidable problem have to bound the size of the output or assume some restrictions on the input. For example, some bound the length of solutions, some the depth of decomposition, others assume totally ordered HTN problems or a bound on the maximum size of task networks under progression – all these restrictions make the problem decidable (Erol et al., 1996; Alford, Bercher, & Aha, 2015b). When no solution is found, the bound needs to be increased.

In *search-based* HTN planning, there are essentially the same algorithms available as in non-hierarchical planning, such as local search as done by the Duet planner (Gerevini et al., 2008)[2] or state-based forward search similar to classical planning. One of the currently best-known search-based techniques does the latter. Progression-based planners generate solutions in a forward manner, i.e., they process those (abstract or primitive) tasks first that have no predecessors in the ordering. They commit to an ordering of the primitive tasks (a prefix of the generated solution) and can thus progress the current state. Yet another kind of HTN planners searches the space of partial plans (which are partially ordered sets of tasks that are not necessarily executable yet). Such planners do not commit to a prefix of a generated plan, but maintain a partial order between tasks. Since no state gets progressed, such planners have less information about the current state – all information about the planning progress is given by the current partial plan. But – due to the partial order – they can represent (the set of reachable) search nodes more compactly therefore exploring potentially smaller search spaces.

To prevent a system from doing exhaustive search, it needs to be guided by heuristics. Since the hierarchy *and* the state restrict the set of valid solutions, planners need to consider *both* when generating plans. This makes the design of heuristics difficult. Current heuristic HTN planning systems rely on plan space-based search and thus have insufficient information about the current state during search.

In this article, we propose the use of progression-based search as basis for heuristic HTN planning systems. Such systems track the current state during search and can incorporate it into their heuristic calculation. We contribute two novel progression algorithms as well as

1. See e.g. FAPE (Dvořák, Barták, Bit-Monnot, Ingrand, & Ghallab, 2014a; Dvořák, Bit-Monnot, Ingrand, & Ghallab, 2014b; Bit-Monnot, Smith, & Do, 2016), PANDA (Bercher, Keen, & Biundo, 2014; Bercher, Behnke, Höller, & Biundo, 2017), HiPOP (Bechon, Barbier, Infantes, Lesire, & Vidal, 2014), Duet (Gerevini, Kuter, Nau, Saetti, & Waisbrot, 2008), HGN-based systems (Shivashankar, Kuter, Nau, & Alford, 2012) or the translations to STRIPS and ADL (Alford, Kuter, & Nau, 2009; Alford, Behnke, Höller, Bercher, Biundo, & Aha, 2016).
2. We would like to note that while that planner makes extensive use of hierarchical concepts, it does not solve HTN problems in the typical sense, as it is allowed to insert tasks using stochastic local search. We discuss this in more detail in the next section.

a method to use classical heuristics to guide the search. More precisely, our contributions are the following:

1. The canonical progression algorithm searches parts of the search space more than once. We give two improved algorithms to avoid this. We show that they are complete, sound, and that they outperform the canonical algorithm. For one of them, we show that it explores every search node (at most) a single time, a property commonly known as systematicity.

2. We introduce a method to use arbitrary heuristics from classical planning in HTN-planning. This is done by relaxing the HTN planning problem to a classical planning problem that includes both state transition and hierarchical reachability information in its state. We show that classical heuristics computed on this heuristic model can effectively guide HTN progression search. We show that, when using classical heuristics with the respective property, the resulting HTN heuristics are safe, goal-aware, and admissible.

The first of our new algorithms and the heuristic model have been presented before in a conference paper (Höller, Bercher, Behnke, & Biundo, 2018a). Our systematic algorithm is presented for the first time in this article. In HTN planning, a systematic algorithm is especially interesting since it has been shown that checking whether a search node has been visited before is infeasible (Behnke, Höller, & Biundo, 2015). Since there is no definition of systematicity in this context, our new content includes a discussion of the issue and a proposed definition. The heuristic model is presented in a more complete way than before by introducing an elaborated example, giving an in-depth discussion of related work, and a more detailed description of the empirical results.

The next section we discuss related work; then we introduce the formal framework that we use throughout the article (Section 3) before showing how to improve the progression algorithm (Section 4). Our method to use classical heuristics in HTN planning is given in Section 5. The overall system is evaluated in Section 6.

2. Search-based HTN Planning

When modeling a certain system for planning, there is always more than one way to represent it. A central design decision is the amount of *advice* – parts of the model that help the planning system to solve the problem – introduced into the model. In PDDL, the standard representation used in classical planning, the philosophy is to model only the *physics* of the domain without any advice.

> "The PDDL language was designed to be a neutral specification of planning problems. That is, every piece of a representation would be a necessary part of the specification of what actions were possible and what their effects are. All traces of "hints" to a planning system would be eliminated."
> (McDermott, 2000, p. 3)

The amount of advice has also been used to categorize planning systems into *domain-specific*, *domain-configurable*, and *domain-independent* systems (Nau, 2007).

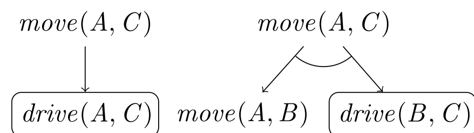

Figure 1: Domain modeling a simple transport problem via the hierarchy as introduced by Höller et al. (2018) (Copyright © 2018, IEEE).

Besides the preconditions and the effects of actions that define a state transition system, HTN planning introduces a second way to define properties of solutions. It distinguishes two kinds of tasks: primitive tasks (also called actions) can be directly executed and cause state transition like actions in classical planning. Abstract (or compound) tasks describe more abstract things to do and can *not* be executed directly. Instead, they are iteratively decomposed into subtasks (that can be either primitive or abstract) until only primitive tasks are left. The subtasks can have a partial order, i.e., the subtasks of several tasks might – in the general case – be interweaved. The overall plan must be a decomposition of the initial task(s), it is not allowed to insert tasks apart from decomposition. The aim is to find a decomposition that is executable. Usually, there is no state-based goal definition given. The decomposition forms a grammar-like structure that has vast influence on the set of solutions.

Hierarchical structures have often been regarded a means of re-introducing advice into the classical representations modeling the physics of the domain (McDermott, 2000). As a consequence, maybe the best known HTN planning system, SHOP2 (Nau, Au, Ilghami, Kuter, Murdock, Wu, & Yaman, 2003), is – in the advice-based categorization given above – characterized as domain-configurable system (Nau, 2007). Though it can be used to add advice, the hierarchy adds a new means of modeling that makes the resulting formalism more powerful than classical (non-hierarchical) planning. This can be seen from the different complexity results on the plan existence problem (Bylander, 1994; Erol et al., 1996; Geier & Bercher, 2011; Alford et al., 2015b). It is even more clear when studying which patterns of behavior can be represented in the common planning formalisms. To do this, the set of all solutions has been regarded a formal language, and the different formalisms have been classified with respect to the Chomsky hierarchy (Höller, Behnke, Bercher, & Biundo, 2014, 2016). When using standard HTN planning, a superset of the context-free languages can be expressed that contains a subset of the context-sensitive languages (Höller et al., 2014), while classical planning formalisms can express a strict subset of the regular languages (Höller et al., 2016). Whenever a problem requires plan structures similar to matching opening and closing parenthesis, it can not be modeled in classical planning anymore, but easily using a hierarchy. An example could be a domain modeling a road network containing loops where vehicles have to pay a charge for each use of a road section when they are back home.

Consider a simple transport domain with a vehicle delivering packages. In a non-hierarchical model, the position of the vehicle is stored in the state and actions change this position until the vehicle reaches the position it needs to go. Figure 1 gives a sketch of a hierarchical definition of this vehicle movement. The abstract task *move* may be decomposed by one of two methods: the left one enables movement on a direct road between

two locations using the *drive* action. The right method is recursive and enables movement between locations without a direct road. That way, the model might store no information on the current vehicle position in the state. However, after executing the task, the vehicle will be at its intended final position. In this example, the hierarchy models physics, not advice. In general, the hierarchy adds a second means of modeling that can, just like the transition system implicitly defined via the state variables and actions, be used to model physics *or* advice. Plans in HTN planning must suffice constraints introduced by hierarchy as well as state. A broader discussion of this topic can be found in the paper on plan repair by Höller, Bercher, Behnke, and Biundo (2018b).

As mentioned before, SHOP2 is a domain-*configurable* system. It performs a progression-based depth-first search and does not use heuristics to estimate goal distance. Instead, search is controlled by several additional features like state-based preconditions that determine when a decomposition method can be applied, an if-then-else structure that defines a precedence on methods, and even calls of external programs can be used. A good configuration results in a fast planning system that can be applied in large real-world domains, but it also increases the effort for the domain designer. Because there is no other guidance (e.g. by heuristics) the designer has to foresee all scenarios the system will be applied to and provide advice for it.

There have been several other approaches to control the search in hierarchical planning by using an *extended* or *modified* model. Waisbrot, Kuter, and Könik (2008) have added a goal definition to methods to calculate heuristic values on these definitions. Shivashankar et al. have introduced a new kind of hierarchical formalism that decomposes *goals* instead of *tasks* (Shivashankar et al., 2012; Alford, Shivashankar, Roberts, Frank, & Aha, 2016) and presented planning algorithms as well as heuristics for this new formalism (Shivashankar, Alford, Kuter, & Nau, 2013; Shivashankar, Alford, Roberts, & Aha, 2016; Shivashankar, Alford, & Aha, 2017). This simplifies the adaptation of classical heuristics.

Gerevini et al. (2008) have introduced an approach that interleaves HTN planning with classical planning, allowing the planner to insert tasks apart from the hierarchy like in HTN planning with task insertion (see e.g. Geier & Bercher, 2011; Alford, Bercher, & Aha, 2015a). As a consequence, they solve a different class of problems that is less expressive than standard HTN planning where task insertion is strictly prohibited (for a discussion of that feature see the survey by Bercher et al., 2019). Domain-independent (local) search techniques are used in the part applying classical planners, not in the HTN search.

There have also been attempts to improve solvers in *standard* HTN planning. Lotem, Nau, and Hendler (1999) have introduced a combination of a *planning graph* (Blum & Furst, 1997) with a structure called *decomposition graph*, combining (relaxed) state-based reachability and hierarchical reachability information. A solution is then extracted from the combined graph using a specialized algorithm (as done by Graphplan, see Blum & Furst, 1997), i.e. this approach does not use a standard search algorithm combined with a heuristic estimation. Alford et al. have introduced a translation from standard HTN planning to classical planning (Alford et al., 2009, 2016). To make this possible – translating an undecidable problem into a PSPACE-complete yet decidable problem – the translation needs a bound as input parameter. For a subset of all HTN planning problems, called *tail-recursive* HTN planning problems, a maximum upper bound can be calculated (Alford et al., 2016), i.e., when there is no solution for the translation using this bound, there is none

for the HTN planning problem at all. When applied to arbitrary HTN planning problems, the bound needs to be increased incrementally as in the translation from classical planning to SAT. However, for HTN to STRIPS translations like the one of Alford et al., there is in general no way to determine an upper limit for the bound. The problem resulting from the translation can then be solved using arbitrary classical planning systems. We will further discuss the relationship of our work to this encoding in Section 5.4 after we have introduced our heuristic model.

So far we have discussed systems that (1) solve non-standard HTN planning problems, (2) use specialized algorithms, or (3) use a translation. Apart from these, there are several systems that use standard search combined with heuristic and/or pruning techniques. However, all of the following systems search a *plan space*, not a state space. One of these systems is FAPE (see Dvořák et al., 2014a, 2014b; Bit-Monnot et al., 2016). FAPE has a slightly different focus than the other systems given here: it comes with sophisticated support for planning with time, but does not support recursion in the HTN models (Dvořák et al., 2014b). The need for non-recursive models results in a far less expressive problem class (Erol et al., 1996; Höller et al., 2014; Alford et al., 2015b). FAPE searches a plan space and combines this with a pruning technique, it does not use heuristics (Bit-Monnot et al., 2016). We will further discuss the relationship of the pruning technique to our work in Section 5.4. A second system based on standard search is PANDA (Bercher et al., 2014, 2017). PANDA combines techniques from Partial Order Causal Link (POCL) planning (Weld, 1994) with HTN planning. There are several heuristics available, all based on the *Task Decomposition Graph* (TDG), a structure similar to the decomposition graph of Lotem et al. (1999) that captures the dependencies in the decomposition hierarchy. Elkawkagy, Bercher, Schattenberg, and Biundo (2012) showed how to extract (task-)landmarks – tasks that need to be done to reach a primitive decomposition – from a TDG. More recent approaches extract heuristic values directly from the TDG (Bercher et al., 2017). This is the work with the closest connection to our heuristic calculation. The relationship will be further discussed in Section 5.4. A system that is – at least regarding its search mechanisms – quite similar to PANDA is HiPOP (Bechon et al., 2014).

So far, all *heuristic search* systems were based on a plan space search. In the following sections, we introduce a progression-based system. Here the current state is available during search and can be exploited for calculating heuristic values.

3. Formal Framework

We need to formalize both classical and hierarchical planning problems, so we use a formalization that defines HTN planning problems as extension to classical problems to have a common formalism for both. It was first introduced by Höller et al. (2016) and is based on the (pure HTN) formalism by Geier and Bercher (2011). Recent work on HTN planning has often been based on the formalism by Geier and Bercher (2011). It is much more simplistic than e.g. the one used by Erol et al. (1996), yet still capable of describing undecidable problems (Geier & Bercher, 2011). One reason for its simplicity is that it does not support as many constraint types as Erol et al.'s formalism and also restricts the ordering definition between tasks to a partial order.

There is also a lifted variant of the formalism in the literature (Alford et al., 2015b), but we use the propositional one. Since input models will often be provided in a lifted language (e.g. HDDL, see Höller, Behnke, Bercher, Biundo, Fiorino, Pellier, & Alford, 2020), we assume a grounding process prior to the actual search (like it is often used in classical planning). Little work has been done on grounding in HTN planning, but two approaches have recently been introduced by Ramoul, Pellier, Fiorino, and Pesty (2017) and by Behnke, Höller, Schmid, Bercher, and Biundo (2020).

3.1 Classical Planning

A classical planning problem is a tuple $P_c = (L, A, s_0, g, \delta)$, where L is a set of propositional environment facts, $s_0 \subseteq L$ defines the initial state and $g \subseteq L$ gives the goal definition. A is a set of action names. These are symbols that are mapped to preconditions and effects by the functions contained in $\delta = (\mathit{prec}, \mathit{add}, \mathit{del})$. All these functions map an action name to a subset of the environment facts $\{\mathit{prec}, \mathit{add}, \mathit{del}\} : A \to 2^L$.

A state is given by the set of facts that hold in that state, all other facts are supposed to be false (closed world assumption). Whether an action a is applicable in a state s is given by the relation $\tau \subseteq A \times 2^L$ with $\tau(a, s) \Leftrightarrow \mathit{prec}(a) \subseteq s$. When it is applicable (i.e., $\tau(a, s)$ holds) the state resulting from the application is given by the function $\gamma : A \times 2^L \to 2^L$ with $\gamma(a, s) = (s \setminus \mathit{del}(a)) \cup \mathit{add}(a)$. γ is called the state transition function.

A sequence of actions $\langle a_0 a_1 \dots a_l \rangle$ with $a_i \in A$ is applicable in a state s_0 if and only if a_i is applicable in s_i (i.e., $\tau(a_i, s_i)$ holds), where s_i is defined as $\gamma(a_{i-1}, s_{i-1})$ for $i > 0$. We will say that the state s_{l+1} results from the application. A solution to a classical planning problem is a sequence $\langle a_0 a_1 \dots a_l \rangle$ that is applicable in s_0 and that results in a state s_{l+1} that is a goal state, i.e., $s_{l+1} \supseteq g$.

3.2 HTN Planning

An HTN planning problem is defined as a tuple $P = (L, C, A, M, s_0, tn_I, g, \delta)$, where L, A, s_0, g, and δ are defined as in classical planning. C is the set of abstract task names, symbols representing abstract courses of action. It is disjunct with the primitive task names $C \cap A = \emptyset$. Tasks are organized in *task networks* $tn = (T, \prec, \alpha)$, where T is a (possibly empty) set of (task) identifiers (ids). Ids are symbols that are mapped to task names by the function $\alpha : T \to A \cup C$. This definition allows a task (name) to be contained in a task network more than once. $\prec \subseteq T \times T$ is a strict (i.e. irreflexive, transitive and asymmetric) partial order on the ids.

Two task networks $tn = (T, \prec, \alpha)$ and $tn' = (T', \prec', \alpha')$ that differ only in their identifiers are *isomorphic* (denoted $tn \cong tn'$). Formally, there must be a bijection $\sigma : T \to T'$ such that $\forall t, t' \in T : [(t, t') \in \prec] \Leftrightarrow [(\sigma(t), \sigma(t')) \in \prec']$ and $\alpha(t) = \alpha'(\sigma(t))$.

Decomposition is defined by (decomposition) *methods*. A method is a pair (c, tn), where $c \in C$ is an abstract task name and tn a task network (also called the method's subtask network). When a task t with $\alpha(t) = c$ is decomposed using a method (c, tn), it is removed from the task network, the tasks in the subtask network are added and inherit the ordering constraints from t. Formally, a method (c, tn) decomposes a task network $tn_1 = (T_1, \prec_1, \alpha_1)$ into a task network $tn_2 = (T_2, \prec_2, \alpha_2)$ if $t \in T_1$ with $\alpha_1(t) = c$ and there is a task network

$tn' = (T', \prec', \alpha')$ with $tn' \cong tn$ and $T_1 \cap T' = \emptyset$. The task network tn_2 is defined as follows:

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

M is the set of decomposition methods. We will write $tn \xrightarrow[t,m]{} tn'$ to denote that tn can be decomposed into a task network tn' by applying the method m to the task t, and $tn \rightarrow^* tn'$ if it is possible to decompose tn into tn' using a sequence of methods.

Decomposition starts either with a single initial task or with a task network (that may include more than one task). A definition using a single initial task is sometimes beneficial to keep proofs simple. Using an initial task *network* makes it possible to interpret every search node in the search space as a new planning problem (Alford, Shivashankar, Kuter, & Nau, 2012). However, both definitions are equivalent: Given an initial task network tn_I, it can be compiled away by introducing a new task c_I (that is the new initial task) and a method m that decomposes the new task into the original task network $m = (c_I, tn_I)$. Throughout the article, we will use the variant that is most convenient for the definition at hand. The definition given above uses an initial task network tn_I.

A task network $tn = (T, \prec, \alpha)$ is a solution to a given HTN planning problem $P = (L, C, A, M, s_0, tn_I, g, \delta)$ if and only if the following solution criteria hold:

- $tn_I \rightarrow^* tn$, i.e. it can be reached by decomposing the initial task network.
- $\forall t \in T : \alpha(t) \in A$, i.e. all task names are primitive.
- There is a sequence $\langle t_1 t_2 \ldots t_n \rangle$ of the tasks in T that is in line with $\prec$, i.e. $\forall i < j : (t_j, t_i) \notin \prec$, and the application of $\langle \alpha(t_1)\alpha(t_2) \ldots \alpha(t_n) \rangle$ in s_0 results in a goal state.

The last solution criterion requires an HTN solution to result in a goal state. This is quite unusual, since most hierarchical formalisms lack a (state-based) goal definition (though some have one, e.g. these introduced by Bercher, Höller, Behnke, & Biundo, 2016, or by Bercher et al., 2017). This is because it is easy to compile a state-based goal into the hierarchy. The approach presented in this article smoothly combines task decomposition with state-based goals, so we decided to include a state-based goal. However, be aware that our approach works perfectly without state-based goal (in fact, most domains in the evaluation do not include one).

4. Progression Search in HTN Planning

In this section we introduce the canonical progression algorithm that is given in Algorithm 1. It is equal to those known from the literature (see e.g. those introduced by Ghallab, Nau, & Traverso, 2004, p. 243; or Alford et al., 2012, p. 5). We show that it searches parts of the search space more than once. We then introduce a progression algorithm (Algorithm 2) from previous work (Höller et al., 2018a) that prevents this behavior to a certain extent but is still not systematic. Finally we introduce a novel algorithm (Algorithm 3) first presented here that is systematic. We prove completeness and soundness of Algorithm 2 and 3 and systematicity for Algorithm 3.

```
1  fringe ← {(s_0, tn_I, ε)}
2  while fringe ≠ ∅ do
3  |   n ← fringe.pop()
4  |   if n.isgoal then return n
5  |   U ← n.unconstrainedNodes
6  |   for t ∈ U do
7  |   |   if isPrimitive(t) then
8  |   |   |   if isApplicable(t) then
9  |   |   |   |   n' ← n.apply(t)
10 |   |   |   |   fringe.add(n')
11 |   |   else
12 |   |   |   for m ∈ t.methods do
13 |   |   |   |   n' ← n.decompose(t, m)
14 |   |   |   |   fringe.add(n')
15 return unsolvable
```

Algorithm 1: The canonical progression-based HTN planning algorithm as grounded, fringe-based formulation as given by Höller et al. (2018a).

4.1 Algorithms

First we introduce the canonical progression algorithm. It includes multiple nondeterministic choice points. When it is implemented, they can be realized by using recursion or a fringe. A fringe-based formulation enables a simple realization of different search strategies by replacing the data-structure underlying the fringe. We use such a fringe-based formulation that is depicted in Algorithm 1. It is now discussed in detail.

A search node is composed of three elements:

1. The current state.
2. A task network holding the tasks that still need to be processed.
3. We additionally maintain the sequence of actions progressed so far, this is the prefix of the generated solution.

The fringe is initialized with the triple containing the initial state, the initial task network, and – since there are no progressed actions so far – an empty sequence (s_0, tn_I, ε). While the fringe is not empty, a search node $n = (s, (T, \prec, \alpha), \pi)$ is removed (popped) from it (Line 3). The order in which the nodes are returned determines the search behavior (such as Depth First Search). We will have a short discussion on that later on page 847. Next, it is tested if the popped node is a solution (Line 4).

Definition 1 (Solution)**.** *A node* $n = (s, (T, \prec, \alpha), \pi)$ *is a solution (also called goal node) if and only if its task network is empty* ($T = \emptyset$) *and its state is a goal state* ($s \supseteq g$)*.*

By performing the goal test *after* popping a node from the fringe, it is – in combination with a certain ordering of the fringe, e.g. one that results in a Uniform Cost Search – possible

to find optimal solutions. If this is not important, the goal test might be done *before* adding a node to the fringe (before Line 10 and Line 14). This might result in finding a solution in less time.

A central characteristic of progression search is that only those tasks are processed that have no predecessors in the network, the *unconstrained* tasks (denoted U in the algorithm, Line 5).

Definition 2 (Unconstrained Tasks)**.** *A task id t is* unconstrained *in a task network $(T, \prec, \alpha)$ if and only if there is no task $t' \in T$ such that $(t', t) \in \prec$.*

We divided the task *names* in a domain into primitive and abstract tasks. The elements in U are task *ids* (no task names). By abuse of notation, we will denote an id as primitive (Line 7) if it is mapped to an action, or call it abstract, otherwise. In the same way, we will call a primitive id *applicable* if the corresponding task is applicable in the corresponding search node's state. For unconstrained (grounded) actions, there is no (when it is not applicable) or exactly one (when it is applicable) possible modification: progressing the action and updating the state. The resulting node is inserted into the fringe (Line 10).

Definition 3 (Progression)**.** *Given a search node $n = (s, (T, \prec, \alpha), \pi)$ and an unconstrained task id $t \in T$ that is mapped to an applicable action, the search node n' that results from progressing t is defined as follows (where $\circ$ is the concatenation operator):*

$$\begin{aligned} n' = (s', (T', \prec', \alpha'), \pi') \text{ with} \\ & s' = \gamma(\alpha(t), s),\ T' = T \setminus \{t\}, \\ & \prec' = \prec \setminus \{(t, t') \mid t' \in T\} \\ & \alpha' = \alpha \setminus \{t \mapsto \alpha(t)\},\ \text{and} \\ & \pi' = \pi \circ \alpha(t). \end{aligned}$$

For unconstrained *abstract* tasks $t \in U$, a *set* of new search nodes is generated, one for every method that is applicable to t (this is done in the loop in Line 12).

Definition 4 (Decomposition)**.** *Given a search node $n = (s, (T, \prec, \alpha), \pi)$, an unconstrained task id $t \in T$ with $\alpha(t) = c$, and a method (c, tn), the search node n' that results from decomposing t is defined as $n' = (s, tn', \pi)$ with $tn \xrightarrow[t,m]{} tn'$.*

4.1.1 Towards Systematic Progression Search

It is known that an HTN planning problem is solvable if and only if there is a solution in progression space (Alford et al., 2012, Thm. 3). However, the given algorithm searches parts of the search space more than once. An example is given in Figure 2. Every node contains the tasks that need to be decomposed as well as the prefix progressed so far (starting with $\pi = \dots$). The initial task network contains the two unordered abstract tasks Y and Z. There is a method decomposing Y into the action a, and a method decomposing Z into b. Both primitive tasks have neither preconditions nor effects. The figure depicts the entire search space that will be explored by Algorithm 1. It can be seen that every solution is found three times. This is caused by branching over primitive as well as abstract tasks, in

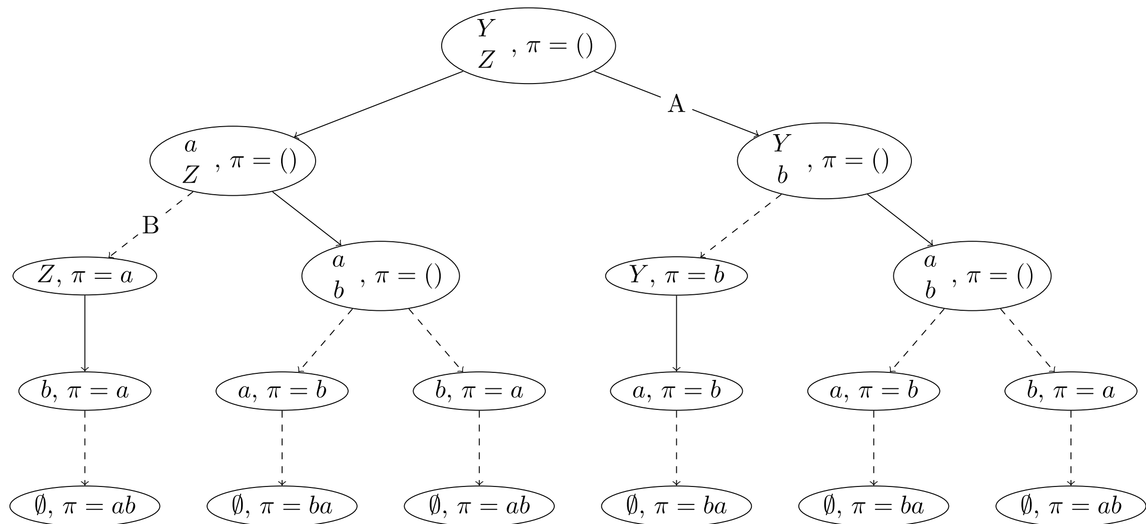

Figure 2: The search space of Algorithm 1 on a small HTN problem. Capital letters represent abstract tasks, lower case letters represent actions. The initial task network and all methods are totally unordered. In each node, tasks that need to be processed are given on the left, the prefix of the generated plan π on the right. The arrows indicate the application of a method, the dashed arrows the progression of an action. We will come back to the edges labeled with A and B later in Section 4.1.1 and 4.1.2.

```
1  fringe ← {(s_0, tn_I, ε)}
2  while fringe ≠ ∅ do
3  |  n ← fringe.pop()
4  |  if n.isgoal then return n
5  |  (U_C, U_A) ← n.unconstrainedNodes
6  |  for t ∈ U_A do
7  |  |  if isApplicable(t) then
8  |  |  |  n' ← n.apply(t)
9  |  |  |  fringe.add(n')
10 |  t ← selectAbstractTask(U_C)
11 |  for m ∈ t.methods do
12 |  |  n' ← n.decompose(t, m)
13 |  |  fringe.add(n')
14 return unsolvable
```

Algorithm 2: Optimized algorithm presented by Höller et al. (2018a).

combination with the partial ordering of Y and Z. So the question is: When is branching, i.e. a nondeterministic choice, actually needed?

To find all possible plans, the algorithm only needs to branch when a commitment to the plan is made. This is the case for every progression of an action. However, the order in which two tasks are decomposed implies no commitment to the solution (only the choice *by which method* they are decomposed does). So the algorithm does not need to branch over the available abstract tasks (but only over the methods for a chosen task to decompose). This was already exploited by Erol, Hendler, and Nau (1994, p. 28) for the plan space-based planner UMCP (Fig. 7, p. 17), where an abstract task is picked rather than being branched over. This is a standard behavior in plan space-based planning, such as also done, for example, by PANDA (Bercher et al., 2014, 2017) and FAPE (Dvořák et al., 2014a, 2014b; Bit-Monnot et al., 2016). Most plan space-based planners (like PANDA and FAPE) usually rely on techniques based on POCL planning (Weld, 1994), where algorithms need to pick *flaws* (syntactical representations of plan elements that prevent a plan from being a solution) in order to resolve them thereby refining the partial plan until it is turned into a solution. Picking an abstract task to decompose (rather than branching over them) is similar to the (non-branching) flaw selection in POCL planning (Williamson & Hanks, 1996).

In previous work (Höller et al., 2018a), we introduced Algorithm 2. The function that returns the unconstrained tasks (Line 5) now returns two sets: U_C contains the abstract and U_A contains the primitive tasks. Instead of branching over the unconstrained abstract tasks, it *picks* a single abstract task (Line 10) and branches over all applicable methods (Line 11). We have shown its soundness, completeness, and empirical efficiency gain (Höller et al., 2018a). When we come back to Figure 2, the new algorithm will reduce the part of the search space that is searched more than once. In the initial node, it selects a single abstract task and decomposes it. WLOG, we assume it picks Y. Then, the right part beginning

```
1  fringe ← {(s_0, tn_I, ε)}
2  while fringe ≠ ∅ do
3  |  n ← fringe.pop()
4  |  if n.isgoal then return n
5  |  (U_C, U_A) ← n.unconstrainedNodes
6  |  if U_C = ∅ then
7  |  |  for t ∈ U_A do
8  |  |  |  if isApplicable(t) then
9  |  |  |  |  n' ← n.apply(t)
10 |  |  |  |  fringe.add(n')
11 |  else
12 |  |  t ← selectAbstractTask(U_C)
13 |  |  for m ∈ t.methods do
14 |  |  |  n' ← n.decompose(t, m)
15 |  |  |  fringe.add(n')
16 return unsolvable
```

Algorithm 3: Novel progression algorithm.

with the edge that is labeled with an A is not explored. Though it searches a significantly smaller search space, it can be seen that there are still parts searched twice.

4.1.2 A Systematic Progression Algorithm

Here we present a progression algorithm that is still sound and complete, but it searches every part of the search space exactly once. It is given in Algorithm 3. In search nodes that contain at least one unconstrained abstract task, it does not progress any action, but first decomposes one of the abstract tasks. When we come back to Figure 2, the new algorithm "prunes" not only the edge labeled with an A, but also the one labeled B, and generates both solutions only once. Please be aware that it does *not* decompose *all* abstract tasks before progressing any action, but the *unconstrained* abstract tasks. It highly depends on the ordering of the problem at hand how many there are.

Before we come to the theoretical properties of the algorithms, we want to point out two lines of the algorithms that determine their behavior:

1. In Line 3 of the three algorithms, the data structure used as fringe determines the search behavior of the algorithm, e.g. a stack results in a Depth First Search, a queue in a Breadth First Search. Since we want to do heuristic search, we will mainly use a priority queue combined with different heuristics and weights to get the standard strategies like Greedy Best-First search, A* search, or Weighted A* search. Our heuristics are discussed in Section 5.

2. In Line 10 of Algorithm 2 and Line 12 of Algorithm 3, the algorithms have to pick a single abstract task that is decomposed. As argued before, the choice of the abstract task does not affect the success or failure of the search, but only its efficiency.

4.2 Formal Properties

For the progression algorithm given in Algorithm 1 it is known that it is sound and complete (Ghallab et al., 2004, p. 243). This means that the *soundness* of our two improved algorithms Algorithm 2 and Algorithm 3 follows directly. However, there is no formal proof published for that, so we provide self-contained proofs for soundness and completeness of our algorithms.

In the following proofs, we need to show that a certain task network can be reached via decomposition of the initial task network. This can be done by showing that there is a valid *decomposition tree* (DT) for it (Geier & Bercher, 2011, Prop. 1). Please be aware that none of the algorithms actually builds the tree, it is only shown in the following proofs that this is possible. DTs are tree representations of sequences of method applications and the resulting task network (Ghallab et al., 2004). We use the formalization by Geier and Bercher (2011, Def. 7-9):

Definition 5 (Decomposition Tree)**.** *Given an HTN planning problem $P = (L, C, A, M, s_0, tn_I, g, \delta)$, a Decomposition Tree (DT) is a tuple $g = (T_g, E_g, \prec_g, \alpha_g, \beta_g)$ with the following elements:*

1. *T_g and E_g are nodes and (directed) edges that form a tree.*
2. *$\prec_g$ is a strict partial order on its nodes.*
3. *The function α_g labels each node of the tree with a task name: $\alpha_g : T_g \to C \cup A$.*
4. *The nodes $N_C = \{n \mid \alpha_g(n) \in C\}$ are additionally labeled with methods that are assigned by the function $\beta_g : N_C \to M$.*

Definition 6 (Valid DTs)**.** *A DT is* valid *if and only if its root is labeled with the problem's initial task*[3] *and for any node $t \in N_C$ with $\beta_g(t) = (c, tn)$, the following conditions hold:*

1. *$\alpha_g(t) = c$ the node is mapped to the task decomposed by the method.*
2. *The task network induced in g by the children of t are isomorphic to tn.*
3. *Let $ch(g, t)$ be the children of t in g. For all $t' \in T_g$ and $c' \in ch(g, t)$, it holds that*
   (a) *if $(t, t') \in \prec_g$, then $(c', t') \in \prec_g$, and*
   (b) *if $(t', t) \in \prec_g$, then $(t', c') \in \prec_g$*

   I.e., ordering constraints between the children of t in g are inherited as defined for decomposition.
4. *$\prec_g$ includes only ordering constraints necessary due to 2 or 3.*

Definition 5 specifies the syntactical representation of decomposition trees, Definition 6 specifies the semantics by connecting a DT to a certain planning problem. We had to adapt the definitions to allow for empty subtask networks of methods (a feature not supported by

3. Please be aware of the equivalence of HTN planning problems with initial *task* and initial *task network*, which has been discussed in Section 3.2 on page 842.

Geier & Bercher, 2011)[4]. In the rest of this article, we will only discuss *valid* trees and – to improve readability – therefore just write *decomposition tree*.

Definition 7 (Yield of DTs). *The yield of a DT is a task network* $tn = (T, \prec, \alpha)$ *that contains tasks represented by the nodes of the tree that are mapped to primitive tasks:* $T = \{t_n \mid n \in N_P\}$ *and* $\alpha = \{t_n \mapsto \alpha_g(n) \mid n \in N_P\}$ *with* $N_P = \{n \mid n \in T_g, \alpha_g(n) \in A\}$. *Let* $\prec'_g$ *be the transitive closure of* $\prec_g$, *then* $\prec = \{(t_n, t_m) \mid n, m \in N_P \text{ and } (n, m) \in \prec'_g\}$.

A DT is a witness for a valid decomposition leading to a task network, i.e., for a given planning problem, there is a valid decomposition tree g with $yield(g) = tn$ if and only if $tn_I \rightarrow^* tn$ (Geier & Bercher, 2011, Prop. 1).

We call two DTs to be *isomorphic* if they only differ in their identifiers:

Definition 8 (Isomorphic DTs). *Two DTs* $g = (T_g, E_g, \prec_g, \alpha_g, \beta_g)$ *and* $g' = (T'_g, E'_g, \prec'_g, \alpha'_g, \beta'_g)$ *are called to be* isomorphic *(denoted* $g \cong g'$*) if and only if there is a bijection* $\sigma : T_g \rightarrow T'_g$ *such that*

- $\forall t, t' \in T_g : [(t, t') \in E_g] \Leftrightarrow [(\sigma(t), \sigma(t')) \in E'_g]$,
- $\forall t, t' \in T_g : [(t, t') \in \prec_g] \Leftrightarrow [(\sigma(t), \sigma(t')) \in \prec'_g]$,
- $\alpha_g(t) = \alpha'_g(\sigma(t))$,
- $\beta_g(t)$ *is defined if and only if* $\beta'_g(\sigma(t))$ *is defined, and*
- $\beta_g(t) = \beta'_g(\sigma(t))$.

4.2.1 Soundness

Now we can show that every path in the search space of the algorithm corresponds to a valid decomposition of the initial task, i.e. to the construction of a valid DT and that nodes that are returned also fulfill the other solution criteria that are concerned with executability and generation of a goal state.

The steps in the algorithms that change search nodes, i.e. those that step through the search space, are decomposition and progression. They are equal for the three algorithms, so the following proof holds for all of them.

Theorem 1. *The algorithms given in Algorithm 1, 2, and 3 are sound.*

Proof. The tree can be maintained during search as follows: For the initial task, the tree is defined as $g = (\{t\}, \emptyset, \emptyset, \{t \mapsto c_I\}, \emptyset)$, where c_I is the initial task. For every decomposition, a new tree is generated as follows: Let $g = (T_g, E_g, \prec_g, \alpha_g, \beta_g)$ be the tree of the current search node $(s, (T, \prec, \alpha), \pi)$. When a task $t \in T$, $\alpha(t) = c$ is decomposed using the method $m = (c, (T_m, \prec_m, \alpha_m))$ the resulting tree of the new search node is defined as follows (where $(T'_m, \prec'_m, \alpha'_m)$ is an isomorphic copy of the subtask network of m with $T_m \cap T'_m = \emptyset$): $g = (T_g \cup T'_m,\ E_g \cup \{(t, t') \mid t' \in T'_m\},\ \prec'_g,\ \alpha_g \cup \alpha'_m,\ \beta_g \cup \{t \mapsto m)\})$. The ordering relation

4. Geier and Bercher (2011) assume that exactly the leafs of the tree are mapped to primitive tasks, this is not the case when there are abstract tasks that are decomposed using a method without subtasks. So we have changed the definitions.

$\prec'_g$ is maintained by adding new relations according to the definition of task decomposition to those of the current tree. In progression steps, the tree remains the same as in the parent node, i.e. progression does not change the DT.

When the algorithms return a node as solution the corresponding tree constructed according to that definition has the following properties:

- It has c_I as its root node,
- its mappings α_g and β_g are consistent, i.e., for each node n it holds that $\beta_g(n) = (\alpha_g(n), tn)$,
- the children of a node n match its method $\beta_g(n)$.

The latter property follows from the fact that if any algorithm returns a solution, it has reached an empty task network (cf. Definition 1), i.e. it has decomposed every abstract task in the constructed tree. The order $\prec_g$ of the tree is also correct with respect to the applied methods. Further, the ordering constraints that are present in the tree have been inserted into the task networks contained in the search nodes. They were inherited according to the definition of decomposition (cf. Section 3.2) and are only removed once a task to which they pertain is progressed.

What remains to be shown is that the yield of the tree g is executable. First, note that a primitive task can only be inserted into the task network of a search node via decomposition. Thus it is also contained in the yield of g. Similarly, every task in the yield was at one point inserted into the current task network of a search node. Since, whenever any of the algorithms returns a solution, the current task network is empty (cf. Definition 1), all of the primitive tasks in the yield will have been progressed at some point. Let the order in which the primitive tasks have been progressed be π. Since π was progressed successfully, it is an executable sequence of the actions in the yield of g. Lastly, we have to show that this order is a valid linearization of the yield of g. Assume that it is not. Then two primitive tasks $p_i, p_j \in \pi$ where p_i occurs before p_j exist such that the yield of g contains the order $p_j \prec p_i$. Further, the algorithm progressed p_i before it progressed p_j. At the time at which p_i was progressed, either p_j or any of its ancestors in g must still have been in the search node's current task network. Since the forced order between p_i and p_j in the yield of g can only have been inserted by the decomposition method applied to their last common ancestor in g (Behnke, Höller, & Biundo, 2019b) and these methods must have already been applied, as p_i is already in the task network, this order is still present. Thus p_i was not an unconstrained task when it was decomposed, which is a contradiction. Thus π is a valid linearization of the order of the yield of g, and thus g is a valid witness that the constructed plan π is a solution. □

4.2.2 Completeness

Each of the three algorithms is compatible with multiple different search strategies, such as Depth First Search (DFS), Breadth First Search (BFS), Uniform Cost Search, and heuristic search strategies like A* or Weighted A* search. Technically, the concrete strategy is determined by the behavior of the fringe.

The different strategies will lead to different properties regarding completeness. Depending on the given input model, a strategy like DFS might often fail to find a solution

(even though there is one). The following theorem will only hold for certain search strategies, because it must be guaranteed that a node that is put into the fringe is at some point given back for further expansion. We therefore assume the system to do a BFS.

Theorem 2. *The algorithms given in Algorithm 1, 2, and 3 are complete.*

We prove the theorem for Algorithm 3. Since Algorithm 2 visits a subset of the search nodes visited by Algorithm 1, and Algorithm 3 a subset of Algorithm 2, their completeness follows directly.

Proof. Let tn_S be an arbitrary but fixed solution of a planning problem at hand. We know that there is a valid decomposition tree $g = (T_g, E_g, \prec_g, \alpha_g, \beta_g)$ that resulted in tn_S, i.e. $\mathit{yield}(g) = tn_S$. As given in the proof of Theorem 1, every path in the search space corresponds to a DT. We need to show that Algorithm 3 will eventually explore a path that corresponds to a tree isomorphic to g.

The algorithm uses two types of modification: decomposition and progression. But while there are unconstrained abstract tasks left, it does not use progression. When no unconstrained *abstract* task is left, i.e. all unconstrained tasks are primitive, the algorithm progresses a primitive task, which is added to the prefix of the solution. This procedure is continued until a solution has been found.

The search starts with a single node containing the initial task c_I that is also the label of the root node of g. While there is at least one unconstrained abstract task, Algorithm 3 selects one of them (Line 12). Let t be that task. The algorithm branches and applies all methods (one per branch) that are applicable to that task (Line 14). This means that there must be a resulting search node that is added to the fringe that resulted from applying the method $\beta_g(t)$ used in the given solution (and since a BFS is used, this node will eventually be processed). Since t is decomposed by the same method as in the given solution, the tasks added by the decomposition are also the same as the subtasks in g.

Eventually, all unconstrained *abstract* tasks will be processed and the algorithm needs to progress primitive tasks from the network. Since we know that tn_S is a solution, there must be a sequence π_S of its primitive tasks that is applicable and is in line with the ordering constraints introduced by the methods in β_g. Since there is a branch where the algorithm applied the same methods as given in the tree, there must be a search node where it ended up with the same set of tasks. Since no further decomposition is possible (i.e. the set of tasks available for progression will not increase anymore), the first task of the original solution must be in the set of available tasks. There is a prefix of one or more action(s) that equals the prefix of the solution π_S, and the algorithm can progress it. The algorithm will branch over the actions available for branching, i.e. all possible orderings are applied, and the one of the given solution will be included. After progressing the first action, one of the following cases holds:

1. There might be a newly unconstrained abstract task that had been ordered after the progressed task and the algorithm will start as given above.

2. Another progression is needed to make more abstract tasks unconstrained.

3. The given solution might be fully reconstructed, then the task network is empty, the algorithm will have processed all tasks using the same methods as in g, and it will

have progressed the primitive tasks in the same order as in π_S, and thus a goal state will be reached. □

4.2.3 Systematicity

From the small example given in Figure 2 we know that the Algorithms 1 and 2 search parts of the search space more than once. In this section, we show that this is not the case for Algorithm 3. This property is known as *systematicity* (Kambhampati, Knoblock, & Yang, 1995). In later work, this property was also discussed (and ensured) in the context of hierarchical planning (Kambhampati, Mali, & Srivastava, 1998), but in a setting different from typical HTN planning, because no initial task network is provided and instead the respective algorithm is allowed to insert primitive and abstract tasks. In their formalism, abstract tasks also possess preconditions and effects for this purpose. We first discuss some issues with defining systematicity in HTN planning and then propose a definition.

Consider a second example domain with two abstract tasks W and X and two primitive tasks c and d. W and X are included in the initial task network (and are not ordered with respect to each other). There are two methods for each abstract task: one decomposing it into c and one decomposing it into d. The preconditions and effects of the primitive tasks c and d enforce that there must always be a c before d is executable. By decomposing W into c and X into d, we get (due to the preconditions/effects) a single solution cd. By decomposing W into d and X into c, we get, again, the same solution cd.

Such *symmetry* in a model leads to solution (multi-)sets that contain a single sequence of actions more than once. The decomposition steps leading to duplicates may, of course, be less obvious than in this example. Due to the solution criteria, systems are forced to process the tasks in a task network and using only a subset of all decomposition methods may lead to incomplete algorithms. In such cases all algorithms given here (including Algorithm 3) will generate duplicate solutions.

However, this is *not* what we have seen in the first example given in Figure 2. There, the same task was decomposed multiple times, i.e. the same search node has been explored more than once. So how can we distinguish the two cases (symmetry in the model versus search nodes reached multiple times)? This can be done by taking the decomposition tree leading to the solution into account.

Figure 3 shows our initial example, but now the search nodes are not represented by the task network and the plan prefix (like in Figure 2), but by representations of the corresponding decomposition tree. In each depicted decomposition tree, the nodes are labeled with tasks and the edges leading from an abstract task to another task are labeled with the applied method. When an action is progressed, it is labeled with its index in the generated solution.

We will denote a decomposition tree where primitive tasks may be labeled with their position in an executable task sequence *position-labeled DT*. This representation includes witnesses for our HTN solution criteria: the tree is a proof (i.e., a witness) that there is a decomposition leading to a certain set of actions, the position labels indicate the correct execution ordering.

Definition 9 (Valid Position-Labeled DTs)**.** *A position-labeled DT is a tuple* $g = (T_g, E_g, \prec_g, \alpha_g, \beta_g, \gamma_g)$ *where* $(T_g, E_g, \prec_g, \alpha_g, \beta_g)$ *forms a valid Decomposition Tree and* γ_g *is a*

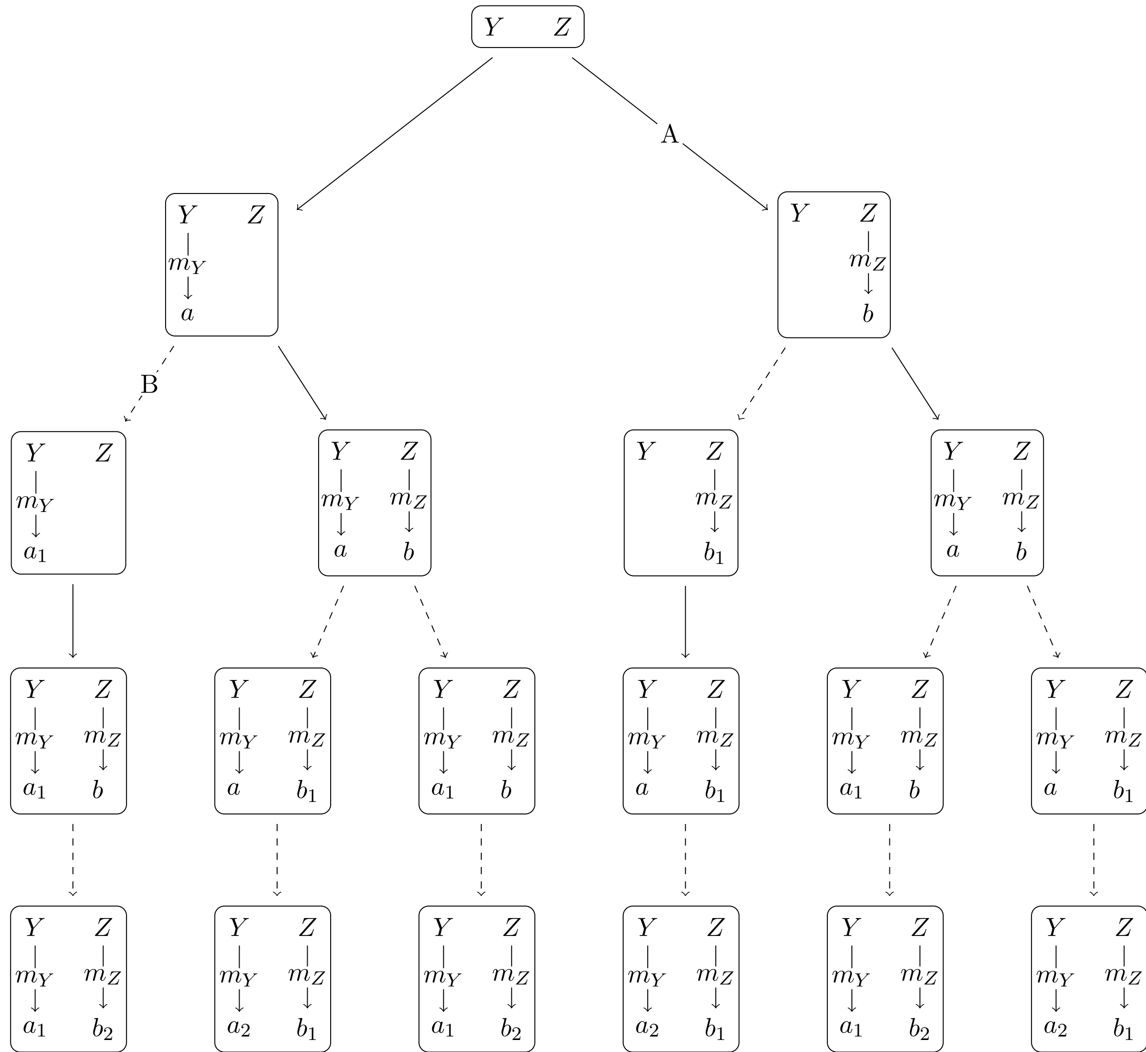

Figure 3: Example given in Figure 2, but now each node shows the decomposition tree generated so far. When an action has already been progressed, it is marked with its index in the solution prefix (e.g., a_1 indicates that a is the first action in the solution).

surjective function that maps the set $N_P = \{n \mid \alpha_g(n) \in A\}$ *of nodes to a number out of* $\{1 \ldots |N_P|\}$.

Every search node represents a certain set of position-labeled DTs – exactly those that can be generated from it. Analog to Definition 6 of Kambhampati et al. (1995) for non-hierarchical planning, we propose the following definition of systematicity in HTN planning:

Definition 10 (Systematicity in HTN Planning)**.** *An HTN search is* systematic *if and only if for any two nodes* n_1 *and* n_2 *such that* n_1 *is no ancestor of* n_2 *and* n_2 *is no ancestor of* n_1 *in the search tree, there is no pair of position-labeled DTs* pdt_1 *and* pdt_2 *with*

- $pdt_1 \cong pdt_2$ *(*pdt_1 *is isomorphic to* pdt_2*),*
- pdt_1 *is in the set of position-labeled DTs represented by* n_1*, and*
- pdt_2 *is in the set of position-labeled DTs represented by* n_2*.*

WLOG, we make the following assumptions about the input HTN problem:

Assumption 1 (No Isomorphic Methods)**.** *The model does not contain two distinct methods* (c, tn_1) *and* (c, tn_2) *with* $tn_1 \cong tn_2$*.*

Assumption 2 (No Equal Subtasks)**.** *There is no method* $(c, (T, \prec, \alpha))$ *with* $t_1, t_2 \in T$*,* $t_1 \neq t_2$*, and* $\alpha(t_1) = \alpha(t_2)$[5]*.*

Given the preceding assumptions, the following theorem holds:

Theorem 3. *The search defined by Algorithm 3 is systematic according to Definition 10.*

Proof. Given an arbitrary search node n, we show the following two points:

1. The position-labeled DTs represented by its children are more constrained than the position-labeled DT belonging to n.
2. The sets of position-labeled DTs belonging to search nodes with the parent node n are disjunct.

Transitions between a search node and its children can be caused by two modifications: (a) decomposition or (b) progression. Both commit to an element in the represented position-labeled DTs. In case (a), a method label is added to an abstract task and child nodes are added. In case (b), some primitive task is labeled with an index. These are the only possible modifications – there is no way to remove some element from a position-labeled DT. This shows point 1.

The sets of position-labeled DTs belonging to search nodes with the same parent node are disjunct. A new search node may have been generated by a decomposition or the progression of an action. When the task network of a given search node n includes at least

5. This assumption is made to have at least one criterion to distinguish two subtrees in the overall tree. If there is such a method $(c, (T, \prec, \alpha))$ with two task ids t_1 and t_2 mapped to the same task name c_1, one could make them distinguishable, e.g. by compiling the planning problem into a new one by introducing a new task name c_2, changing the mapping of t_1 to c_2, and introducing a new method $(c_2, (\{t\}, \emptyset, \{(t \mapsto c_1)\}))$ that decomposes the new task c_2 back into c_1.

one unconstrained abstract task, Algorithm 3 does not progress a task, but selects one abstract task that is decomposed. When there is no abstract task left, it iterates over the unconstrained actions and progresses all of them. I.e. children of a single search node are either

1. decompositions of the same abstract task using different methods applicable to that particular task, or
2. progressions of different actions in the current task network.

In the first case, we know that the position-labeled DTs represented by the nodes differ due to Assumption 1.

In the second case, some action is labeled with a position in the solution. Nodes that are siblings are labeled with the same position in the prefix. When the primitive task names that are progressed differ, the represented position-labeled DTs can be distinguished by the task name. However, a certain task network may include the same action more than once. So there may be siblings that annotate tasks with the same primitive task name with the same position in the prefix. The question is now if such siblings can result in the same position-labeled DT. The answer is no. To have such a situation, it is necessary that the subtrees that lead to the equal task names are isomorphic. But due to Assumption 2, we know that they differ at least in one position: in the layer directly below the least common ancestor. The least common ancestor is an abstract task that was decomposed into (at least) two tasks that are the root nodes of the subtrees we want to distinguish. Assumption 2 assures that these root nodes are different, i.e. we can distinguish the two subtrees and the position-labeled DTs represented by these siblings are disjunct.

Taking (1) and (2) together, we have shown that the position-labeled DTs represented by the children of a search node are more constrained than the position-labeled DTs represented by the node itself and that the sets of position-labeled DTs represented by siblings are disjunct. From that, it follows that Theorem 3 holds. □

4.3 Discussion

We have seen that the three algorithms differ in their behavior when there is more than one unconstrained task, i.e. more than one option for which task is processed next.

- Algorithm 1 branches over all unconstrained primitive and abstract tasks, and – for the latter – over applicable decomposition methods.
- Algorithm 2 branches over all unconstrained primitive tasks and a *single* (unconstrained) abstract task to decompose. For the latter it branches over the applicable decomposition methods.
- Algorithm 3 first decomposes a single (unconstrained) abstract task, branching over applicable decomposition methods. Only if no unconstrained abstract task is left, it branches over all (unconstrained) primitive tasks.

Consider the case of totally ordered HTN planning. Here, there is a single unconstrained task in each search node, and the behavior of all three algorithms will be the same. In an

empirical evaluation, the performance of all algorithms should therefore be similar (though there might be differences due to effects of randomness). When comparing the three algorithms on partially ordered problems, Algorithm 2 should always perform better than Algorithm 1. When comparing Algorithm 2 and Algorithm 3, the systematicity comes with a potential drawback that might lead to worse performance in certain domains/problems. Our main argument to use a progression-based search instead of a plan space-based search was to have the current state at hand to calculate heuristics, but when we postpone progression as in Algorithm 3, the state update is also postponed. There might be planning problems with a certain structure where the advantage of having an up-to-date state that comes with Algorithm 2 compensates for the non-systematic search. In such cases, Algorithm 2 might perform better. Therefore only an empirical evaluation can show which algorithm should be used in practice.

In general, systematicity is especially interesting in HTN planning since testing if two search nodes are isomorphic is as hard as the graph isomorphism problem, while testing whether the ordering constraints of a node are a subset of the ordering constraints of a second node is NP-complete (Behnke et al., 2015). So it is preferable to avoid equivalent search nodes in the first place.

The SHOP2 algorithm (Nau et al., 2003, Fig. 5) differs from the algorithms given here[6]. The closest resemblance is given to Algorithm 1, because it branches over all tasks (i.e. primitive and abstract ones) when an action has been progressed, but when a method has been applied, it only branches over the subtasks of that specific method (this differs from Algorithm 1). As a result, the choice of an abstract task is a commitment to the generated solution, because the next action that is progressed must be a subtask of that specific task. The decomposition trees will therefore always differ (at least) in the position labels. Therefore, the SHOP2 algorithm fulfills our definition of systematicity except from the following special case: For models that contain methods with empty subtask networks, the selection of the abstract task is no commitment to the solution and the algorithm generates duplicate search nodes.

5. Guiding Search in HTN Planning with Classical Heuristics

In classical planning there are many domain-independent heuristics available that might be interesting for HTN planning, too. When adapting classical heuristics to the HTN setting, one has to handle two main problems:

- The hierarchy may have large impact on the set of valid solutions, i.e. heuristics must be informed on both – the hierarchy and the state.
- Naturally, classical planning heuristics depend on a description of a state-based goal to reach. In HTN planning, there is (usually) no such goal definition.

A common way to create planning heuristics is to relax the problem to a simpler one that can be solved efficiently. The goal distance for the original problem is then approximated based on the solution to the simpler problem. In (grounded) classical planning, the plan

6. This means that our system configurations given by Höller et al. (2018a) did not simulate the SHOP2 algorithm, but instead the canonical algorithm.

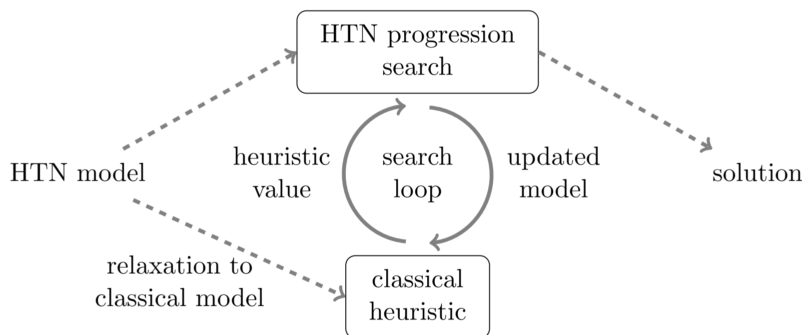

Figure 4: Schema of our overall approach to guide the HTN search with classical heuristics as given by Höller et al. (2019).

existence problem is PSPACE-complete (Bylander, 1994) and the commonly used heuristics are computed in polynomial time. There are several relaxations used in classical planning, but a direct adaptation to HTN planning is difficult. Maybe the best known relaxation is delete-relaxation, i.e. ignoring the delete effects of actions, but in HTN planning, the resulting problem is still NP-complete (Alford, Shivashankar, Kuter, & Nau, 2014). Since a direct adaptation of classical heuristics to HTN planning is difficult, we decided to have a two-stage process:

We introduce a transformation that relaxes the HTN planning problem to a classical planning problem. Since this classical model is used to calculate heuristics, we will call it the *heuristic model*. It is based on the actions and state description of the original HTN problem, but additionally incorporates reachability information from the hierarchy. This already relaxed problem is then passed on to an arbitrary classical heuristic to generate goal distance estimations. The resulting goal distance is then used in the HTN planning system. The schema of the overall approach is depicted in Figure 4.

We first introduce the transformation into the heuristic model, then we give its theoretic properties, and then we discuss how it is implemented efficiently.

5.1 Heuristic Model

We explain our transformation based on the running example given in Figure 5. It is a simple transport domain that divides the overall task of delivering a package into four steps:

1. Get a transporter to the package.
2. Pick-up the package.
3. Get the transporter to the goal destination.
4. Drop the package there.

There are three methods to decompose the get-to task. One results in a no-op action. It is e.g. helpful when there is more than one package at the same location because the

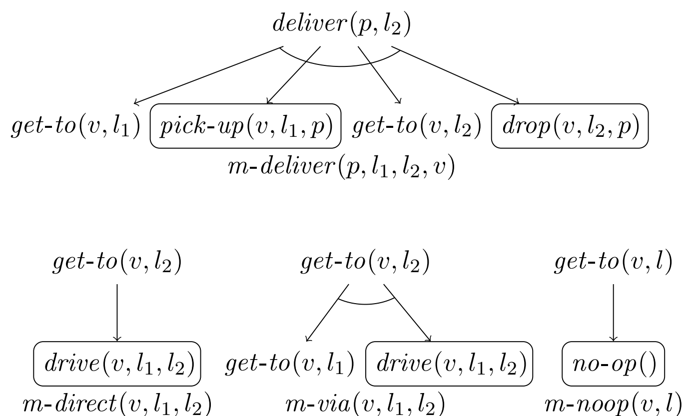

Figure 5: Example domain modeling a transport problem. Abstract tasks are non-boxed. Boxed tasks are primitive. Arrows indicate decomposition by methods. Names as well as parameters of each method are given below the definitions. The parameter p represents packages, v vehicles, and l, l_1, l_2 locations. The method *m-via*(v, l_1, l_2) decomposes e.g. the (abstract) task *get-to*(v, l_2) into the abstract task *get-to*(v, l_1) followed by the primitive task (i.e. action) *drive*(v, l_1, l_2).

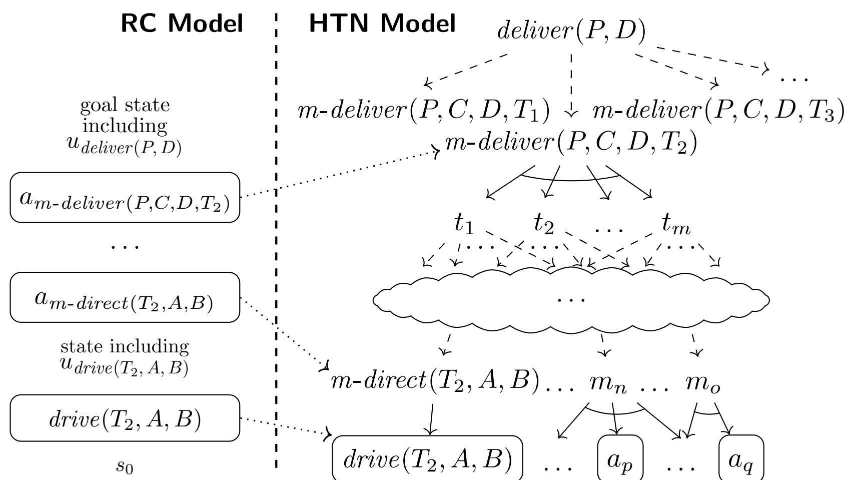

Figure 6: Schema of HTN decomposition as given by Höller et al. (2019).

transporter is already at the location, it needs to pick up the second package after picking up the first one. The second method results in a direct drive action. It can be used when there is a direct connection between the current position of the transporter and its destination. The last one is recursive and decomposes the overall task into a new get-to task and a drive action that is ordered afterwards. The objective of the overall task, i.e. that the package is at its destination, is enforced by the hierarchy, there is no state-based goal definition.

The right part of Figure 6 shows a schema of the HTN decomposition process: In the given example, it starts with a single deliver task at the top. There might be more than one possibility to decompose this task. In the figure, there is e.g. more than one transporter (T_1, T_2, ...) to deliver the package and the planner has to choose one (in the example, T_2 is chosen). The corresponding method has several subtasks that all need to be processed. If one of the subtasks is abstract, the planner needs, again, to choose a method. This is done until all tasks are primitive. Then, a sequence of the resulting actions needs to be found that is executable in the initial state.

The resulting tree can be regarded an AND/OR tree (Ghallab et al., 2004, Chapter 11), where the method nodes are AND nodes (all subtasks need to be done) and tasks are OR nodes (a single method needs to be applied). We encode the process of building that tree in a bottom-up manner into a classical planning problem. The new problem contains two kinds of actions, one represents the actions from the original HTN model (with slightly changed preconditions and effects) and one simulates method application.

We first add a new part to the state that stores which tasks are part of the tree. The actions from the HTN domain get a new effect that marks the action to be part of the tree. For each method, a new action is introduced. It is applicable when all its subtasks are already part of the tree and it adds the abstract task that it decomposes to the tree. The goal of the problem is to make the tasks contained in the current task network part of the tree.

Our approach builds the AND/OR tree in a bottom-up manner, we therefore denote the new state features with the letter u (*bottom-up reachability*), when the action $drive(T, A, B)$ is e.g. part of the tree, $u_{drive(T,A,B)}$ holds.

Consider the deliver method given at the top of Figure 5. In the new model, it results in an action $a_{m\text{-}deliver(?p,?l_1,?l_2,?v)}$ with four preconditions and a single add effect:

$$
\begin{aligned}
prec(a_{m\text{-}deliver(?p,?l_1,?l_2,?v)}) &= \{u_{get\text{-}to(?v,?l_1)}, u_{pick\text{-}up(?v,?l_1,?p)}, u_{get\text{-}to(?v,?l_2)}, u_{drop(?v,?l_2,?p)}\} \\
add(a_{m\text{-}deliver(?p,?l_1,?l_2,?v)}) &= \{u_{deliver(?p,?l_2)}\} \\
del(a_{m\text{-}deliver(?p,?l_1,?l_2,?v)}) &= \emptyset
\end{aligned}
$$

The transformation of all methods and all actions is given in Figure 7 and Figure 8, respectively. In the latter, the bold preconditions and effects are those from the original domain, the non-bold have been added during the transformation. Please ignore the newly added preconditions in Figure 8 for a moment, they result from an optimization that is introduced later on.

When a classical planning system is applied to the heuristic model, the actions in the solution belong to the choice points in the AND/OR tree. This is depicted at the left part of Figure 6. Starting in the initial state of the HTN planning problem (at the left bottom), the classical plan marks certain tasks to be part of the tree, until the *deliver* task at the top is reached.

Be aware that the classical planner would have to find an applicable sequence of all the original actions from the HTN domain that are marked as part of the tree. This makes our heuristics informed about the state transition in the original problem. It needs to compose (apply the decomposition methods in a reverse manner) the tree to reach the top, so our

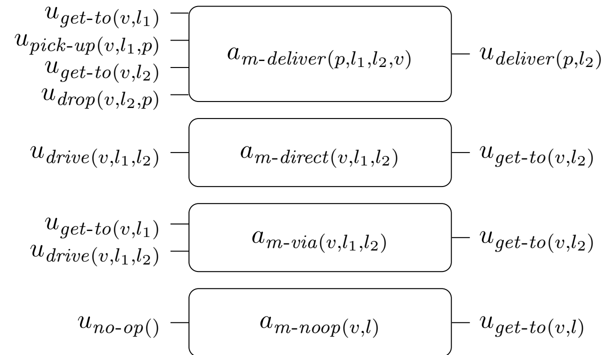

Figure 7: Actions derived from the methods given in Figure 5. Each action has as many preconditions as the respective method has subtasks that enforce the respective tasks to be in the tree. The application of the actions causes the task that the method decomposes to be part of the tree.

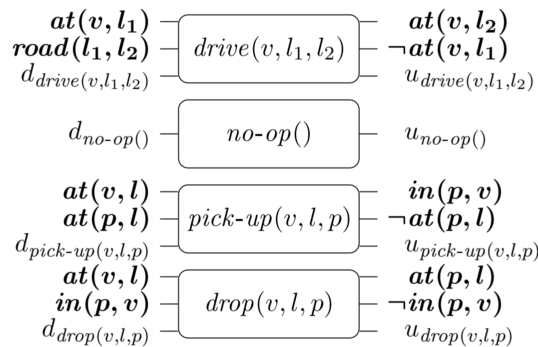

Figure 8: Basic actions from the domain. The preconditions and effects that are depicted bold are those from the original HTN planning problem. The non-bold preconditions and effects are introduced by the transformation.

heuristics are also informed about the hierarchy. However, the transformation makes three relaxations:

1. Tasks may be inserted apart from the decomposition hierarchy – even those that are not reachable anymore from the tasks in the current task network. This can be seen as a kind of HTN planning with task insertion (Geier & Bercher, 2011; Alford et al., 2015a). It may have two reasons: First, it can be done to fulfill preconditions of tasks that have to be inserted due to the hierarchy. Second, when having an additional state-based goal, it might be done to fulfill this state-based goal.

2. Ordering constraints introduced by the used decomposition methods are entirely ignored.

3. Since there is a single state feature for each task, every task needs to be processed only once, regardless of how often it is introduced by the hierarchy. This can be seen as a kind of HTN planning with task sharing (Alford et al., 2016).

In our actual system, we do not apply a classical *planner* to the heuristic model, but a classical *heuristic*. Since actions in the classical model represent nodes in the tree, the heuristic estimates the number of nodes in the tree. We will see in Section 5.2.2 that this number is exactly the goal distance in our search space.

As a next step, we want to reduce the set of tasks that the system is free to insert to those that are actually still reachable from the current task network. From a given task network tn, a task n is reachable via the hierarchy if and only if there is a task network $tn' = (T', \prec', \alpha')$ with $tn \rightarrow^* tn'$ and $\exists t \in T'$ with $\alpha'(t) = n$. Since tasks are inserted to fulfill preconditions or state-based goals, it is sufficient to determine this reachability information for *primitive* tasks (and not for abstract tasks). To make the information available for any applied heuristic, we decided to add it to the state of the transformed problem and to the preconditions of every action. We denote the state feature indicating that a task n is reachable as d_n (the task is *top-down reachable*). Figure 8 shows the added (non-bold) preconditions. The same optimization could also be made by changing the set of actions that is available for the heuristic function. By encoding it into the problem, we just have to adapt the initial state (that is updated anyway), but neither the rest of the model nor the heuristic function.

Now we have motivated all parts of our heuristic model and we give the formal definition. Since it mimes a relaxed *composition* of tasks, we denote it *Relaxed Composition Model* (RC).

Definition 11 (Relaxed Composition Model)**.** *Given an HTN planning problem $P = (L, C, A, M, s_0, tn_I, g, (prec, add, del))$ with $tn_I = (T_I, \prec_I, \alpha_I)$, we define our RC model as the following classical planning problem P' that we will denote* RC(P)*:*

$$
\begin{aligned}
P' &= \big(L', A', s'_0, g', (prec', add', del')\big) \\
L' &= L \cup L_d \cup L_u \\
&\quad L_u = \{u_n \mid n \in A \cup C\},\ \textit{with } L \cap L_u = \emptyset \\
&\quad L_d = \{d_n \mid n \in A\},\ \textit{with } (L \cup L_u) \cap L_d = \emptyset \\
A' &= A \cup A_M, A_M = \{a_m \mid m \in M\},\ \textit{with } A \cap A_M = \emptyset
\end{aligned}
$$

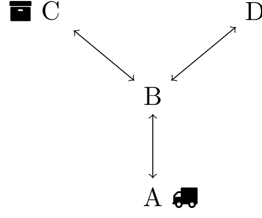

Figure 9: Schema of our example problem.

$$s_0' = s_0 \cup \{d_n \mid \exists tn' : tn_I \rightarrow^* tn' = (T', \prec', \alpha'), t \in T', \alpha'(t) = n\}$$
$$g' = g \cup \{u_n \mid t \in T_I, \alpha_I(t) = n\}$$

The actions $a \in A'$ of the new problem have the following preconditions and effects:

$$prec'(a) = \begin{cases} prec(a) \cup \{d_a\}, & \textit{iff } a \in A \\ \{u_n \mid t \in T, \alpha(t) = n\}, & \textit{iff } a \in A_M, a = a_m \textit{ with } m = (c, (T, \prec, \alpha)) \end{cases}$$

$$add'(a) = \begin{cases} add(a) \cup \{u_a\}, & \textit{iff } a \in A \\ \{u_c\}, & \textit{iff } a \in A_M \textit{ with } a = a_m \textit{ and } m = (c, tn) \end{cases}$$

$$del'(a) = \begin{cases} del(a), & \textit{iff } a \in A \\ \emptyset, & \textit{else} \end{cases}$$

In principle, we can use the resulting heuristic model with any classical heuristic to create new HTN heuristics.

Definition 12 (RC-based HTN heuristic (RC^h))**.** *Let P be an HTN planning problem and h a classical heuristic. We define an RC-based HTN heuristic RC^h as $h(\text{RC}(P))$.*

Before we come to theoretical properties of the heuristic model, we want to give a full example that shows how the solution to it could look like. We do this based on a simple transport problem including the methods given in Figure 5 and the actions given in Figure 8. A schema of the initial state of the problem is given in Figure 9. The package P is initially located at position C and a transporter T at position A. P has to be delivered at position D. This is guaranteed by the hierarchy, the state-based goal definition is empty. More formally, initial state, initial task network, and state-based goal are defined as follows:

$$\begin{aligned} s_0 = \{&at(T, A), at(P, C), \\ &road(A, B), road(B, A), road(B, C), road(B, D), road(C, B), road(D, B)\} \\ tn_I = (&\{t\}, \emptyset, \{t \mapsto deliver(P, D)\}) \\ g = \emptyset& \end{aligned}$$

The transformed problem is given by the actions shown in Figure 7 and 8. Initial state and goal definition of the transformed problem are defined as follows:

$$\begin{aligned} s_0 = \{&road(A, B), road(B, A), road(B, C), road(B, D), road(C, B), road(D, B), \\ &at(T, A), at(P, C), d_{drive(T,A,B)}, d_{drive(T,B,A)}, \ldots, d_{drive(T,D,B)}, d_{no\text{-}op()}, \\ &d_{pick\text{-}up(T,A,P)}, \ldots, d_{pick\text{-}up(T,D,P)}, d_{drop(T,A,P)}, \ldots, d_{drop(T,D,P)}\} \\ g = \{&u_{deliver(P,D)}\} \end{aligned}$$

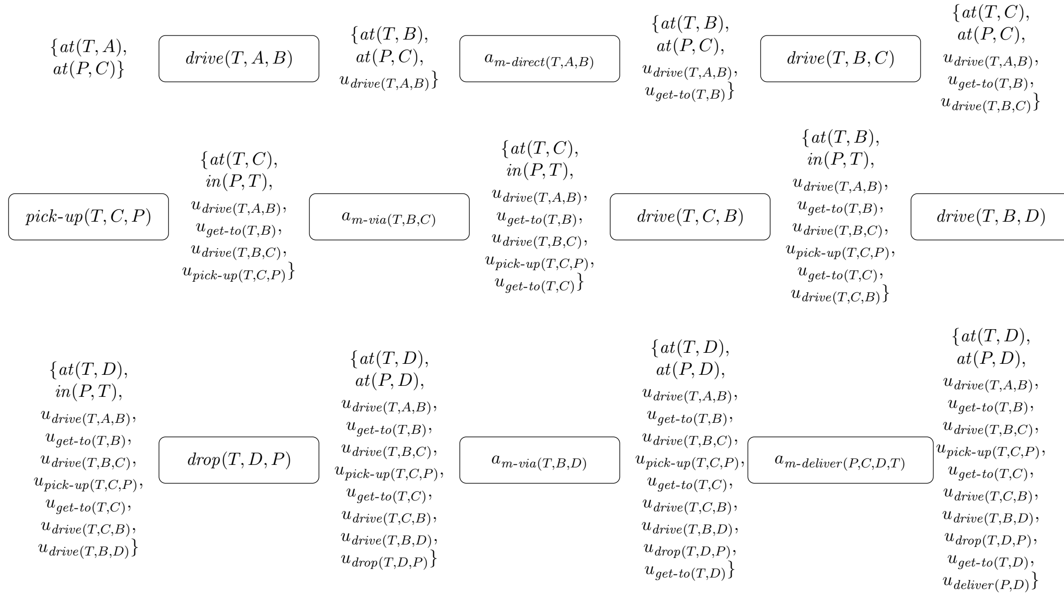

Figure 10: Solution to the transformed problem as it might be generated by a classical planning system and all intermediate states (top left to bottom right). Facts defining top-down reachability and the road network are omitted.

Figure 10 gives a solution to the transformed problem as it might be generated by a classical planning system.

5.2 Formal Properties

In this section we show that our heuristic model is linear in the size of the input HTN model and that it can be used to create safe, goal-aware, and admissible HTN heuristics.

5.2.1 Properties of the Heuristic Model

The transformation is based on the actions and state features of the original HTN planning problem. It adds one state feature for each abstract task and two state features for each action. One precondition and one effect is added to the actions of the original problem. There is a single new action per method that simulates its application. Such an action has as many preconditions as the method has subtasks and a single effect.

The number of state features is increased (compared to the size of the one of the input HTN planning problem) by a linear factor (two times the number of actions plus the number of abstract tasks), the sets of preconditions and effects of the original actions as well (two times the number of actions). The method encoding is as large as the set of methods in the input. Thus, for the entire heuristic model, the following theorem holds:

Theorem 4 (Size of the Heuristic Model). *The size of* $\text{RC}(P)$ *is linear in the size of the input problem* P*.*

The model can be created in polynomial time. For most parts this is straightforward, the only part where it is less obvious is the top-down reachability. We will explain how to calculate it in Section 5.3. Since it only calculates the hierarchical reachability ignoring state transition, it can be computed in lower polynomial time. Therefore the following theorem holds:

Theorem 5 (Computational Complexity of Creating the Heuristic Model)**.** *The heuristic model can be created in polynomial time.*

Further, we will see later on in Section 5.3 that the model does not need to be recreated in every search node, but that it can be updated.

This means that the calculation of the HTN heuristic will usually[7] be as hard as calculating the underlying classical heuristic. However, there is an issue that needs to be considered when selecting the classical heuristic. In classical planning, there is only one part of the model that changes from search node to search node: the current state. In our model, there are two parts that change: the current state and the goal definition. For some classical heuristics this is no problem, but others (like abstraction heuristics) use a computation-intensive preprocessing that depends on the goal definition and needs to be redone. This means that, though the combination with our encoding is possible, it might lead to poor results.

5.2.2 Properties of the Resulting HTN Heuristics

We have pointed out the relationship of progression-based search to forward search in a state space in classical planning. However, there is a difference that we want to emphasize: In state space search in classical planning, the estimation of the remaining costs to complete a plan corresponds to the distance in the state transition system to find a search node that is a goal node. This does, e.g., not hold in a plan space search in classical planning, because there are modifications that need to be done (inserting ordering constraints, causal links, etc.) that increase the distance in the search space, but are not reflected in the solution costs. When defining new heuristic functions, this leads to the question whether it should estimate the distance to a goal node or the costs of the resulting solution.

The first aim of our approach is to guide the search to find a solution. Therefore we do not want to estimate the costs of the actions in a solution, but the number of transitions between search nodes. In progression search, there are two kinds of modifications that cause such a transition between search nodes: progression of actions and decomposition of abstract tasks. Every abstract task on the way from the initial node in the search space causes exactly one transition; every action that is progressed also causes a single transition. Therefore the following lemma holds:

Lemma 1 (Goal Distance in Progression Search Space)**.** *The goal distance in HTN progression search equals the sum of decompositions and applied actions.*

For some HTN planning problem P, let h^*_m be the perfect goal distance estimation.

When a solution to an HTN planning problem is represented as decomposition tree, there is exactly one node for every abstract task that has been decomposed and exactly one node for every primitive task in the primitive plan. Therefore the following lemmas hold:

7. I.e. when the calculation of the classical heuristic takes at least polynomial time.

Lemma 2 (Number of Nodes in a DT)**.** *The number of nodes in a decomposition tree equals the sum of decompositions and applied actions to find the solution.*

Taking Lemma 1 and Lemma 2 together, we get:

Lemma 3 (The Number of Nodes in a DT Equals the Goal Distance)**.** *The number of nodes in a decomposition tree equals the goal distance in progression search.*

For some classical planning problem P_c, let $h^*(P_c)$ be the perfect goal distance estimation, i.e., the number of actions that need to be inserted to find a goal.

Theorem 6 ($\text{RC}^{h^*} \leq h^*_m(P)$)**.** *The perfect (classical) heuristic value computed on the RC heuristic model is smaller or equal to the actual goal distance in progression search.*

Proof. For this proof we treat the current search node as new planning problem. WLOG, we assume that there is at least one solution. Let $g^* = (T_g, E_g, \prec_g, \alpha_g, \beta_g)$ be a decomposition tree belonging to the solution with the minimal distance from the current search node. According to Lemma 3, $|T_g|$ is equal to the goal distance. We show that $h^*(\text{RC}(P)) \leq |T_g|$ by showing that there is always a solution to $\text{RC}(P)$ with exactly $|T_g|$ actions. We will denote the corresponding plan π^*.

Since g^* represents a solution, we know that all tasks are reachable via decomposition, i.e., the preconditions related to the top-down reachability are fulfilled. We know that there is a linearization of the primitive tasks in g^* that is applicable and fulfills the state-based goal. By construction, this sequence is also applicable in the transformation; the original preconditions and effects are unchanged, i.e., the application results in a goal state. Let π^* start with this sequence. Since g^* contains no loops, we can go up the tree and will find some node n such that all its subtasks are already contained in π^*. Let $m = \beta_g(n)$ be the method that has been applied to n. When all subtasks are executed, they will have fulfilled the preconditions belonging to a_m, the action representing m. We add this action to π^*. The action fulfills $\alpha_g(n)$. We repeat the procedure until all tasks are processed. The top-most tasks belonging to tn_I fulfill the newly introduced state-based goal of the transformation. □

From Theorem 6, the following corollaries follow directly:

Corollary 1 (RC Preserves Safety)**.** *For any safe classical heuristic h, i.e., if $(h(P_c) = \infty) \Rightarrow (h^*(P_c) = \infty)$, holds that $(h(\text{RC}(P)) = \infty) \Rightarrow (h^*_m(P) = \infty)$.*

Corollary 2 (RC Preserves Goal-Awareness)**.** *For any goal-aware classical heuristic h, i.e., if $h(P_c) = 0$ when the goal is fulfilled, holds that $h(\text{RC}(P)) = 0$ when the goal in P is fulfilled.*

The properties shown so far are interesting to control the search towards a goal node effectively. However, by setting the costs of all actions in A_M to 0, we can also create admissible heuristics to find optimal solutions. Let $h^*_{ac}(P)$ be the optimal HTN heuristic accumulating action costs.

Corollary 3 (RC Preserves Admissibility)**.** *For any admissible classical heuristic h it holds that $h(\text{RC}(P)) \leq h^*_{ac}(P)$.*

For a more detailed description on how to make our system find cost-optimal solutions as well as benchmark results, we refer to Behnke et al. (2019b, Section 4).

5.3 Efficient Implementation of RC Heuristics

Most parts of the heuristic model can be precomputed. The only parts that need to be updated are the initial state and the goal description. The initial state includes information on the state in the HTN search and top-down reachability information. The goal state represents the tasks in the current task network.

The reachability analysis is fully based on the hierarchy and does not analyze the interplay with state transition. It checks whether a certain task is reachable by decomposing the tasks in the current task network. This information can be precomputed in the following way: Given an HTN planning problem $P = (L, C, A, M, s_0, tn_I, g, \delta)$, it is determined based on a directed graph $G = (N, E)$ that includes as nodes the problem's task names: $N = C \cup A$. The set of edges is defined as $E = \{(c, c') \mid (c, (T, \prec, \alpha)) \in M \text{ with } \exists\, t \in T \text{ s.t. } \alpha(t) = c'\}$, i.e. two nodes are connected when there is a method decomposing the first node into a network including the second node. We compute the strongly connected components (SCCs) of this graph and contract nodes in the same SCC. The reachability of all tasks in the same SCC is equal. Now we have a directed acyclic graph and can compute reachability in a single bottom-up pass of the graph. We define the set of reachable tasks of nodes representing a primitive task a as $\{a\}$. For every other node, we define it as the union of the reachability sets of the children.

Reachability is stored for each task before search and needs to be accumulated over the tasks in a task network during search. For a certain search node, the reachability information as well as the goal definition can be calculated incrementally based on its parent nodes.

5.4 Discussion

In Section 2 we have already given a short introduction to Alford et al.'s work (2009, 2016) that – like our approach – encodes an HTN planning problem as a classical planning problem. However, instead of using heuristic values calculated on the classical model in the HTN search, the actual search is also done in the classical planning system.

The first encoding (Alford et al., 2009) is restricted to totally ordered problems, but the more recent one (Alford et al., 2016) could also be used as heuristic model in our overall approach. It simulates an HTN progression search in the state of a classical planning problem. Therefore the task network of a search node is stored in the state of that problem. To be able to represent the undecidable HTN problem in a decidable formalism the encoding needs a bound – that is denoted the *progression bound* – as input that determines the maximum number of tasks that can be stored in a task network. That way, tasks and ordering relations of a task network can be stored in the finite state of a classical problem. Newly introduced actions simulate method application and change the part of the state that represents the current task network according to the rules of task decomposition. The actions from the original domain are only applicable when they are contained in the current task network and are unconstrained. When an action is applied, it is deleted from the task network and the position it was stored before is not occupied anymore. The goal in the classical planning problem is to find an empty task network, i.e., to make all positions free.

While our encoding relaxes the problem (increasing the set of solutions), the encoding of Alford et al. (2016) does not. The set of solutions to the classical problem contains exactly these solutions to the HTN problem that can be generated with progression search with

the given bound in task network size. When it is used in a system like ours to calculate heuristics and the bound is chosen too low, the resulting heuristic is not safe, i.e. it might come up with infinite heuristic values even when the problem is still solvable. When the bound is increased, the model gets too large to extract heuristic values efficiently, because the size of the model is bounded by b^{st}, where b is the progression bound and st is the number of subtasks of the largest method. Besides the size of the resulting problem, the combination with delete-relaxed heuristics causes problems. The bound determines the maximum number of tasks that can be stored in a task network. When the bound is n, there are n positions where a task can be stored. When using a delete-relaxed heuristic, positions are not becoming occupied (because the state variable representing that it is free is not deleted). Together with the fact that most positions are initially free, this would lead to rather uninformed HTN heuristics when combining this encoding with delete-relaxed classical heuristics.

Another approach that is similar from a more abstract point of view is the work by Bercher, Geier, and Biundo (2013) (though the actual encoding it uses is quite different from ours). It is concerned with solving classical planning problems using POCL planning. During search, the POCL search nodes are translated into classical planning problems to calculate heuristic values on that translation by using standard heuristics from classical planning. Since the setting is classical planning, they can do this without relaxation.

The FAPE system (Bit-Monnot et al., 2016) uses a reachability analysis to prune search nodes that is based on a transformation of the hierarchical model into a temporal model. The used formalism defines *abstract actions* that are associated with a certain task and include the definition of subtasks as well as ordering constraints between them (like methods in HTN planning). FAPE's abstract actions are applied to tasks in a current plan and replace them with their subtasks. For the reachability analysis, each abstract action is translated into a temporal (non-hierarchical) action with an at-start condition that the corresponding task is *required*, i.e. necessary due to some other abstract action or the goal tasks (that correspond to the initial task network in HTN planning) and effects causing that the task it is associated with has been started (at the beginning of the action) and ended (at its end). The *required* condition ensures that actions can only be executed if they are part of the decomposition hierarchy, i.e. it prevents task insertion. Additional timed conditions enforce that the subtasks are *started* and *ended* at the positions they are ordered while timed effects cause them to be *required*. With this encoding, FAPE relaxes the condition that a task may be decomposed only once, i.e. any occurring abstract task can be decomposed multiple times with different methods. As such it is e.g. possible to execute two tasks that are mutex to each other due to restrictions introduced via the hierarchy. Notably, the two decompositions of the same task can be used to make each other executable, which is ordinarily not possible, hence it is a relaxation. FAPE's encoding contains the domain's ordering constraints, but the *started* and *ended* conditions can be fulfilled with different decompositions. This allows to ignore the ordering for a decomposition used to satisfy the *ended* condition and to arrange the subtasks before preceding tasks as the ordering constraint restricts only the time at which *ended* is made true. The same holds for the *started* conditions.

Based on the resulting temporal model, a reachability analysis is done by first optimistically applying the actions, i.e. without knowing if at-end conditions are fulfilled at the time

they have to be; and then recursively removing those actions that do not become reachable until a fix point is reached. The reachability analysis is performed in each search node. Based on its results, search nodes are pruned and the set of modifications is restricted.

The TDG-based work on heuristics by Bercher et al. (2017) is the most related approach to the one given here. Heuristic calculation starts with the primitive tasks and assigns each of them its own costs as heuristic value. It then proceeds with methods where the heuristic value of the subtasks has already been assigned. The heuristic value of methods is defined as the sum of the heuristic values of their subtasks. The heuristic value of abstract tasks is set to the minimum of the one of the methods that can be used to decompose it. When there are cycles, this procedure has to be done until it has converged (though this can be done efficiently, as described in the paper).

When we switch from a bottom-up to a top-down view, the calculation of the heuristic for a certain task can be seen as a tree with *min nodes* and *sum nodes*. To improve the heuristic, a reachability analysis (based on planning graph techniques) has been added, such that only primitive tasks are included that are still reachable and abstract tasks and methods of the HTN are deleted when they necessarily include unreachable actions. The resulting heuristic value is the sum of the effort[8] of tasks that are enforced by the hierarchy to be in the solution where all preconditions can be fulfilled using all primitive tasks still reachable via the hierarchy (similar to our top-down reachability). However, these additional actions that might be necessary to make other actions executable have no (direct) impact on the heuristic value.

Our heuristic model makes the classical heuristic build a similar tree (more precisely, it needs to estimate the size of the tree) implicitly by applying the actions that simulate method application. But by using an encoding in a classical model, we can combine it with arbitrary classical heuristics. A second advantage – that is maybe even more important – is that the actions added to ensure the applicability of other actions are incorporated in our heuristic value. Another interesting difference concerns task sharing: in our heuristic, every task name needs to be processed only once. The TDG-based heuristics can deal with multiple instances of the same task and increase the overall heuristic value according to the number of instances included in the solution. Therefore it can – in principle – even come up with heuristic values that are exponential in the size of the input model (which we can not when using common and unchanged classical heuristics).

6. Evaluation

Our search is based on a fully grounded model. Like the implementation used in previous work (Höller et al., 2018a), we realized it based on the preprocessing, i.e. grounding and reachability analysis, of the PANDA planning system (see Behnke et al., 2020). However, we reimplemented the search engine for this article that is now based on C++ instead of Java[9].

8. This might e.g. be action costs to plan cost-sensitive or the number of modifications to estimate the goal distance in the search space.
9. We call our new system $\text{PANDA}_{\text{pro}}$ (indicating the progression search). The source code of the PANDA planners and all domains and problems are available online at www.uni-ulm.de/in/ki/panda.

Since we are interested in a satisficing planner, we make an early goal test (as discussed on page 843), i.e. placed it before inserting a search node into the fringe. We included the h^{add}, the h^{FF}, and the $h^{LM\text{-}Cut}$ as classical heuristics (see Bonet and Geffner (2001), Hoffmann and Nebel (2001), and Helmert and Domshlak (2009), respectively).

6.1 Experiment Setup

All experiments ran on Xeon E5-2660 v3 CPUs with a base frequency of 2.60 GHz, a memory limit of 4 GB and time limit of 10 minutes.

6.1.1 Planning Systems

We included the following HTN planners into our evaluation. In the following, we will use the abbreviations G, A*, and WA* for Greedy Best-First search, A* search and Weighted A* search (with a weight of 2), respectively.

- We included 27 heuristic search configurations of our system: {Alg. 1, Alg. 2, Alg. 3} × {RC^{add}, RC^{FF}, $\text{RC}^{\text{LM-Cut}}$} × {G, A*, WA*}. We further included DFS with the three algorithms, and a special filter "heuristic" that will be explained below.
- TDG_{M}-R and TDG_{C}-R – The plan space-based PANDA system with its most recent heuristics (Bercher et al., 2017) and G, A*, WA* search. We have included the variants of the heuristic recomputing the TDG. Since our implementation uses the preprocessing of PANDA, the two systems are started with the same grounding of the model as search input.
- 2ADL – The approach introduced by Alford et al. (2016). It translates the HTN planning problem into a series of classical problems until a solution has been found. The given time is the accumulated time over the runs. We tested the ADL translation with the best-performing planning system of the original paper (Jasper) and with the following other planning systems from the agile and satisficing tracks of the International Planning Competition 2018:
  - 2ADL Jasper – combination with the Jasper planning system (Xie, Müller, & Holte, 2014) as used in the original paper.
  - 2ADL FDSS – combination with the Fast Downward Stone Soup planning system (Seipp & Röger, 2018).
  - 2ADL SaarPlan – combination with the SaarPlan planning system (Fickert, Gnad, Speicher, & Hoffmann, 2018).
  - 2ADL LAPKT – combination with the LAPKT-BFWS-Preference planning system (Francès, Geffner, Lipovetzky, & Ramírez, 2018).
- JShop2 – The JSHOP2 HTN planning system. The system is the only one in this evaluation that plans in a lifted manner. So we tested it with the original lifted input models. However, we found that starting it with the grounding generated by our grounder leads to a better performance, so we included these results in the evaluation instead. The better performance is not surprising. The domains are intended to

	#instances	Weighted A* Search									A* Search									Greedy Best-First Search								
		Alg. 3			Alg. 2			Alg. 1			Alg. 3			Alg. 2			Alg. 1			Alg. 3			Alg. 2			Alg. 1		
		$RC^{LM\text{-}Cut}$	RC^{add}	RC^{FF}	$RC^{LM\text{-}Cut}$	RC^{add}	RC^{FF}	$RC^{LM\text{-}Cut}$	RC^{add}	RC^{FF}	$RC^{LM\text{-}Cut}$	RC^{add}	RC^{FF}	$RC^{LM\text{-}Cut}$	RC^{add}	RC^{FF}	$RC^{LM\text{-}Cut}$	RC^{add}	RC^{FF}	$RC^{LM\text{-}Cut}$	RC^{add}	RC^{FF}	$RC^{LM\text{-}Cut}$	RC^{add}	RC^{FF}	$RC^{LM\text{-}Cut}$	RC^{add}	RC^{FF}
UM-Translog	22	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**
Satellite	25	23	24	**25**	24	23	24	21	21	24	23	24	24	21	24	24	16	22	21	23	24	24	23	24	23	23	21	22
Woodworking	11	**10**	**10**	**10**	**10**	9	**10**	8	9	9	8	**10**	**10**	8	9	9	6	7	8	9	9	9	8	9	**10**	8	**10**	**10**
SmartPhone	7	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	4	**5**	**5**	**5**	**5**	**5**	**5**	**5**	4	**5**	**5**	**5**	**5**	**5**	**5**	**5**
PCP	17	13	**14**	13	9	12	11	5	11	6	13	13	13	5	11	5	3	9	3	2	2	2	2	2	2	2	2	2
entertainment	12	11	11	11	11	11	11	11	11	11	8	11	8	8	11	11	8	11	11	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**
rover	20	4	**7**	5	4	4	4	2	4	3	0	4	2	0	4	0	0	4	0	4	4	4	4	5	5	5	4	4
transport	30	**13**	4	11	11	5	10	3	8	**13**	3	7	1	3	8	3	1	4	1	1	1	2	2	2	2	1	2	1
total	144	101	97	**102**	96	91	97	77	91	93	82	96	84	72	94	79	61	84	71	78	78	80	78	81	81	78	78	78

Figure 11: Coverage of our new system in different configurations.

be solved with domain-independent planning systems and are not optimized towards the planner. The grounding process includes a reachability analysis that combines hierarchical and state-based techniques that simplifies the search.

6.1.2 Domains

We included the following domains into our evaluation:

- The UM-Translog, Satellite, and Woodworking domains are HTN versions of the corresponding domains known from classical planning. They are further described by Bercher et al. (2014).
- Smartphone – A domain describing the task of operating a smartphone. A description is also given by Bercher et al. (2014).
- Rover – A version of the SHOP2 domain that makes no use of the SHOP2-specific features but only the standard HTN features.
- Transport – An HTN version of the transport domain. In this version of the model, the hierarchy restricts the set of solutions to improve the solutions (it is, e.g., impossible to pick-up or drop a single package more than once), but the main physics are fully given by the actions.
- Entertainment – It describes the problem of assembling a home entertainment system (see Bercher, Biundo, Geier, Hoernle, Nothdurft, Richter, and Schattenberg (2014) for details on the real-world problem). This is a hierarchical model of the domain where signal flow between devices, e.g. a DVD player and a TV, is not represented in state, but via the hierarchy.
- PCP – A domain that models Post's Correspondence Problem. Since this is an undecidable problem, it can (in general) not be represented in classical planning, but in HTN planning. However, we know for all instances that there is a solution.

6.2 Results

First we want to compare the three algorithms. Especially the comparison between Algorithm 2 and 3 is interesting. As discussed in Section 4.3, the systematicity of the search

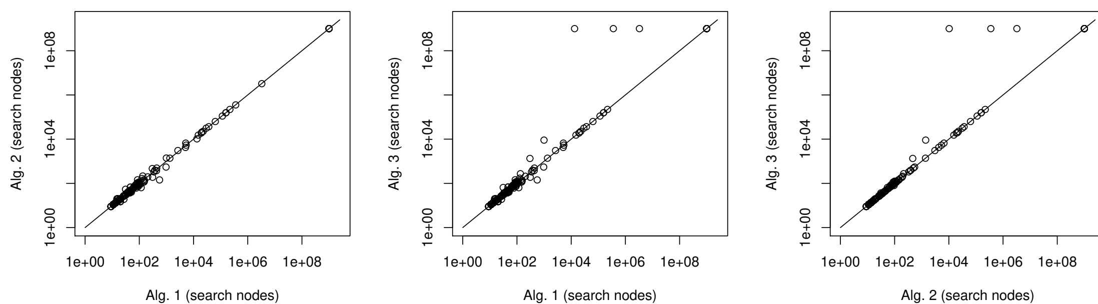

Figure 12: Totally ordered instances. Each point stands for the search nodes needed by two of the three algorithms on a single combination of problem instance, search strategy, and heuristic. The left plot compares Algorithm 1 and 2, the plot in the middle Algorithm 1 and 3, and the right plot Algorithm 2 and 3.

comes with the drawback of postponing the state update, which might result in less informed heuristic values. Figure 11 shows the coverage of the heuristic search configurations of our system. When we compare the three algorithms, we can see that the results are similar for all algorithms in Greedy Best-First search. Here it seems that the systematicity has no effect on the performance. In Weighted A* search, the search strategy with the highest coverage, it can be seen that all configurations benefit from the improved algorithms. The greatest effect can be seen in the $\text{RC}^{\text{LM-Cut}}$ configuration, this might be due to the high computational effort to calculate the heuristic. The RC^{add} configuration shows the smallest effect. An even greater relative improvement between the algorithms can be seen in A* search and especially in the $\text{RC}^{\text{LM-Cut}}$ and RC^{FF} configurations.

When we compare the overall performance of all configurations, it seems that the search contains too many dead ends that are not detected by our system to use a Greedy Best-First search. All algorithms and heuristics benefit from including the costs of already applied modifications into the overall evaluation of a search node. For RC^{FF} and $\text{RC}^{\text{LM-Cut}}$, a higher influence of the heuristic value as given by the Weighted A* search (compared to A*) has a positive effect on the performance.

So far we have only seen the coverage of the three algorithms. Next we want to compare the number of search nodes necessary to find a solution. To see whether the postponed state update has a negative effect, we do this also by using the heuristic search algorithms, i.e. G, A*, and WA* search. As discussed in Section 4.3, on instances that are totally ordered, the performance of all algorithms should be the same except for random effects. Figure 12 shows the performance on these instances. Each point represents the search nodes needed by two of the three algorithms on a single combination of problem instance, search strategy (G, A*, and WA* search), and heuristic (RC^{add}, RC^{FF}, and $\text{RC}^{\text{LM-Cut}}$). In the left plot, the x-axis shows the number of search nodes of Algorithm 1 and the y-axis the number of search nodes of Algorithm 2. Points on the diagonal result from instances where the compared algorithms need a similar number of nodes. The plot in the middle compares Algorithm 1 and 3, and the right plot Algorithm 2 and 3. The instances at the top margin indicate instances where one of the algorithms did not find a solution while the other one has. It can

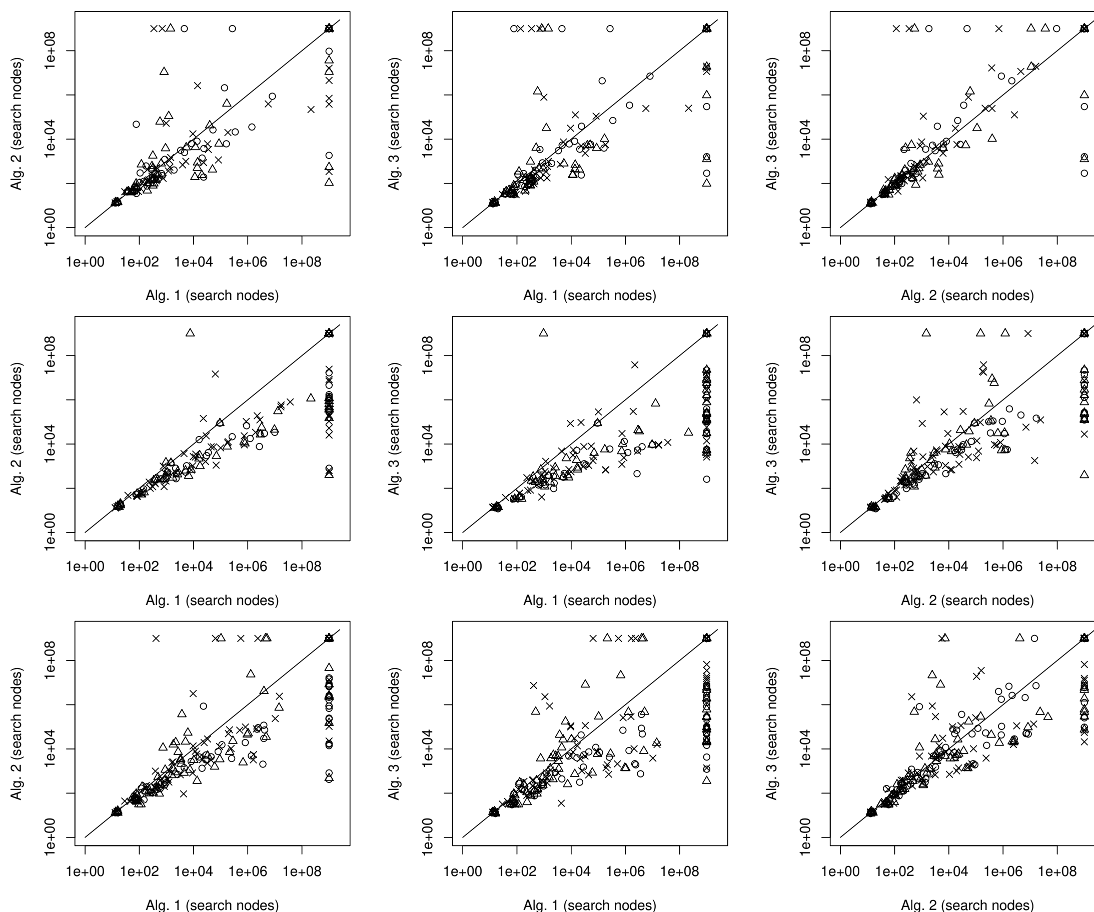

Figure 13: Search nodes needed to find a solution in partially ordered problem instances. Each row contains a comparison between Algorithm 1 and 2 (left), Algorithm 1 and 3 (middle) and Algorithm 2 and 3 (right). The rows show the different search strategies: from top row to bottom row, these are G, A*, and WA* search. The different heuristics are indicated by the different symbols. The p-values of the Wilcoxon signed rank test are (from top to bottom and from left to right): 1.4×10^{-4}, 5.0×10^{-7}, 0.033, 1.7×10^{-15}, 1.3×10^{-14}, 7.0×10^{-6}, 2.7×10^{-11}, 1.4×10^{-7}, 0.423.

be seen that the performance on totally ordered problem instances is similar for all three algorithms.

Next we want to have a look at partially ordered planning instances. These are given in Figure 13. Each row contains a comparison between Algorithm 1 and 2 (left), Algorithm 1 and 3 (middle) and Algorithm 2 and 3 (right). The rows contain the different search strategies: the top-most row shows the results for Greedy Best-First search, the row in the middle those for A* search, and the row at the bottom the results for Weighted A* search. The different heuristics are indicated by the different symbols: cycles stand for $\text{RC}^{\text{LM-Cut}}$, triangles for RC^{FF}, and crosses for RC^{add}.

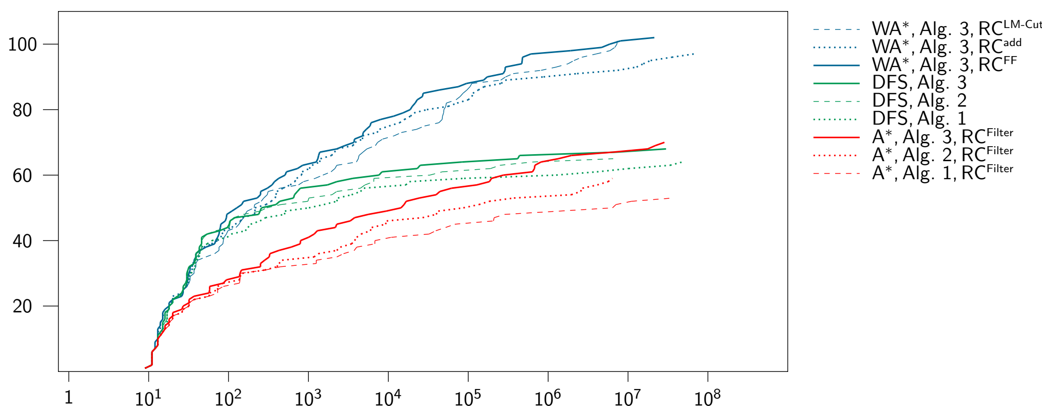

Figure 14: Accumulated number of solved planning problems (on the y-axis) for a given number of generated search nodes (on the x-axis). Please be aware of the log-scale of the x-axis.

Points below the diagonal indicate that a change of the algorithm has a positive effect on the performance, points above indicate a negative effect. For Greedy Best-First search the difference between the original algorithm and the two others is limited (as seen in the coverage table) and Algorithm 2 and 3 have a similar performance. In Weighted A* search (in the bottom row), it can be seen that both new algorithms have a positive effect on the performance, though the comparison of Algorithm 2 and 3 shows that there are several outliers (first of all from the RC^{FF} and RC^{add} heuristics), where the former algorithm needs less search nodes. But especially the instances at the right side of the diagram (where Algorithm 3 found a solution while Algorithm 2 did not) show the improvement. Like in coverage, the effect of the improvements can be seen best in A* search. Here, both improvements have a positive effect on the performance. However, it can be seen that especially the RC^{add} heuristic leads to several outliers.

We tested whether there is an statistical significant improvement between two algorithms using the Wilcoxon signed rank test on those instances that are solved by the respective algorithms (for the top-left diagram, this would e.g. be Algorithm 1 and 2, for the bottom-right diagram Algorithm 2 and 3), the p-values are given in the caption of the figure. Only the comparison of Greedy Best-First search using Algorithm 2 and 3 does *not* result in a significant improvement. However, this is the configuration where the heuristic has the highest impact.

Next we want to have a closer look at our heuristics. Figure 14 shows the accumulated number of solved planning problems (on the y-axis) for a given number of generated search nodes (on the x-axis). When comparing equal search strategies, a more informed heuristic will lead to a steep rising curve, a perfect result would be one leading from the left bottom to the left top corner of the diagram. Please be aware of the log-scale of the x-axis.

All heuristic search systems are close together, though the RC^{FF} shows the best performance in this evaluation. In DFS we can see the benefit of both algorithm changes. We tested whether this improvement is statistical significant using the Wilcoxon signed rank test on those instances that are solved by the respective algorithms. The p-value of the

		WA*			A*			G			DFS			WA*		A*		G						
		Alg. 3			Alg. 3			Alg. 3																
	#instances	$RC^{LM\text{-}Cut}$	RC^{add}	RC^{FF}	$RC^{LM\text{-}Cut}$	RC^{add}	RC^{FF}	$RC^{LM\text{-}Cut}$	RC^{add}	RC^{FF}	Alg. 3	Alg. 2	Alg. 1	TDG_m-R	TDG_c-R	TDG_m-R	TDG_c-R	TDG_m-R	TDG_c-R	2ADL Jasper	2ADL FDSS	2ADL SaarPlan	2ADL LAPKT	JSHOP2
UM-Translog	22	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	19	17	17	17	**22**
Satellite	25	23	24	**25**	23	24	24	23	24	24	24	23	22	24	21	21	24	**25**	24	23	19	14	12	22
Woodworking	11	**10**	**10**	**10**	8	**10**	**10**	9	9	9	8	6	7	8	**10**	8	8	8	9	5	5	5	5	8
SmartPhone	7	5	5	5	5	5	4	5	4	5	4	4	4	**6**	5	5	**6**	**6**	**6**	**6**	**6**	5	5	4
PCP	17	13	**14**	13	13	13	13	2	2	2	1	1	1	8	8	5	8	8	0	3	3	3	3	0
entertainment	12	11	11	11	8	11	8	**12**	**12**	**12**	6	6	6	9	11	7	9	9	9	5	5	5	4	5
rover	20	4	**7**	5	0	4	2	4	4	4	3	3	2	4	6	4	5	3	3	5	5	4	4	3
transport	30	13	4	11	3	7	1	1	1	2	0	0	0	1	1	2	1	1	1	**19**	17	13	13	0
total	144	101	97	**102**	82	96	84	78	78	80	68	65	64	82	84	74	83	82	74	85	77	66	63	64

Figure 15: Coverage of several search-based HTN planning systems (including some configurations of our system already given in Figure 11).

comparison of Algorithm 1 and 2 is 0.0044, of Algorithm 1 and 3 it is 0.0005, and that of Algorithm 2 and 3 is 0.0003. I.e. all improvements are significant[10].

An interesting question is whether it is actually possible to estimate the goal distance based on our heuristic model, or if the classical heuristics only do a reachability analysis and prune search nodes that can not lead to solutions anymore. Therefore we have included a fourth "heuristic" into our evaluation that checks whether the goal of the heuristic model is reachable in the planning graph and returns a heuristic value of 0 if it is, or ∞, if it is not (denoted RC^{filter}[11]). By comparing the configurations based on RC^{filter} to the other heuristics, it can be seen that our RC heuristics do not only prune the search space, but are able to guide the search based on the heuristic model. When we compare the performance of the three algorithms using the RC^{filter} heuristic we can, again, see the positive effect of the reduction of the search space. Since the search equals a BFS with dead end pruning, the effect is even greater than in DFS.

Figure 15 shows the coverage table including the results of several systems from related work as well as the three WA* configurations of our new algorithm (Algorithm 3). The configurations of our system have the highest coverage. The second best is reached by the approach of Alford et al. (2016), but the (plan space-based) PANDA system shows similar performance when using WA* search.

Figure 16 shows the number of planning problems solved within a given time. It can be seen that our overall system does not only reach the highest coverage, but does also need less time than the other systems. However, the preprocessing of the Jasper planner (that is based on the Fast Downward system) seems to be much faster than the one of the PANDA system. When we compare the figure with these given in previous work (Höller et al., 2018a, Figure 4), it can be seen that the preprocessing did take more time. This is caused by a reengineering of the parser to support more languages features. Another difference is that

10. Please be aware that the test included instances where *both* algorithms included in the particular test returned a solution, i.e., the set of pairs included in the test of Algorithm 1 and 3 is a different one than included in the test of Algorithm 2 and 3.

11. Please be aware that we want to increase the impact of the heuristic on the search and therefore show WA* configurations. For the RC^{filter} heuristic, however, this search strategy is equal to an A* search because the heuristic values are either 0 or ∞.

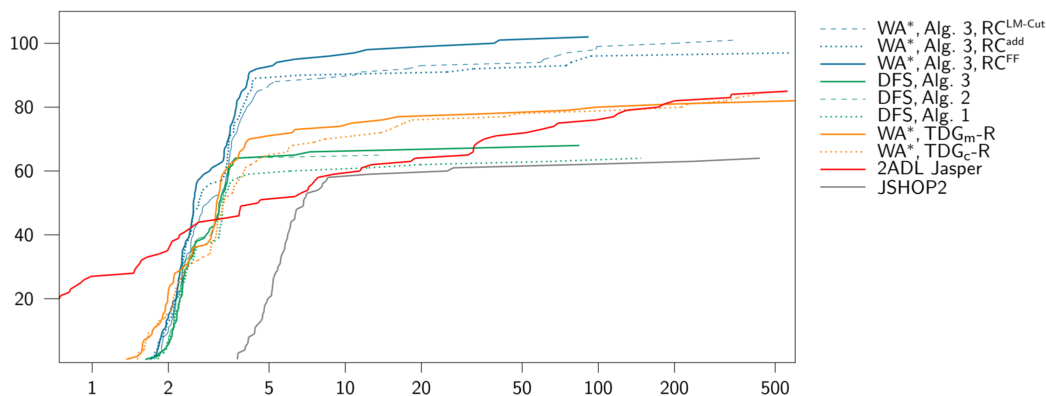

Figure 16: Accumulated number of solved planning problems (on the y-axis) for a time (on the x-axis, given in seconds). Please be aware of the log-scale of the x-axis.

the TDG-based heuristics solve the simplest problems more quickly than our system. This is caused by the transmission of the grounded model between the (Java-based) preprocessing and the (C++-based) reimplementation of our progression search engine.

When we sum up the results we see that both new progression algorithms improve the performance of the overall system – though there are configurations where all three algorithms perform equally well. Though the algorithm newly introduced in this article postpones the state update, it shows the best performance in the evaluation. The evaluation showed that the three classical heuristics are able to guide the search based on the heuristic model. The overall system outperforms the state of the art in search-based HTN planning.

7. Conclusion

In this article we propose progression-based heuristic search to solve HTN planning problems. We contribute two novel search algorithms and a family of heuristics.

We show that the canonical progression algorithm searches parts of the search space more than once. This is especially a problem in HTN planning, since checking whether a search node has been visited before is infeasible. We introduce two novel algorithms to avoid this, one that has been presented before in a conference paper, and one that is presented here for the first time. We show that both are sound and complete. We discuss systematicity in HTN planning, propose a definition, and show that our new algorithm introduced in this article is systematic according to this definition.

We further introduce a generic method to use arbitrary classical heuristics to guide the search. This is done by relaxing the HTN planning problem into a classical problem that is only used to calculate the heuristics. Heuristics calculated on that model are informed about hierarchical reachability and the state transition caused by actions. We show how the model can be updated efficiently during search and that it can be used to create safe, goal-aware, and admissible heuristic functions for HTN planning.

Our empirical evaluation shows that both algorithms have a positive effect on the performance of the overall system. It can be seen that classical heuristics calculated on the

heuristic model do not only prune dead ends from the search space, but that they provide goal distance estimations capable of guiding the search. The evaluation shows that our overall system outperforms the state of the art in search-based HTN planning.

Acknowledgements

The authors want to thank the anonymous reviewers of ICAPS 2018, IJCAI 2019, and JAIR for their valuable suggestions that helped to improve our work.

This work was partly funded by 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., Behnke, G., Höller, D., Bercher, P., Biundo, S., & Aha, D. W. (2016). Bound to plan: Exploiting classical heuristics via automatic translations of tail-recursive HTN problems. In *Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 20–28. AAAI Press.

Alford, R., Bercher, P., & Aha, D. (2015a). Tight bounds for HTN planning with task insertion. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1502–1508. AAAI Press.

Alford, R., Bercher, P., & Aha, D. W. (2015b). Tight bounds for HTN planning. In *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 7–15. AAAI Press.

Alford, R., Shivashankar, V., Roberts, M., Frank, J., & 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)*, pp. 3022–3029. AAAI Press.

Alford, R., Kuter, U., & Nau, D. S. (2009). Translating HTNs to PDDL: A small amount of domain knowledge can go a long way. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1629–1634. AAAI Press.

Alford, R., Shivashankar, V., Kuter, U., & Nau, D. S. (2012). HTN problem spaces: Structure, algorithms, termination. In *Proceedings of the 5th Annual Symposium on Combinatorial Search (SoCS)*, pp. 2–9. AAAI Press.

Alford, R., Shivashankar, V., Kuter, U., & Nau, D. S. (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)*, pp. 2–10. AAAI Press.

Bechon, P., Barbier, M., Infantes, G., Lesire, C., & Vidal, V. (2014). HiPOP: Hierarchical partial-order planning. In *Proceedings of the 7th European Starting AI Researcher Symposium (STAIRS)*, pp. 51–60. IOS Press.

Behnke, G., Höller, D., & 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)*, pp. 25–33. AAAI Press.

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

Behnke, G., Höller, D., & 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)*, pp. 7520–7529. AAAI Press.

Behnke, G., Höller, D., & 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)*, pp. 5500–5508. ijcai.org.

Behnke, G., Höller, D., Schmid, A., Bercher, P., & Biundo, S. (2020). On succinct groundings of HTN planning problems. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI)*. AAAI Press.

Bercher, P., Alford, R., & 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)*, pp. 6267–6275. ijcai.org.

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

Bercher, P., Biundo, S., Geier, T., Hoernle, T., Nothdurft, F., Richter, F., & Schattenberg, B. (2014). Plan, repair, execute, explain – How planning helps to assemble your home theater. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 386–394. AAAI Press.

Bercher, P., Geier, T., & Biundo, S. (2013). Using state-based planning heuristics for partial-order causal-link planning. In *Proceedings of the 36th Annual German Conference on AI (KI)*, pp. 1–12. Springer.

Bercher, P., Höller, D., Behnke, G., & 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)*, pp. 225–233. IOS Press.

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

Bit-Monnot, A., Smith, D. E., & Do, M. (2016). Delete-free reachability analysis for temporal and hierarchical planning. In *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI)*, pp. 1698–1699. IOS Press.

Blum, A. L., & Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, *90*, 281–300.

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

Bylander, T. (1994). The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, *94*(1-2), 165–204.

Dix, J., Kuter, U., & Nau, D. S. (2003). Planning in answer set programming using ordered task decomposition. In *Proceedings of the 26th Annual German Conference on AI (KI)*, pp. 490–504. Springer.

Dvořák, F., Barták, R., Bit-Monnot, A., Ingrand, F., & Ghallab, M. (2014a). Planning and acting with temporal and hierarchical decomposition models. In *Proceedings of the 26th International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 115–121. IEEE.

Dvořák, F., Bit-Monnot, A., Ingrand, F., & Ghallab, M. (2014b). A flexible ANML actor and planner in robotics. In *Proceedings of the 2nd ICAPS Workshop on Planning and Robotics (PlanRob)*, pp. 12–19.

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

Erol, K., Hendler, J., & Nau, D. S. (1994). Semantics for hierarchical task-network planning. Tech. rep. CS-TR-3239, UMIACS-TR-94-31, ISR-TR-95-9, Institute for Advanced Computer Studies, Institute for Systems Research, Computer Science Department, University of Maryland, College Park, MD 20742.

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

Fickert, M., Gnad, D., Speicher, P., & Hoffmann, J. (2018). SaarPlan: Combining Saarland's greatest planning techniques. In *Proceedings of the 9th International Planning Competition (IPC)*.

Francès, G., Geffner, H., Lipovetzky, N., & Ramírez, M. (2018). Best-first width search in the IPC 2018: Complete, simulated, and polynomial variants. In *Proceedings of the 9th International Planning Competition (IPC)*.

Fritz, C., Baier, J. A., & McIlraith, S. A. (2008). ConGolog, Sin Trans: Compiling ConGolog into basic action theories for planning and beyond. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pp. 600–610. AAAI.

Gabaldon, A. (2002). Programming hierarchical task networks in the situation calculus. In *Proceedings of the AIPS Workshop on On-line Planning and Scheduling*.

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

Gerevini, A., Kuter, U., Nau, D. S., Saetti, A., & Waisbrot, N. (2008). Combining domain-independent planning and HTN planning: The Duet planner. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI)*, pp. 573–577. IOS Press.

Ghallab, M., Nau, D. S., & Traverso, P. (2004). *Automated planning – theory and practice*. Elsevier.

Helmert, M., & Domshlak, C. (2009). Landmarks, critical paths and abstractions: What's the difference anyway?. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 162–169. AAAI Press.

Hoffmann, J., & Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, *14*, 253–302.

Höller, D., Behnke, G., Bercher, P., & Biundo, S. (2014). Language classification of hierarchical planning problems. In *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI)*, pp. 447–452. IOS Press.

Höller, D., Behnke, G., Bercher, P., & 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)*, pp. 158–165. AAAI Press.

Höller, D., Behnke, G., Bercher, P., & Biundo, S. (2018). Plan and goal recognition as HTN planning. In *Proceedings of the 30th International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 466–473. IEEE Computer Society.

Höller, D., Behnke, G., Bercher, P., Biundo, S., Fiorino, H., Pellier, D., & Alford, R. (2020). HDDL: An extension to PDDL for expressing hierarchical planning problems. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI)*. AAAI Press.

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

Höller, D., Bercher, P., Behnke, G., & Biundo, S. (2018b). HTN plan repair using unmodified planning systems. In *Proceedings of the 1st ICAPS Workshop on Hierarchical Planning (HPlan)*, pp. 26–30.

Höller, D., Bercher, P., Behnke, G., & Biundo, S. (2019). On guiding search in HTN planning with classical planning heuristics. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 6171–6175. ijcai.org.

Kambhampati, S., Knoblock, C. A., & Yang, Q. (1995). Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning. *Artificial Intelligence*, *76*(1-2), 167–238.

Kambhampati, S., Mali, A., & Srivastava, B. (1998). Hybrid planning for partially hierarchical domains. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI)*, pp. 882–888. AAAI Press.

Lotem, A., Nau, D. S., & Hendler, J. A. (1999). Using planning graphs for solving HTN planning problems. In *Proceedings of the 16th National Conference on Artificial Intelligence and 11th Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI)*, pp. 534–540. AAAI Press.

Mali, A. D., & Kambhampati, S. (1998). Encoding HTN planning in propositional logic. In *Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems (AIPS)*, pp. 190–198. AAAI.

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

Nau, D. S. (2007). Current trends in automated planning. *AI Magazine*, *28*(4), 43–58.

Nau, D. S., Au, T., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., & Yaman, F. (2003). SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, *20*, 379–404.

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

Schreiber, D., Pellier, D., Fiorino, H., & Balyo, T. (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)*, pp. 382–390. AAAI Press.

Seipp, J., & Röger, G. (2018). Fast downward stone soup 2018. In *Proceedings of the 9th International Planning Competition (IPC)*.

Shivashankar, V., Alford, R., & Aha, D. W. (2017). Incorporating domain-independent planning heuristics in hierarchical planning. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI)*, pp. 3658–3664. AAAI Press.

Shivashankar, V., Alford, R., Roberts, M., & Aha, D. W. (2016). Cost-optimal algorithms for planning with procedural control knowledge. In *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI)*, pp. 1702–1703. IOS Press.

Shivashankar, V., Alford, R., Kuter, U., & Nau, D. S. (2013). The GoDeL planning system: A more perfect union of domain-independent and hierarchical planning. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 2380–2386. AAAI Press.

Shivashankar, V., Kuter, U., Nau, D. S., & Alford, R. (2012). A hierarchical goal-based formalism and algorithm for single-agent planning. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 981–988. IFAAMAS.

Waisbrot, N., Kuter, U., & Könik, T. (2008). Combining heuristic search with hierarchical task-network planning: A preliminary report. In *Proceedings of the 21st International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, pp. 577–578. AAAI Press.

Weld, D. S. (1994). An introduction to least commitment planning. *AI Magazine*, *15*(4), 27–61.

Williamson, M., & Hanks, S. (1996). Flaw selection strategies for value-directed planning. In *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS)*, pp. 237–244. AAAI Press.

Xie, F., Müller, M., & Holte, R. (2014). Jasper: The art of exploration in greedy best first search. In *Proceedings of the 8th International Planning Competition*, pp. 39–42.