Towards Search Node-Specific Special-Case Heuristics for HTN Planning – An Empirical Analysis of Search Space Properties under Progression

Lijia Yuan, Pascal Bercher
School of Computing, The Australian National University, Canberra, Australia
{lijia.yuan, pascal.bercher}@anu.edu.au

Abstract

In hierarchical task network (HTN) planning, heuristic search is highly effective, but currently, there are only a few available heuristics and they are pre-selected for use. However, during progression-based search, many search nodes exhibit specific properties, e.g., they may become totally ordered or acyclic allowing for the application of specialized heuristics. For these search nodes, we conducted an experimental evaluation, employing reachability analysis, to examine the special cases encountered during search. Measuring how often various special cases (like acyclic problems) occur informs us of which heuristics developed for special cases – selected on a per-search node basis – are most promising.

Introduction

Hierarchical Task Network (HTN) planning (Erol, Hendler, and Nau 1996; Geier and Bercher 2011; Bercher, Alford, and Höller 2019) is a framework within *AI planning* where tasks are organized into hierarchies, consisting of primitive tasks that are directly executable and abstract tasks that require further decomposition. Solving HTN planning problems involves a range of different methods. Among the most successful ones is *progression-based search* (Höller et al. 2020), which operates in a forward manner adhering to specific orderings from left to right. By integrating heuristics, it refines the search trajectory, minimizing exhaustive exploration and effectively guiding the path to the goal.

Heuristic search has consistently demonstrated its effectiveness in HTN planning (Höller, Bercher, and Behnke 2020; Olz and Bercher 2023; Olz, Höller, and Bercher 2023). However, there are only a few heuristics available: There's the *TDG Heuristic* (Bercher et al. 2017), *Relax Composition Heuristic (RC Model)* (Höller et al. 2018, 2020), *ILP HTN Heuristic* (Höller et al. 2020), and a *Landmarks Heuristic* (Höller and Bercher 2021) – to the best of our knowledge, these are the only available ones up to now. All of them are designed for the general case without further restriction on the partially ordered (PO) tasks or how they interact via the task hierarchy. However, some search nodes show specific properties during the search. For instance, even if the initial problem is partially ordered, certain search nodes could become totally ordered (TO) during search or recursive parts might become non-recursive. Designing a heuristic for the general case is complex, so it might be easier to design a heuristic for one of the various special cases. As evidence for this, a recent pruning technique (discarding dead-ends and reducing the branching factor of search) (Olz and Bercher 2023) was developed for TO HTN problems, and yet has to be transferred to partial order HTN planning. We believe that developing pruning techniques or heuristics for special cases like this one (total order) or others, like acyclic problems / search nodes, thus shows great potential. However, choosing such a heuristic or technique would currently only be possible in advance, i.e., before search starts. We however hypothesized that special cases start arising during search, thus allowing to choose specialized heuristics during search thereby increasing their impact as they can be deployed even in problems that don't adhere to the respective special case in advance.

If more heuristics or techniques are tailored to specific special cases, analyzing each search node would enable the selection of a heuristic dedicated solely to that particular case and search node – and thus all search nodes below that one, since once a special case is established, it cannot be violated anymore. This could provide benefits, allowing us to solve planning problems previously impossible or, at the very least, expedite the process compared to before. Special case heuristics could also contribute to that: even if they are not "more informed", they might still be easier to compute.

To assess the potential of choosing specialized heuristics and/or techniques during search, we conducted an experimental evaluation checking how often the various known special cases occur. In this paper we only check TO problems and can hence not report how often TO search nodes occur while solving PO problems. We investigate a range of special cases and document their occurrence in percentage to all search nodes created under the respective search strategy (which was chosen based on the results of the IPC 2024). As a minor side contribution, we also propose how the set of reachable methods – which impact the accuracy of the respective current special case – can be computed in a tighter way, based on a *relaxed reachability analysis*. We report the number of special cases according to both investigations: a naive but quick one, the number and a more informed, but slower one. Finally, we draw a conclusion based on our findings – i.e., whether we believe that specialized heuristics might significantly impact search performance and if so, which special cases are the most promising ones.

HTN Planning Formalism

Our work builds upon the HTN planning formalization initially introduced by Geier and Bercher (2011) and further developed by Bercher, Alford, and Höller (2019), maintaining the core concepts established by Erol, Hendler, and Nau (1996). We would like to note in advance that whereas the formalization provided here is the general one (admitting *any* special case, including "none" by allowing partial order), the empirical study carried out will focus on totally ordered problems only.

A *task network* tn, represented as a tuple $(T, \prec, \alpha)$, consists of a finite set of *task id symbols* T, a strict partial order on T denoted by $\prec \subseteq T \times T$ (which is irreflexive, asymmetric, and transitive), and a mapping α that assigns each task id in T to either a *primitive task name* in N_p or *abstract task name* in N_a.

An *HTN domain* D is a tuple (F, N_p, N_a, δ, M), consisting of a finite set of *facts* F, a finite set of primitive task names N_p, a finite set of abstract task names N_a, a mapping $\delta : N_p \to 2^F \times 2^F \times 2^F$ that assigns each primitive task (also called an *action*) to its *preconditions*, *add effects*, and *delete effects*, and a finite set of *decomposition methods* M where each method $m \in M$ is a tuple (c, tn) pairing an abstract task c with a task network tn. An *HTN problem* is a tuple $\mathcal{P} = (D, s_I, \mathrm{tn_I}, g)$, comprising an HTN domain D, an *initial state* $s_I \subseteq 2^F$, an *initial task network* $\mathrm{tn_I}$, and a goal description $g \subseteq 2^F$.

A task network $\mathrm{tn_a} = (T_a, \prec_a, \alpha_a)$ will be decomposed by a decomposition method $m = (c, \mathrm{tn_m})$ into a new task network $\mathrm{tn_b} = (T_b, \prec_b, \alpha_b)$ if and only if there exists a task identifier $t \in T_a$ such that $\alpha_a(t) = c$ is replaced by subtasks in tn_m, and all ordering constraints from t will be inherited. It is written as $\mathrm{tn_a} \xrightarrow{t,m} \mathrm{tn_b}$. There exists a task network $\mathrm{tn}' = (T', \prec', \alpha')$ equivalent to $\mathrm{tn_m}$ such that $T' \cap T_a = \emptyset$. The only difference between tn' and $\mathrm{tn_m}$ are task identifiers to avoid repeating task identifiers. The application of m to $\mathrm{tn_a}$ results into the task network $\mathrm{tn_b}$ given as follows.

$$
\begin{aligned}
T_b &:= (T_a \setminus \{t\}) \cup T', \\
\prec_b &:= \prec_a \cup \prec' \cup \prec_x, \\
\alpha_b &:= \alpha_a|_{T_a \setminus \{t\}} \cup \alpha' \\
\prec_x &:= \{(t_a, t_b) \in T_a \times T' \mid (t_a, t) \in \prec_a\} \cup \\
&\quad\ \{(t_a, t_b) \in T' \times T_a \mid (t, t_b) \in \prec_a\}
\end{aligned}
$$

The notation $\mathrm{tn} \xrightarrow{*} \mathrm{tn}'$ indicates tn can be decomposed into tn' by using a sequence of methods.

A task network is *executable* if it has an *executable linearization* of its tasks, where a primitive task $p \in N_p$ linked to action a with $\delta(p) = (pre(a), add(a), del(a))$ is executable in the state s if and only if $pre(a) \subseteq s$, and its execution modifies s to the resulting state $(s \setminus del(a)) \cup add(a)$. An executable linearization for task network $\mathrm{tn} = (T, \prec, \alpha)$ is a sequence $(t_1, t_2, ..., t_n)$ where each $t_i \in T$ and $\alpha(t_i) \in N_p$ can be executed sequentially. A task network $\mathrm{tn_s} = (T_s, \prec_s, \alpha_s)$ is called a solution of an HTN problem $\mathcal{P} = (D, s_I, \mathrm{tn_I})$ if and only if $\mathrm{tn_I}$ is decomposed into $\mathrm{tn_s}$ through a series of decompositions, $\mathrm{tn_s}$ solely comprises primitive tasks ($\forall t \in T_s : \alpha(t) \in N_p$), and $\mathrm{tn_s}$ has an executable linearization. Solution task networks can only be obtained from the initial task network via decomposition without inserting any other tasks.

Known Special Cases

In this section, we provide the definitions for known problem classes, which are *primitive HTN problems*, *totally ordered problems*, *regular problems*, *acyclic problems* (Erol, Hendler, and Nau 1996) and *tail-recursive problems* (Alford et al. 2012; Alford, Bercher, and Aha 2015). We refine the *stratification* by Alford et al. (2012) and propose a more tight *HTN stratification* to assist in defining acyclic and tail-recursive problems. More specifically, our slightly changed formalization of stratifications can be regarded as another minor contribution of the paper, as it has some advantages over the existing one. While the existing one was not wrong, our "tighter version" allows us to differentiate more classes based on the stratification alone, without having to consult the underlying HTN problem.

HTN planning is undecidable (Erol, Hendler, and Nau 1996; Geier and Bercher 2011) but various special cases can make the plan existence problem easier. Erol, Hendler, and Nau (1996) provided tight complexity results (i.e., with matching upper and lower bounds) for primitive HTN problems and for regular problems, and provided upper bounds for totally ordered problems. Alford, Bercher, and Aha (2015) provided matching lower bounds for total-order problems, and tight bounds for tail-recursive problems.

Primitive HTN problems (when all tasks in the initial task network are primitive) are the base case, appearing at the end of the search if a solution exists. Deciding whether a primitive task network has an executable linearization is NP-complete (shown independently by Erol, Hendler, and Nau (1996) and Nebel and Bäckström (1994), later refined by Tan and Gruninger (2014)).

Definition 1 (Totally Ordered Problem)**.** *An HTN problem $\mathcal{P}$ is called* totally ordered *if the ordering of its initial task network $\mathrm{tn_I}$ is totally ordered and all decomposition methods are totally ordered, i.e., for each $m \in M$ with $m = (c, \mathrm{tn_m})$, $\mathrm{tn_m}$ is a totally ordered task network.*

Totally ordered HTN planning is in EXPTIME (Erol, Hendler, and Nau 1996) and EXPTIME-hard (Alford, Bercher, and Aha 2015) and hence EXPTIME-complete.

Definition 2 (Regular Problem)**.** *An HTN problem $\mathcal{P}$ is* regular *if its initial task network $\mathrm{tn_I}$ and $\mathrm{tn_m}$ in all its methods $(c, \mathrm{tn_m}) \in M$ are regular. A task network tn = $(T, \prec, \alpha)$ is regular if*

- *there is at most one task in T that is abstract and*
- *if $t \in T$ and $\alpha(t) \in N_a$, it is the last task in tn, i.e., for all $t' \in T$ with $t' \neq t$, we have $(t', t) \in \prec$.*

In a regular problem $\mathcal{P}$, given that the initial task network $\mathrm{tn_I}$ and task networks $\mathrm{tn_m}$ with all methods $(c, \mathrm{tn_m}) \in M$ are regular, every primitive task in a task network needs to be progressed first, then the abstract task will be decomposed. The abstract task will be the last task in each search node until the primitive task network is found. The largest size

of the search node during the progression search will hence be bounded by the method $(c, \mathrm{tn_m})$ with the largest task network $\mathrm{tn_m}$. These problems were shown to be PSPACE-complete (Erol, Hendler, and Nau 1996).

For the more complex problem restrictions like tail-recursive ones, we provide stratifications, as introduced by Alford et al. (2012) and also used for defining tail-recursiveness (Alford, Bercher, and Aha 2015).

Definition 3 (Stratification). *A set $R \subseteq N_a \times N_a$ is called a* stratification *if it is a total preorder (i.e., reflexive, transitive, and total). A* stratum *is an inclusion-maximal subset $S \subseteq N_a$ such that for all $x, y \in S$ both $(x, y) \in R$ and $(y, x) \in R$ hold.*

According to this definition, stratifications are a concept independent of the underlying HTN problem. In the original definition of tail-recursive problems (Alford, Bercher, and Aha 2015), a specific stratification has to exist for the respective problem to be called tail-recursive. However, although those definitions were sufficient to define tail-recursiveness adequately, there could still be *several* stratifications adhering to the required restrictions. Thus, even for tail-recursive problems, the set of possible stratifications defined over the respective compound tasks wasn't unique. For example, consider a problem an initial abstract task c_I decomposes into the total-order task network $c_1 \to c_2$ with compound tasks c_1 and c_2. As we will see later, tail-recursiveness will require that $(c_I, c_2) \in R$ and $(c_2, c_I) \notin R$, i.e., it requires c_2 to be on a strictly lower stratum than c_I, but it will *not* impose a restriction on where exactly c_1 sits with regard to c_I. That is, whereas at least $(c_1, c_I) \in R$ is demanded, the original definition of the interplay of stratification and tail-recursiveness would also allow $(c_1, c_I) \in R$ to hold, thus making stratifications not unique. In fact, totality requires that any two compound tasks are "artificially" put into some kind of relationship, even if none of the tasks can be decomposed into another. We propose a stricter definition that removes the requirement of totality and imposes an *exact* relationship between the decomposition hierarchy and the underlying stratification – thus making the stratification a formal one-to-one mapping of the underlying task hierarchy.

Another advantage (on top of having a *unique* stratification per problem, having a *clear, intuitive semantics* for stratifications, and simplified problem definitions for tail-recursive problems) is that in our proposed definition we can differentiate whether singleton stratums represent recursive tasks or not. The original definition required reflexivity, meaning that we have $(c, c) \in R$ for every compound task. This however means that it is impossible to identify, based on the stratification alone, whether the abstract task c can actually reach itself (and hence is recursive) or not, because reflexivity is demanded by definition rather than being a consequence of reachability. In our definition, reflexivity follows only if a task can reach itself.

Definition 4 (HTN Stratification). *Given an HTN domain $D = (F, N_p, N_a, \delta, M)$, a set $R_{HTN} \subseteq N_a \times N_a$ is called an* HTN stratification *of D if and only if it is transitive and it holds $(c', c) \in R_{HTN}$ if and only if c' is reachable from c via decomposition.*

For simplicity, we will use the terms "stratification" and "stratum" and hence skip the "HTN".

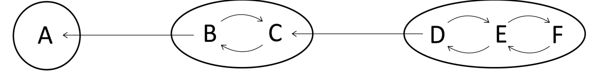


Figure 1: An example of stratification R_{HTN} that has a height of 3, featuring abstract tasks A, B, C, D, E, and F. Circles denote strata, $S_1 = \{A\}$, $S_2 = \{B, C\}$ and $S_3 = \{D, E, F\}$. Directed arrows between circles show the decomposition hierarchy, with arrows pointing from higher to lower levels (e.g., $(A, B) \in R_{HTN}$). Example taken from an ICAPS tutorial on HTN planning (Bercher and Höller 2018).

A directed graph can represent stratification diagrammatically (Figure 1). Task A cannot be decomposed into any other abstract task, and tasks B, C and tasks D, E, F can be decomposed into each other. Due to the requirement of strata being inclusion-maximal subsets of N_a, there are no other strata. As the number of strata in the stratification is 3, the height of the stratification is 3. We also can say that S_2 is a stratum lower than S_3 and S_1 is lower than both S_2 and S_3. Tasks are in different strata if they are in different decomposition hierarchy levels. For instance, task D has a stratum height of 3 since it is at the highest hierarchy level. Primitive tasks, unrelated to the hierarchy, are assigned a stratum height of 0.

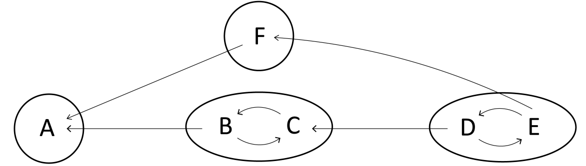


Figure 2: An example of a partially ordered stratification R_{HTN} with a height of 3, featuring abstract tasks A, B, C, D, E, and F. Directed arrows between tasks indicate decomposition methods. Circles denote strata: $S_1 = \{A\}$, $S_2 = \{B, C\}$, $S_3 = \{F\}$, and $S_4 = \{D, E\}$.

Tasks B, C in Figure 2 have stratum height 2 as they are on a strictly higher stratum as A, which is on the lowest level. B and C share a stratum since they can be turned into each other, but they don't share a stratum with A since they can't be turned into each other. Since tasks D, E can be turned into B, C but not vice versa, they are on a stratum with height 3. Now, F is on a higher stratum than A and thus has height 2 or 3. They are not in the same stratum as any of the other tasks, because they cant be turned into each other.

A stratum with a single abstract task c implies that no other task c' exists for which c can reach c' while c' can also reach c via decomposition. However, c may be decomposed into itself, indicating a self-loop if $(c, c) \in R_{HTN}$. Therefore, R_{HTN} effectively differentiates whether c is in a self-loop.

Definition 5 (Acyclic Problem). *An HTN problem $\mathcal{P}$ is acyclic if for its HTN stratification $R_{HTN} \in N_a \times N_a$ holds: if $(c', c) \in R_{HTN}$, then $(c, c') \notin R_{HTN}$.*

The stratification R_{HTN} for the acyclic problem is irreflexive – so demanding that R_{HTN} is irreflexive is an alternative definition. As the stratification is transitive, it becomes asymmetric when it is irreflexive. The search space will be finite during the progression search because the algorithm does not need to deal with recursion. The acyclicity will bring the computational complexity of an HTN problem down to NEXPTIME-complete (Alford, Bercher, and Aha 2015). If an HTN problem is acyclic and totally ordered, it will be PSPACE-complete (Alford, Bercher, and Aha 2015).

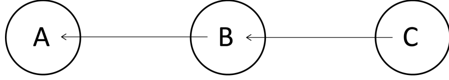
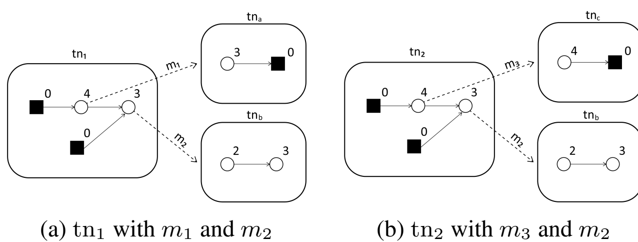


Figure 3: An example of stratification R_{HTN} of the acyclic HTN problem $\mathcal{P}$.

For example, there is an acyclic HTN problem $\mathcal{P}$ where all tasks in N_a are A, B and C, as shown in Figure 3. The stratification of $\mathcal{P}$ is $R_{HTN} = \{(A, B), (B, C), (A, C)\}$ and the strata are $\{A\}, \{B\}$ and $\{C\}$ as per Definition 4. All strata for the acyclic problem contain exactly one abstract task since all abstract tasks are in the different decomposition hierarchy levels.

For the definition of tail-recursiveness, we require the concept of last tasks or non-last tasks, respectively. A task is called *last task* in a task network if and only if all other tasks in that task network are ordered to occur before it. A task is called *non-last task* if and only if it is not a last task. Note that, with the exception of task networks of size 0 or 1, all task networks have non-last tasks (potentially all of them), but not every task network has a last task.

Definition 6 (Tail-Recursive Problem)**.** *An HTN problem $\mathcal{P}$ is* tail-recursive *if for its HTN stratification R_{HTN} and for all methods $(c, \mathrm{tn_m}) \in M$ it holds that for any non-last abstract task c_n in $\mathrm{tn_m}$ with $c_n \in N_a$, $(c, c_n) \notin R_{HTN}$.*



(a) tn_1 with m_1 and m_2 (b) tn_2 with m_3 and m_2

Figure 4: Comparison of tail-recursiveness (left) and non-tail-recursiveness (right). Filled squares denote primitive tasks; circles represent abstract tasks; numbers beside indicate the height of the stratum. Example borrowed from an ICAPS tutorial on HTN planning (Bercher and Höller 2018).

In Figure 4a, the abstract task c in tn_1, which is the one with stratum height 4, can be decomposed into a task network $\mathrm{tn_a}$, which contains tasks that are on a (strictly) lower stratum than c. Therefore, task network $\mathrm{tn_a}$ cannot possibly contradict tail-recursiveness. Also note that the position of c in tn_1 does not play any role, only the position of the tasks within $\mathrm{tn_a}$ are relevant.

The last task c_l in tn_1 can be decomposed into a task network that contains a task that has the same stratum height as c_l. However, that task in $\mathrm{tn_b}$ happens to be the last task, hence this is not a problem. All non-last tasks in $\mathrm{tn_b}$ are indeed on a (strictly) lower stratum than c_l, so this task network also does not contradict tail-recursiveness. Again, the position of c_l within tn_1 was irrelevant, only the position of tasks within $\mathrm{tn_b}$ matters.

In Figure 4b we see an example of a problem that is *not* tail-recursive. Whereas method m_2 is as before and hence doesn't cause problems, method $m_3 = (c, \mathrm{tn_c})$ does. Here, we can see that $\mathrm{tn_c}$ contains a task with stratification height 4, which is the same as the task c from which it got decomposed. Hence that task (in $\mathrm{tn_c}$) would be required to be its last task – but it is not. Again, the position of c within tn_2 was not relevant (it never is); only the positions of tasks within the decomposition method count – interestingly this implies that the form and structure of the initial task network are completely irrelevant. This is interesting because the proof by Erol, Hendler, and Nau (1996) for the undecidability of HTN planning required only two partially ordered compound tasks in the initial task network, whereas all methods were totally ordered. However, tail-recursive methods are enough to achieve decidability, whereas totally ordered methods are not. (This is not a contribution of this paper, but we find this an interesting observation to point out.)

The computational complexity of (ground) tail-recursive problems is EXPSPACE-complete (Alford, Bercher, and Aha 2015), and of tail-recursive and totally ordered problems is PSPACE-complete (Alford, Bercher, and Aha 2015).

Testing for these special cases can clearly be done in polynomial time (with respect to the size of a ground model) since stratifications are essentially the very same as Task Decomposition Graphs (see next section), and checking for the respective properties equals checking simple graph properties. Hence, we will not provide details on how these checks can be done.

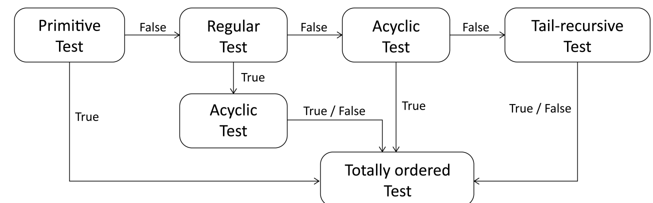


Figure 5: Control flow graph of tests ordering to minimize computational costs and addresses property dependencies efficiently. Arrows with labels to show the path based on test outcomes.

We do, however, mention that the order in which these tests are done can have a large impact on efficiency. Some problem class tests depend on others and are more expensive or time-consuming than others. Therefore, optimizing the sequence of these property tests can lead to greater effi-

ciency (or the other way around: some test orders might be redundant). As shown in Figure 5, if a task network is found to be primitive, it bypasses the need for regular, acyclic, and tail-recursive tests (as they all will be positive), proceeding directly to the totally ordered test. Otherwise, it conducts regular and acyclic tests. Skipping the tail-recursive test is feasible if either the acyclic or regular test is positive, as either acyclicity or regularization is a special case of tail-recursiveness. After these, the task network's total order is assessed, which is independently unaffected by the results of other tests.

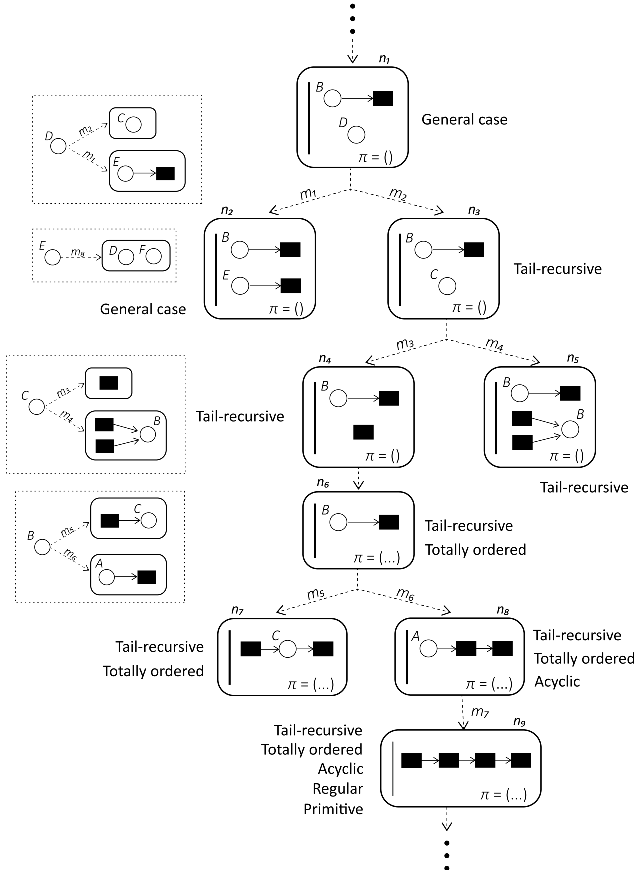


Figure 6: An example of special cases during the search: Filled squares represent primitive tasks, and unfilled circles represent compound tasks. π is the prefix of the generated plan. Vertical lines in search nodes from n_1 to n_9 illustrate the states during the search. We assume no primitive task has preconditions or effects, so the state remains unchanged. The properties of each task network are shown next to each node, with existing methods for tasks B, C, E, D. It has the same stratification as Figure 1.

In progression search, if a task network within a search node possesses a particular property, all task networks within its child nodes will inherit this property. As illustrated in Figure 6, once a property is present in a search node, it propagates to all its descendant nodes. To enhance efficiency, we check the parent node's property before conducting property tests in a search node, potentially reducing overall complexity by avoiding redundant checks.

Reachability Information from Task Decomposition Graphs

In a task network, its properties like total ordering, regularity, acyclicity, or tail-recursiveness depend on both the initial task network and all its reachable methods. The *task decomposition graph (TDG)* is the foundational data structure for hierarchical reachability analysis. It was first introduced by Elkawkagy et al. (2012) and refined by Bercher et al. (2017).

For simplicity, for an HTN problem, we introduce an artificial abstract task c_I that is not originally in the domain. c_I can be decomposed into the initial task $\mathrm{tn_I}$. Subsequently, we incorporate c_I into the domain, establishing $(c_I, \mathrm{tn_I}) \in M$.

Definition 7 (Task Decomposition Graph (TDG))**.** *For a given HTN problem $\mathcal{P}$, the* task decomposition graph (TDG) *is defined as a directed bipartite graph $G = \langle V_T, V_M, E_{T\to M}, E_{M\to T}\rangle$. This graph comprises a set of task vertices V_T, a set of method vertices V_M, edges from tasks to methods $E_{T\to M}$, and from methods to tasks $E_{M\to T}$, such that the following conditions are satisfied:*

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

The TDG can represent the hierarchical reachability of an HTN planning problem, i.e. tasks and methods that can be reached via decomposition. According to the simplistic definition provided, it is only based on the reachability from decomposition without considering the states. However, Elkawkagy, Schattenberg, and Biundo (2010) and Bercher et al. (2017) suggested that method nodes containing unreachable primitive tasks based on the state reachability analysis can be removed. (For a more formal definition, consult Def. 4 by Behnke et al. (2020), where this idea is incorporated for grounding lifted HTN planning problems.)

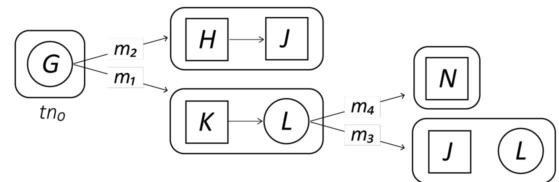


Figure 7: A fragment of a TDG (Bercher et al. 2017): task networks are denoted by surrounding boxes, abstract tasks by circles, and primitive tasks by square boxes.

Bercher et al. (2017) observed that the reachable methods per search node change, as illustrated in Fig. 7. Here, the reachable method set from the initial task network is still m_1 to m_4. However, after m_2 was applied, the reachable methods even became empty, since the new current task network is primitive. If alternatively m_1 would have been chosen, at least m_2 will not be reachable anymore, but potentially even one of m_3 and m_4 as it might be that those actions from m_2's task network were used to (inaccurately) determine the reachability of m_3 or m_4. Hence, recomputing the TDG after some decisions were made can lead to a reduced reachable method set (Bercher et al. 2017).

This is highly relevant for our endeavor of determining search space properties since these properties directly depend on the reachable methods. Going back to our example (Figure 7), if task K cannot be executed, method m_1, and consequently m_3 and m_4 become unreachable. This leaves only m_2 as reachable and thus the sub-TDG makes tn_0 primitive, acyclic, and totally ordered – properties that the initial TDG did not show.

Thus, when computing the properties of a search node – which naturally are based on the set of methods reachable from it – one has two possibilities for what to do:

- We use the initial TDG but restrict it to the methods reachable from the current search node. This process is relatively quick since it does not require TDG recomputation.
- We recompute the TDG from the current search node. This is more expensive but leads to potentially fewer reachable methods (since the initial TDG was computed based on all primitive tasks reachable in it, whereas the made decisions reduced this set and might hence further reduce the size of the new TDG (Bercher et al. 2017)).

In our empirical evaluation we do both. For the recomputation of the TDG, we use the Relaxed Composition Heuristic (Höller et al. 2018, 2020) as a basis. It transforms an HTN search node into a classical problem, which can represent a superset of all reachable decomposition methods (expressed as classical actions). We used this transformation as a basis, estimating all reachable methods by computing a fixed point in a relaxed planning graph (RPG) computed from the classical problem.

Evaluation

We now report on our findings.

Benchmarks.

We analyzed the entire total-order (TO) benchmark set from the IPC 2023[1]. We restrict to TO domains both due to space restrictions and also since the IPC showed that in most instances, the partial order can be compiled away in advance without making the respective problem unsolvable (Wu et al. 2022, 2023).

[1]https://ipc2023-htn.github.io/

Configuration.

Experiments were conducted in a virtualized environment. The underlying hardware was powered by an Intel(R) Core(TM) i9-8950HK CPU, clocking at 2.90GHz, which was allocated a single CPU core and provisioned with 8GB of RAM for the experiments, and 30 minutes limited for each instance (memory and time limit of IPC 2023).

We run the latest progression-based version of the PANDA_π system (Höller et al. 2018, 2020; Höller and Behnke 2021). We opted for the currently best-performing configurations, i.e., Greedy Best First Search (GBFS) combined with the Relax Composition (RC) heuristic (Höller et al. 2018), which utilizes the classical Add heuristic (Bonet and Geffner 2001) that measures the distance to the solution by accounting for both action applications and method decompositions.

Reported Problem Classes.

We analyze the search nodes in terms of the problem classes outlined in Section *Known Special Cases*. However, since several classes overlap (e.g., any regular problem and any acyclic problem are also tail-recursive), we account for those overlappings. We define the following classes:

- *Primitive*: If all tasks in the current task network are primitive. (I.e., defined as usual.)
- *Regular & Acyclic*: If a problem is regular and acyclic, but not primitive.
- *Regular & Cyclic*: If a problem is regular and cyclic.
- *Acyclic*: If a problem is acyclic but not regular.
- *Tail-recursive*: If a problem is tail-recursive but not acyclic.
- *Undecidable*: If a problem is not tail-recursive.

By adhering to these definitions, all classes are exclusive (and so values should add up to 100% per line for the same test in Table 1).

Results.

A summary of our results (per domain) is provided in Table 1. Note that we only include instances that were solved by both runs: those that don't recompute the TDG (referred to as "Simple" in the table) and those that do (referred to as "Reachable"). This is to make the results comparable because only then the explored search spaces are the same.

This is also why we only report on 22 domains although the IPC TO benchmark set encompasses 23. This discrepancy arises because the "Freecell" domain was excluded due to the absence of problem instances that were solved in both runs.

In the table, we report per domain over all solved instances the minimum percentage of the respective problem class ($\downarrow$), the maximum ($\uparrow$), and the average (μ).

Computation Time. Ideally, class identification of problem classes should exert negligible influence on the computational time. In particular, when our ultimate goal is to choose the most informed heuristic based on the respective

Domain		Undecidable						Tail-recursive						Acyclic						Regular & Cyclic						Regular & Acyclic						Primitive					
		Simple			Reachable			Simple			Reachable			Simple			Reachable			Simple			Reachable			Simple			Reachable			Simple			Reachable		
		↓	↑	μ	↓	↑	μ	↓	↑	μ	↓	↑	μ	↓	↑	μ	↓	↑	μ	↓	↑	μ	↓	↑	μ	↓	↑	μ	↓	↑	μ	↓	↑	μ	↓	↑	μ
Assembly	(30)	0	0	0	0	0	0	82	~A	97	82	~A	97	0	0	0	0	~0	~0	0	7	**2**	0	2	~**0**	0	9	~**0**	~0	9	**2**	~0	9	1	~0	9	1
Barman-BDI	(15)	0	0	0	0	0	0	0	0	0	0	0	0	96	~A	~A	96	~A	~A	0	0	0	0	0	0	~0	2	~0	~0	2	~0	~0	2	~0	~0	2	~0
Blocksw.-GTOHP	(29)	0	~A	**94**	0	~A	**93**	0	0	0	0	0	0	0	99	**5**	0	99	**6**	0	0	0	0	0	0	0	3	~0	0	3	~0	~0	5	1	~0	5	1
Blocksw.-HPDDL	(28)	0	0	0	0	0	0	98	~A	~A	96	~A	~A	0	0	**0**	0	~0	~**0**	0	0	0	0	0	0	0	0	0	~0	2	~0	~0	2	~0	~0	2	~0
Depots	(22)	0	~A	**35**	0	~A	**32**	0	0	**0**	0	24	**1**	0	~A	**63**	~0	~A	**65**	0	0	0	0	0	0	0	0	0	0	0	0	~0	8	2	~0	8	2
Factories	(8)	0	0	0	0	0	0	96	~A	99	96	~A	99	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	~0	4	1	~0	4	1
Hiking	(24)	87	97	92	87	97	92	3	12	7	3	10	7	0	0	**0**	~0	1	**1**	0	0	0	0	0	0	0	0	0	0	0	0	~0	1	1	~0	1	1
Lamps	(16)	0	0	0	0	0	0	80	~A	95	80	~A	95	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	~0	20	5	~0	20	5
Logistics-Learned	(44)	0	0	0	0	0	0	98	~A	~A	98	~A	~A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	~0	2	~0	~0	2	~0
Minecraft Pl.	(1)	0	0	0	0	0	0	84	84	84	84	84	84	15	15	15	15	15	15	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
Minecraft Reg.	(42)	0	0	0	0	0	0	0	0	0	0	0	0	97	~A	99	97	~A	99	0	0	0	0	0	0	~0	1	~0	~0	1	~0	~0	1	~0	~0	1	~0
Monroe FO	(17)	74	~A	95	74	~A	95	0	0	0	0	0	0	0	17	2	0	17	2	0	0	0	0	0	0	0	4	~0	0	4	~0	~0	14	3	~0	14	3
Monroe PO	(8)	25	~A	80	25	~A	80	0	0	0	0	0	0	0	58	14	0	58	14	0	0	0	0	0	0	0	8	2	0	8	2	~0	11	5	~0	11	5
Multiarm-Blocksw.	(74)	0	0	0	0	0	0	98	~A	~**A**	6	96	**50**	0	0	**0**	0	94	**48**	0	0	0	0	0	0	0	0	**0**	~0	60	**2**	~0	2	~0	~0	2	~0
Robot	(20)	0	0	0	0	0	0	67	~A	**97**	4	83	**31**	0	0	**0**	0	~0	~**0**	0	0	0	0	0	0	0	0	**0**	0	96	**67**	~0	33	3	~0	33	3
Rover	(21)	0	~A	**89**	0	~A	**88**	0	0	0	0	0	0	0	97	10	~0	97	10	0	0	0	0	0	0	0	0	0	0	0	0	~0	7	2	~0	7	2
Satellite	(19)	83	99	95	83	99	95	0	0	0	0	0	0	~0	8	2	~0	8	2	0	0	0	0	0	0	0	5	1	0	5	1	~0	8	2	~0	8	2
SharpSAT	(10)	0	0	0	0	0	0	0	0	0	0	0	0	98	~A	99	98	~A	99	0	0	0	0	0	0	0	0	0	0	0	0	~0	2	1	~0	2	1
Snake	(2)	0	0	0	0	0	0	~A	~A	~A	99	~A	~A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	~0	~0	~**0**	~0	~0	~0	~0	~0	~0
Towers	(13)	0	0	0	0	0	0	75	~A	97	75	~A	97	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	~0	25	3	~0	25	3
Transport	(25)	88	~A	97	88	~A	97	0	0	0	0	0	0	0	2	~0	0	2	~0	0	0	0	0	0	0	0	5	1	0	5	1	~0	12	2	~0	12	2
Woodworking	(19)	0	0	0	0	0	0	0	0	0	0	0	0	0	~A	85	0	~A	85	0	0	0	0	0	0	0	75	8	0	75	8	~0	25	6	~0	25	6

Table 1: The minimum (↓), maximum (↑), and average (μ) percentages (%) of undecidability, tail-recursion, acyclicity, regularity, and primitiveness in the search space without and with TDG-recomputation ("Simple" vs. "Reachable"). The computational complexity of properties decreases progressively from left to right. Next to the name of each domain is the number of (solved) problem instances that were used as a basis. "A" (short for "all") represents 100%. Values highlighted in bold show where improvements were achieved due to TDG-recomputation.

search node property, it is essential that the overhead incurred from identifying this special case pays off.

In the "simple" experiment, the total computational time for problem class identification varied across domains, ranging from 0.64% to 42% of total computation time (with an average of 11%), compared to the heuristic's average computation time of 34%. This aligns with expectations, suggesting that class identification times are within acceptable bounds. However, runtimes can still be reduced by optimizing the way classes are identified based on reachable methods (see below).

Conversely, in the "reachable" experiment, the average total computation time for class identification surged to 64%, much overshadowing the heuristic computation time, which averaged 14%. This indicates that class identification significantly impacts the overall computation time in this context, underlining a need for optimization in re-computing the TDG. That said, we did not put any effort into recomputing the TDG effectively, so there is still lots of possible optimizations that could be done. For example, since we use the RC heuristic anyway, we could have used the RPG computed by it when extracting the reachable methods. However, for the sake of simplicity, we re-computed this anyway. Also, we hypothesize that our TDG construction (based on these reachable methods) *and* the way we identify special cases based on the reachable methods could still be optimized (which is the reason why we don't report individual runtimes).

Thus, our reported results can only be used to show which special cases occur and how frequently, but not to draw conclusions about the feasibility of their computation. They do show however that it will be beneficial to optimize TDG recomputation and the identification of special cases.

Difference between Experiments. Before we report on the main findings of our experiments, namely the frequency of certain problem classes, we report on the impact of TDG recomputation. Overall, Table 1 reveals that only a select few domains exhibit notable differences between the "simple" and "reachable" experiments.

The domain exhibiting the most substantial disparity is "Multiarm-Blocksworld" where the average percentage of tail-recursion in the "simple" experiment approaches 100%. However, in the reachable experiment, this average drops to just half. In the remaining 50%, 48% become acyclic (and non-regular), and 2% become regular & acyclic.

"Robot" is another domain showing significant differences. In the "simple" experiment, it only has tail-recursive and primitive search spaces. However, in the "reachable" experiment the average percentage for tail-recursion decreased dramatically from 97 to 31 in favor of 67% regular and acyclic search nodes.

All in all, it seems that TDG recomputation only pays off in a very few domains, so unless recomputation cost can be reduced significantly, our results suggest that on average this might not pay off for the sake of deploying specialized heuristics.

Frequency of Special Cases. We can see that only 5 out of the 22 domains have 90% or more of undecidable search nodes, with 2 further domains having 80% and 89%. Those are the domains where heuristics dealing with the general

case (in TO HTN planning) would be required. So in the huge majority of domains, at least tail-recursiveness could be exploited.

Domains such as “Blocksworld-GTOHP” and “Hiking” exhibit very high ratios of “undecidability”, which are over 90%. The number of solved instances of these domains is large, indicating that the existing general heuristic is efficient.

Three domains show nearly 100% or exactly 100% acyclicity: “Barman-BDI”, “Minecraft Regular” and “SharpSAT’ in Table 1. As the statistics of acyclic search nodes exclude those that are regular (and cyclic) and primitive, these three domains are actually fully acyclic when looking at the data more closely.

In examining the data on acyclicity, it is evident that not all domains exhibit a uniform pattern. For instance, “Blocksworld-GTOHP” ranges wildly in acyclicity, from being virtually cyclic to fully acyclic, as evidenced by its minimum of approximately 0% and a maximum of 99%. Conversely, “Monroe-FO” and “Satellite-GTOHP” show more restrained ranges, peaking at 17% and 8% respectively. It indicates that there are only relatively few nodes in the domain that are acyclic. While the distribution of acyclicity across domains is varied, certain domains inherently exhibit acyclicity, whereas others only manifest it as the problem simplifies during the search. Nonetheless, acyclicity seems to occur often enough within the search space that it seems beneficial to develop specialized heuristics. It is particularly noteworthy that only one single *domain* seems to be acyclic. For all others, acyclicity only occurs within the search space, but not already for the initial search node.

Another *clear* observation is that regularity seems to only occur extremely rarely. In only one domain the ratio of search nodes is about 67%, in one it is 8%, in most others, however, it is 0%, and in just a few 1% or 2%. This however shows a possible optimization as it means that we can stop the TDG recomputation and special case identification once acyclic search nodes are discovered, thus speeding up computation time.

Related Work

There are two lines of related work: one involves choosing a heuristic based on the current search node, which we have not yet explored but forms the motivation for our study; the other is the identification of special cases per search node.

Regarding the first, we are not aware of any such work in HTN planning (which also would not make much sense at the very moment, since no special case heuristics exist as of now). In classical planning, however, Speck et al. (2021) followed a similar idea: selecting the most promising heuristic per search node. Their work does however not do so based on explicit search node properties, but makes dynamic heuristic selection based on Reinforcement Learning. We, however, propose to make this selection dependent based on the specific search node properties and select heuristics that are designed specifically for the respective case.

The other line of related work is investigating search node-specific properties. We are not aware of work along those lines, other than those that investigate properties of entire problem classes. Such a result would be a special case for us. I.e., if exactly 100% of all search nodes show a property P (like being tail-recursive or stronger), then the problem instance has that respective property. Höller (2021) does report on problem class properties (for total-order HTN problems) per problem instance (cf. his Table 1). He reports on a slightly distinct, albeit stronger, criterion concerning tail-recursiveness called non-selfembedding. Any instance that is non-selfembedding, right-recursive (r-rec), and not left-recursive (l-rec) is also tail-recursive. He also reports on acyclic domains ($\neg rec$), but again only per instance.

Höller (2021) does report on problem class properties (for total-order HTN problems) per problem instance (cf. his Table 1). He reports on a slightly distinct, albeit stronger, criterion to decide tail-recursiveness called non-selfembedding. From the instances that are non-self-embedding, those that are only right-recursive (i.e., neither left-recursive nor left-*and* right-recursive), are tail-recursive. He also reports on acyclic domains (¬rec), but again only per instance. A more detailed discussion can be found in Sec. 5 of his most recent work (Höller 2024).

We would also like to note that the PANDA_π planner prints out whether a problem is totally ordered and acyclic when it starts to solve a problem, though it doesn’t show the properties of any of the other classes.

Conclusion

Based on the total-order IPC 2023 benchmark set, we analyzed the explored search space of a progression-based plan for the frequency of special cases within the problem classes, such as tail-recursive, acyclic, and regular search nodes. Our motivation for this investigation is the search node-specific deployment of specialized heuristics, tailored towards the respective special case. Our empirical findings indicate a high potential for such heuristics aimed at tail-recursiveness and acyclicity due to the high number of such search nodes. However, computation time for the identification of these classes is still relatively high and thus requires consideration as well.

Acknowledgements

Pascal Bercher is the recipient of an Australian Research Council (ARC) Discovery Early Career Researcher Award (DECRA), project number DE240101245, funded by the Australian Government.

References

Alford, R.; Bercher, P.; and Aha, D. W. 2015. Tight Bounds for HTN Planning. In *ICAPS 2015*, 7–15. AAAI Press.

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. S. 2012. HTN Problem Spaces: Structure, Algorithms, Termination. In *SOCS 2012*, 2–9. AAAI Press.

Behnke, G.; Höller, D.; Schmid, A.; Bercher, P.; and Biundo, S. 2020. On Succinct Groundings of HTN Planning Problems. In *AAAI 2020*, 9775–9784. AAAI Press.

Bercher, P.; Alford, R.; and Höller, D. 2019. A Survey on Hierarchical Planning - One Abstract Idea, Many Concrete Realizations. In *IJCAI 2019*, 6267–6275. IJCAI.

Bercher, P.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An Admissible HTN Planning Heuristic. In *IJCAI 2017*, 480–488. IJCAI.

Bercher, P.; and Höller, D. 2018. Tutorial: Introduction to Hierarchical Task Network (HTN) Planning. ICAPS. Available online. Accessed: 25 March 2024.

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

Elkawkagy, M.; Bercher, P.; Schattenberg, B.; and Biundo, S. 2012. Improving Hierarchical Planning Performance by the Use of Landmarks. In *AAAI 2012*, 1763–1769. AAAI Press.

Elkawkagy, M.; Schattenberg, B.; and Biundo, S. 2010. Landmarks in Hierarchical Planning. In *ECAI 2010*, 229–234. IOS Press.

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

Geier, T.; and Bercher, P. 2011. On the Decidability of HTN Planning with Task Insertion. In *IJCAI 2011*, 1955–1961. AAAI Press.

Höller, D. 2021. Translating Totally Ordered HTN Planning Problems to Classical Planning Problems Using Regular Approximation of Context-Free Languages. In *ICAPS 2021*, 159–167. AAAI Press.

Höller, D. 2024. The Toad System for Totally Ordered HTN Planning. *JAIR 2024*, (80): 613–663.

Höller, D.; and Behnke, G. 2021. Loop Detection in the PANDA Planning System. In *ICAPS 2021*, 168–173. AAAI Press.

Höller, D.; and Bercher, P. 2021. Landmark Generation in HTN Planning. In *AAAI 2021*, 11826–11834. AAAI Press.

Höller, D.; Bercher, P.; and Behnke, G. 2020. Delete- and Ordering-Relaxation Heuristics for HTN Planning. In *IJCAI 2020*, 4076–4083. IJCAI.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2018. A Generic Method to Guide HTN Progression Search with Classical Heuristics. In *ICAPS 2018*, 114–122. AAAI Press.

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

Nebel, B.; and Bäckström, C. 1994. On the Computational Complexity of Temporal Projection, Planning, and Plan Validation. *AIJ 1994*, 66(1): 125–160.

Olz, C.; and Bercher, P. 2023. A Look-Ahead Technique for Search-Based HTN Planning: Reducing the Branching Factor by Identifying Inevitable Task Refinements. In *SoCS 2023*, 65–73. AAAI Press.

Olz, C.; Höller, D.; and Bercher, P. 2023. The PANDADealer System for Totally Ordered HTN Planning in the 2023 IPC. In *IPC: Planner Abstracts – Hierarchical Task Network (HTN) Planning Track (IPC)*.

Speck, D.; Biedenkapp, A.; Hutter, F.; Mattmüller, R.; and Lindauer, M. 2021. Learning Heuristic Selection with Dynamic Algorithm Configuration. In *ICAPS 2021*, 597–605. AAAI Press.

Tan, X.; and Gruninger, M. 2014. The Complexity of Partial-Order Plan Viability Problems. In *ICAPS 2014*, 307–313. AAAI Press.

Wu, Y. X.; Lin, S.; Behnke, G.; and Bercher, P. 2022. Finding Solution Preserving Linearizations For Partially Ordered Hierarchical Planning Problems. In *33rd PuK Workshop “Planen, Scheduling und Konfigurieren, Entwerfen”*.

Wu, Y. X.; Olz, C.; Lin, S.; and Bercher, P. 2023. Grounded (Lifted) Linearizer at the IPC 2023: Solving Partial Order HTN Problems by Linearizing Them. In *IPC 2023*.