Making Translations to Classical Planning Competitive with Other HTN Planners

Gregor Behnke[1,2], Florian Pollitt[1], Daniel Höller[3], Pascal Bercher[4], Ron Alford[5]

[1]University of Freiburg, Germany,
[2]ILLC, University of Amsterdam, The Netherlands,
[3]Saarland University, Saarland Informatics Campus, Saarbrücken, Germany,
[4]The Australian National University, College of Engineering & Computer Science, Canberra, Australia,
[5]MITRE, McLean, VA, USA
g.behnke@uva.nl, florian.pollitt@mars.uni-freiburg.de, hoeller@cs.uni-saarland.de,
pascal.bercher@anu.edu.au, ralford@mitre.org

Abstract

Translation-based approaches to planning allow for solving problems in complex and expressive formalisms via the means of highly efficient solvers for simpler formalisms. To be effective, these translations have to be constructed appropriately. The current existing translation of the highly expressive formalism of HTN planning into the more simple formalism of classical planning is not on par with the performance of current dedicated HTN planners. With our contributions in this paper, we close this gap: we describe new versions of the translation that reach the performance of state-of-the-art dedicated HTN planners. We present new translation techniques both for the special case of totally-ordered HTNs as well as for the general partially-ordered case. In the latter, we show that our new translation generates only linearly many actions, while the previous encoding generates and exponential number of actions.

Introduction

Hierarchical Task Network (HTN) planning has attracted increased interest in the last couple of years (Bercher, Alford, and Höller 2019), yet the amount of research in solving HTN problems is still lacking behind the vast amount of research done in classical planning. The sophisticated solving techniques in classical planning have spawned many techniques that reuse or extend them in the field of HTN planning. Some extend them, e.g. for grounding and reachability analysis (Behnke et al. 2020), or when encoding problems as propositional logic (Behnke, Höller, and Biundo 2019; Behnke 2021b) or IP/LP (Höller, Bercher, and Behnke 2020); others apply them directly, e.g. to calculate heuristics (Höller et al. 2018). There are also approaches to translate HTN planning problems to classical planning problems directly (Alford, Kuter, and Nau 2009; Alford et al. 2016; Höller 2021). That way, solvers from classical planning can be applied. To overcome the differences in expressiveness (Höller et al. 2014; 2016), there are two different approaches: the work by Alford et al. (2016) bounds the HTN problem before the translation based on the *progression bound* (Alford et al. 2012; Alford, Bercher, and Aha 2015) – the maximum number of tasks that a task network could possibly contain when performing progression search. This restricts possible task networks to a fixed size, which can be represented in the state of a classical problem. Höller (2021) over-approximates the set of solutions to the HTN problem and verifies whether a solution generated by the classical system is actually an HTN solution.

We follow Alford et al. (2016), whose approach we call HTN2STRIPS. While recent results show that it is competitive with the winner of the track on totally-ordered HTN planning of the recent International Planning Competition in terms of coverage (Höller 2021), there are some drawbacks.

- First, the size of the target model highly depends on the progression bound. When it is chosen too low, the classical planner will not find a solution: and the whole process needs to be redone with a higher bound. A larger bound increases the size of the classical model and may make it harder to solve. However, the bound can be influenced by model transformations prior to translation.
- Second, HTN2STRIPS performs the translation on the *lifted* model and outputs a *lifted* classical planning problem. The resulting problem may be hard to ground for the classical grounder. This is even more of a problem since grounding is redone for different bounds.
- Third, when translating partially-ordered HTN problems, the classical model gets harder to solve.

In this paper, we show how to make bound-based classical encodings of HTN planning problems competitive with other solvers from the literature both for totally-ordered and partially-ordered problems. We first show that it is beneficial to apply the translation on the model grounded by an HTN grounder. Using a grounder that's specifically designed to HTN planning problems exploits the (hierarchical) problem structure, which not only makes the process much faster, but also generates much smaller models. Grounding also does not need to be redone when increasing the bound. Further, having a ground model enables us to perform several steps that decrease the size of the classical model as well as the progression bound that would be much harder on the lifted model. Second, we show that a recently-introduced transformation of the HTN model decreases the progression bound without changing the set of solutions to the HTN model. Third, we introduce improved translations for the case of partially-ordered HTN models that make the translated problem simpler to solve for the classical planner.

Copyright © 2022, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Preliminaries

This section describes our formal framework. We combine the SAS$^+$ formalism by Bäckström and Nebel (1995) with the HTN formalism by Geier and Bercher (2011).

Classical Planning and SAS$^+$

In the SAS$^+$ formalism, states are described based on a set of variables $\mathcal{V}$. Each variable $v \in \mathcal{V}$ has an associated (finite) domain D_v with $2 \leq |D_v| < \infty$. A partial state p is a partial assignment of variables v to values $p(v) \in D_v$. We write $s = [v_1 \mapsto x_1, \ldots, v_n \mapsto x_n]$ to denote a partial state with $s(v_1) = x_1, \ldots, s(v_n) = x_n$. For two partial states s_1, s_2 we denote with $s_3 = s_1 \circ s_2$ the partial state such that $s_3(v) = s_2(v)$ iff s_2 defines a value for v and $s_3(v) = s_1(v)$ iff s_1 defines a value for v, but not s_2. A complete state (or *state* for short) is a partial state that assigns a value for each variable. An action $a = \langle pre_a, \mathit{eff}_a \rangle$ is a pair of a partial state pre_a and a set of conditional effects eff_a. A conditional effect is a formula of the type $p \triangleright e$, where p and e are partial states. If p is empty, we will as a simplification just write e as the (non-conditional) effect. Two effects of the same action may not set different values for the same variable, i.e. we require the effects of an action to be well-defined. The action a is applicable in a state s iff $pre_a \subseteq s$, i.e., if $s(v_i) = x_i$, $\forall [v_i \mapsto x_i] \in pre_a$. If a is applicable, applying a in s yields a new state $\gamma(s, a) = s \circ \{e \mid p \triangleright e \in \mathit{eff}_a, p \subseteq s\}$. The initial state s_0 is a complete state while the goal definition g is a partial state. A plan $\pi = \langle a_1, \ldots, a_n \rangle$ is a sequence of actions such that a sequence of states $\sigma = \langle s_0, \ldots, s_n \rangle$ exists where: (1) $g \subseteq s_n$, (2) $\forall i \in \{1, \ldots n\}$, a_i is applicable in s_{i-1}, and (3) $s_i = \gamma(s_{i-1}, a_i)$ for $0 < i \leq n$.

Hierarchical Task Networks

In HTN planning, we distinguish two types of tasks: the set of actions A (also called primitive tasks) and the set of abstract tasks C (or compound tasks). We assume state transition semantics for actions as given in the SAS$^+$ formalism.

Task Networks (TNs) are partially-ordered multi-sets of tasks. A TN $(T, \alpha, \prec)$ consists of a set T of task identifiers, a function α mapping the task ids to tasks $\alpha : T \rightarrow A \cup C$, and a partial order $\prec$ on T. *Decomposition methods* are used to decompose abstract tasks. A method (c, tn) describes that the abstract task c can be decomposed into the TN tn – the method's *subtasks*. The set of methods is denoted with M. Applying a method to an abstract task c in a TN replaces c with the method's subtasks. These subtasks inherit the relative ordering of c with respect to other tasks in the TN. A method $m = (c, tn)$ decomposes a TN $tn_1 = (T_1, \prec_1, \alpha_1)$ including a task $t \in T_1$ with $\alpha_1(t) = c$ into a TN tn_2 defined as follows. Let $tn' = (T', \prec', \alpha')$ be a TN that is equal to tn but using ids not contained in the decomposed network (i.e. $T_1 \cap T' = \emptyset$).

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

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

An HTN planning problem is defined as $P = (F, C, A, M, s_0, g, t_I)$. The two elements s_0 and t_I define the initial state and the initial TN tn_I – which is a task network solely containing the task t_I. A solution to the problem is a TN $tn = (T, \alpha, \prec)$ with:

- $tn_I \rightarrow^* tn$, i.e. it can be obtained by decomposing the initial task network.
- $\forall t \in T : \alpha(t) \in A$, i.e. all tasks are primitive.
- There is a sequence $t_{i_1} t_{i_2} \ldots t_{i_n}$ of all task identifiers in T that satisfies the ordering relation $\prec$ such that $\alpha(t_{i_1})\alpha(t_{i_2}) \ldots \alpha(t_{i_n})$ is a plan leading from s_0 to a state s' in which the goal g holds, i.e., $g \subseteq s'$.

An HTN planning problem is *totally-ordered* iff the ordering relations of the subtasks of all methods are total, i.e. the orderings are linear paths.

Translations to Classical Planning

We build on the work by Alford et al. (2016), which describes a bound-based translation of HTN planning problems to classical planning – HTN2STRIPS. It is based on the following observation: HTN planning problems are commonly solved by a progression search in which a current TN and a current state are maintained – both of which get updated during search. One can then either apply a decomposition method to any of the *first* abstract tasks of the TN (those not preceded by any other task) or apply any of the *first* primitive actions to the current state (thus progressing/updating it) and remove the action from the TN. We have solved the problem once the current TN is empty – the plan is then the sequence of actions that lead to that empty network. Since HTN planning is in general undecidable (Erol, Hendler, and Nau 1996), the progression search space (or any other) can be infinitely large and thus cannot be translated into an equivalent classical planning problem. Alford et al. (2012) proved that if the problem's task hierarchy is of a certain structure (called tail-recursive), then we know that there is a maximum size that task networks may grow to under progression. This is what is referred to as the *maximum progression bound*. Note that even when a problem is not tail-recursive, we can still easily enforce a bound on progression and increase it if it was not sufficient. Assume that we bound the number of tasks in the current TN by a number pb – the progression bound. The number of non-isomorphic task networks that can be derived via progression search becomes finite and can thus be represented in the state of a classical planning problem. Such a state consists of two parts: one describing the current original state of the problem and one describing the current TN, i.e., the tasks that still need to be processed and their ordering relation. HTN2STRIPS translates the rest of the instance as follows: First, the original actions get new preconditions such that they are only applicable when they are in the current TN and have no predecessors in the ordering relation. Second, the methods from the original model are translated to new actions that model the process of decomposition on the TN representation. In addition to the HTN2STRIPS encoding for partially-ordered models, HTN2STRIPS also features a specialized encoding for totally-ordered HTN planning.

By iteratively increasing a progression bound, the translation process can be used to solve arbitrary HTN planning problems. For satisficing planning, we can stop as soon as we have found a solution. For each particular solution, we call the smallest progression bound necessary to derive it its *minimum progression bound*. Ideally, we would try only the minimum progression bound of any solution. Unfortunately, determining this number is hard, which is the cause for the bound iteration. Alford et al. (2016) have presented an under-approximation of this minimum progression bound, which we use for all planners in this paper. If we can bound the maximum progression bound from above, we only need to check this bound to see if the problem is solvable. For problems with a large maximum progression bound, solutions may be found for significantly smaller bounds and as such – similar to SAT-based planning – iteration may be more effective. Further, maximum progression bounds do not always exist. We know only for tail-recursive problems that they must have a maximum progression bound, which can be computed or (over-)estimated from the problem.

Recently, a second translation from (totally-ordered) HTN problems to classical problems has been introduced (Höller 2021), which approximates the set of solutions. The translation completely blends the hierarchy into the state, i.e., there are no additional actions that mimic method application, and no state features that directly correspond to tasks.

When and How to Ground?

The translations of HTN2STRIPS (Alford et al. 2016) were described and implemented in a lifted-to-lifted fashion. The (lifted) HTN input problem is translated into a lifted classical planning problem. The employed classical planner grounds the (classical) problem and attempts to solve it. If multiple progression bounds pb are tested, the problem is grounded multiple times – once for each pb. The problems for the different pbs are however extremely similar as they differ only in the number of ID objects in the instance. Consequently, the groundings of the problems are very similar, but still have to be re-calculated for each pb, each grounding a potentially expensive operation (Gnad et al. 2019).

Classical grounders seem to be ill-equipped for grounding some of the translated HTN planning problems. The problem lies in that (most) classical grounders like the one of Fast Downward (Helmert 2006) only perform a *forward pass*, i.e., they consider all actions that can be reached under delete-relaxation from the initial state. A translated HTN problem however usually has many dead-end actions – not appearing in any solution, which are not pruned by such an approach. Fortunately, there are dedicated HTN grounders (Ramoul et al. 2017; Behnke et al. 2020). Thus our first main change compared to HTN2STRIPS is that we *start* by grounding the HTN planning problems with the HTN grounder by Behnke et al. (2020). Only thereafter, we translate the problem into SAS$^+$ problems.

To increase the utility of the HTN grounder for this setup, we have added an option to infer SAS$^+$ variables. For this we used the lifted FAM-groups computed by Fišer (2020). We create SAS$^+$ variables via a greedy procedure. We first compute all possible SAS$^+$ variables based on the FAM-groups and sort them by size. We then take the variable v with the largest D_v and remove all variables from consideration that have a non-empty intersection with D_v. We then repeat the process until all variables have been created. For remaining facts, we generate boolean variables.

To summarize, this approach results in the following advantages: (1) We ground once, regardless of how many bounds are tried, (2) We can use a grounder specialized to HTN planning that is faster and generates a smaller model. There is also another advantage, as will be discussed in the next sections. Since the ground model is much simpler, it allows us to describe some optimizations – like method compression – clearly, while they are extremely complicated in the lifted setting as we e.g. have to deal with corner cases.

TO-HTN Translations

As the main contribution of this paper, we formally describe the grounded HTN-to-SAS$^+$ translation and significant performance improvements. We start with the simplest case: totally-ordered HTN (TO-HTN) planning problems. This sub-class of HTNs has received significant attention in past research (e.g. Nau et al.; Schreiber (1999; 2021)) and was even featured in a separate track at the 2020 International Planning Competition (Behnke, Höller, and Bercher 2021).

Basic Encoding

We start by describing the basic HTN2STRIPS encoding for TO-HTNs developed by Alford et al. (2016). We intertwine this description already with discussing how to translate a grounded TO-HTN into an SAS$^+$ problem.

The HTN2STRIPS encoding for TO-HTN problems represents the current TN as a stack with a limited size. In the grounded representation we do the same. Given the progression bound pb, we create pb positions for the stack. At each stack position, there can be either one specific task or none. Further, the top of the stack is any one of the possible positions. These structures naturally form SAS$^+$ variables (pos_i and $stack$, resp.). Performing the translation on the grounded level allows us to create appropriate SAS$^+$ variables for the encoding without relying on them being automatically discovered by a SAS$^+$ inference mechanism of the grounder. Note that this also improves the general readability of the translation. In the original lifted translation, the stack and its top had to be represented with boolean predicates which can be quite unnatural to read.

This is especially important as current automatic inference mechanisms for SAS$^+$ variables fail to properly detect these variables. Fast Downward's SAS$^+$ inference (Helmert 2006) fails to detect any of the SAS$^+$ groups pertaining to the positions on the stack. The inference method based on the lifted FAM groups by Fišer (2020) partially detects the SAS$^+$ groups, but fails if they contain more than 100 elements in the default configuration. This might be problematic as several classical planning techniques, like merge and shrink (Helmert et al. 2014) or decoupled search (Gnad and Hoffmann 2018), that require a good SAS$^+$ representation of the problem to achieve good performance.

Encoding the TO-HTNs actions and methods then works as follows. An action a can be executed if a is the top element of the stack, which it removes, and moves the top one position down. A method m can be applied if the top element of the stack is the task it decomposes. It then removes that task from the stack, pushes its task sequences, and moves the top of the stack to the last pushed task (i.e. the first task of the method). Initially, the stack contains only t_I and we reach the goal if the stack is empty and we have reached the state-based goal.

Consider that we are given a grounded TO-HTN problem $P = (F, C, A, M, s_0, g, t_I, prec, add, del)$ and a progression bound pb. We then construct an SAS$^+$ problem $P^{pb} = (\mathcal{V}^{pb}, \mathcal{A}^{pb}, s_0^{pb}, g^{pb})$ as follows. With $\dot\cup$ we denote a disjoint union.

- $\mathcal{V}^{pb} = F \,\dot\cup\, \{stack\} \,\dot\cup\, \{pos_i \mid 1 \leq i \leq pb\}$
  $D_{stack} = \{top_i \mid 0 \leq i \leq pb\}$
  $D_{pos_i} = \{c \mid c \in C \,\dot\cup\, A \,\dot\cup\, \{noTask\}\}$
- $s_0^{pb} = s_0 \circ [stack \mapsto top_1, pos_1 \mapsto t_I]$
  $\circ\, [pos_i \mapsto noTask \mid 2 \leq i \leq pb]$
- $g^{pb} = g \circ [stack \mapsto top_0]$
- $\mathcal{A}^{pb} = \{a_i \mid a \in A, 1 \leq i \leq pb\} \cup$
  $\{m_i \mid m = (c, (T, \prec, \alpha)) \in M,$
  $1 \leq i \leq \min\{pb, pb - |T| + 1\}\}$
  - $pre_{a_i} = pre_a \circ [stack \mapsto top_i, pos_i \mapsto a]$
    $\mathit{eff}_{a_i} = \mathit{eff}_a \circ [stack \mapsto top_{i-1}, pos_i \mapsto noTask]$
  - for $m = (c, (T, \prec, \alpha)) \in M$ with $|T| = 0$ we get the following preconditions and effects for the m_i:
    * $pre_{m_i} = [stack \mapsto top_i, pos_i \mapsto c]$
    * $\mathit{eff}_{m_i} = [stack \mapsto top_{i-1}, pos_i \mapsto noTask]$
  - for the actions m_i with $m = (c, (T, \prec, \alpha)) \in M$, $0 < |T| = k \leq pb$, let $t_1, ..., t_k$ be the task identifiers in T as arranged by the total order $\prec$. Then:
    * $pre_{m_i} = [stack \mapsto top_i, pos_i \mapsto c]$
    * $\mathit{eff}_{m_i} = [stack \mapsto top_{i+k-1}]$
      $\circ\, [pos_{i+j} \mapsto \alpha(t_{k-j}) \mid 0 \leq j \leq k-1]$

Since this is the grounding of the translation by Alford et al. (2016), soundness and completeness follow directly.

Encoding Size Lastly, we discuss, for this encoding and any following encoding we present, the size of the produced encoding relative to progression bound pb and the size of the original model. The most interesting element is the number of actions. For this encoding, we generate pb new actions per action in the HTN model and up to pb many instances (if $|T| \leq 1$). Note that fewer instances of method actions might be generated for methods with more subtasks. As such, the encoding can contain up to $pb \cdot (|A| + |M|)$ actions, which is linear in both the size of the original HTN problem and the progression bound.

Compressing Methods

After adapting the Base encoding to the grounded case, we next strive for making it empirically as effective as other HTN planners. For doing so, let's consider the encoding of a decomposition method m_i which decomposes the abstract task C into the action a followed by the abstract task B. If the task C is at position i of the stack, the action m_i will cause B to be put to position i of the stack, a to position $i+1$, and the top of the stack to be moved to position $i+1$. In the resulting state at most one action is applicable: either the (original) preconditions of a are satisfied – then a_{i+1} is executable, or not – then we are in a dead-end. The first case is depicted on the left of Fig. 1. Effectively, the method-compilation m_i adds the action a to the stack though we know that we have to remove it as the next step anyway. Instead, we could have applied a directly at the time at which we applied m_i without the detour of pushing it to the stack. This is depicted on the right of Fig. 1. Performing this application seems to be useless at first glance – recent classical planners will detect that only a_{i+1} is applicable and apply it immediately. This compression however has two other side-effects: (1) the overall plan becomes shorter, which might be beneficial for the planner and (2) the minimum progression bound of all solutions might decrease. The second point is quite significant as a lower minimum progression bound for solutions implies both that the minimum and the maximum progression bound might decrease. As such, solutions can be found with smaller progression bounds, requiring fewer calls to the classical planner – notably while omitting the runs with the largest models. To see that the maximum progression bound can actually decrease, consider our example method. If we do not compress m_i and a_{i+1} we need two positions on the stack, while for the compressed version one is enough. We study this in our evaluation.

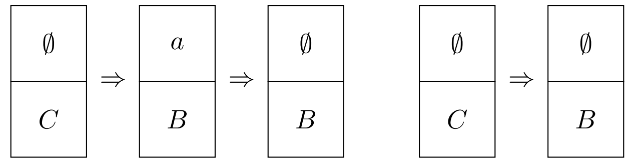

Figure 1: We see two examples for how the stacks behave after a method application. *Left:* Progression without method contraction; *Right:* Progression with method contraction

Next, we formally describe the compression. We not only compress the *first* primitive action, but the full *prefix* of primitive actions this method contains. Given $m = (c, (T, \prec, \alpha))$ with tasks $\alpha(t_1), ..., \alpha(t_{|T|})$ ordered according to $\prec$. Further, let the first ℓ tasks be primitive. We calculate the macro action $a = \langle pre_a, \mathit{eff}_a \rangle$ that simulates the successive application of $\alpha(t_1), ..., \alpha(t_\ell)$. Its correctness follows as it simulates the application of a sequence of actions.

- $pre_a = [(v \mapsto val) \mid \exists i : (v \mapsto val) \in pre_{\alpha(t_i)}$ and $\forall j : 1 \leq j < i \leq \ell, x \in D_v : (v \mapsto x) \notin pre_{\alpha(t_j)} \cup \mathit{eff}_{\alpha(t_j)}]$
- $\mathit{eff}_a = [(v \mapsto val) \mid \exists i : (v \mapsto val) \in \mathit{eff}_{\alpha(t_i)}$ and $\forall j : 1 \leq i < j \leq \ell, x \in D_v : (v \mapsto x) \notin \mathit{eff}_{\alpha(t_j)}]$
- if for some $i < j : (v \mapsto x) \in \mathit{eff}_{\alpha(t_i)}, (v \mapsto y) \in pre_{\alpha(t_j)}, x \neq y$ and not $(v \mapsto y) \in \mathit{eff}_{\alpha(t_l)}$ for some $i < l < j$ then the sequence of actions is not applicable. In this case m is never part of a solution and can be pruned.

The third item correctly detects inapplicable actions as irrespective of the state prior to $\alpha(t_1)$, $v \mapsto y$ cannot hold when

$\alpha(t_j)$ is executed. It was set by action $\alpha(t_i)$ to the different value $v \mapsto x$ and was not reset to y by any intermediate action. Now we can apply a immediately together with our translated method action m_i. This results in m_i' with:

- $pre_{m_i'} = pre_{m_i} \circ pre_a$
- if $|T| = \ell$: $\mathit{eff}_{m_i'} = \mathit{eff}_a \circ [stack \mapsto top_{i-1}, pos_i \mapsto noTask]$
- if $|T| = k > \ell$: $\mathit{eff}_{m_i'} = \mathit{eff}_a \circ [stack \mapsto top_{i+k-1-\ell}]$ $\circ [pos_{i+j} \mapsto \alpha(t_{k-j}) \mid 0 \leq j \leq k-1-\ell]$

Encoding Size Compressing methods can only reduce the number of actions in the encoding. It is possible that no compression can be performed at all, i.e. the size bound for the compressed model is the same as for the Base encoding.

Two-Regularization

Since the overall number of calls to the classical planner as well as the size of the translated problems have presumably a high influence on the overall runtime, we have considered means to further decrease the maximum progression bound: We use *2-Regularisation* as recently introduced by Behnke and Speck (2021). A TO-HTN problem is 2-regular if all methods have at most two subtasks. A given TO-HTN problem can be translated into a 2-regular one in linear time by introducing additional intermediate abstract tasks. We deploy this normal form due to the following advantage:

Theorem 1. *2-Regularisation cannot increase the minimum and maximum progression bounds, but it may decrease them.*

Proof. Consider a sequence of progression steps from tn_I to the empty task network. Consider any two consecutive task networks tn_1 and tn_2 such that tn_2 is created from tn_1 by decomposing its first task c. Let this result in the tasks $t_1, \ldots, t_n$. If $n \leq 2$ we have nothing to show. If $n > 2$, tn_2 has $n - 1 > 1$ more tasks than tn_1. In the 2-Regularisation, we can apply the method decomposing c into t_1, c_1 to the first task c of tn_1. This yields the task network tn_{c_1}, which has the size $|tn_1|+1 < |tn_2|$. We now apply all progressions up to the point where c_1 is the first task in the task network as we did to tn_2. Compared to the non 2-Regularised problem, these TNs contain the same tasks, apart from $t_2, \ldots, t_n$ being replaced by c_1. As such, these TNs cannot contain more tasks. When reaching with c_1, we apply its method decomposing c_1 to t_2, c_2. For the resulting task network, the same reasoning as for tn_{c_1} applies. We can inductively extend this argument up to and including the last decomposition of c_{n-1} into t_{n-1}, t_n.

To see that 2-Regularisation may actually improve progression bounds, consider a planning problem where the initial task t_I decomposes into a sequence of three actions a, b, c. Without 2-Regularisation, its min and max progression bounds are 3. With 2-Regularisation they are only 2, as we will have at most one primitive action and one abstract task on the stack at the same time. □

Note that 2-Regularisation will increase the size of the grounded model. More specifically, it will increase the number of abstract tasks and methods by summed total size of methods exceeding two (formally $\sum_{(c,(T,\prec,\alpha)) \in M} \max\{0, |T|-2\}$). Since this is only a linear increase, the overall encoding remains linear in size.

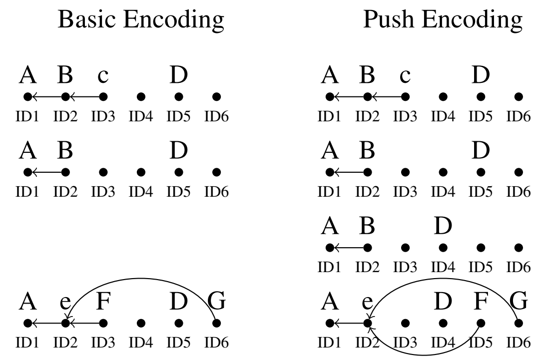

Figure 2: Encoding of the TN containing the tasks A, B, c, and D with the orders $B \prec A$, $c \prec B$ and the subsequent encodings if first c is progressed and then B is decomposed into the tasks e, F, and G with $F \prec e$ and $G \prec e$.

PO-HTN Translations

In the previous section, we have discussed only encodings that exploit the fact that the task networks of all methods contained in the problem describe sequences of tasks. In this section, we turn toward general encodings, i.e. encodings that can deal with arbitrary partially-ordered task networks.

Basic Encoding

The base encoding of HTN2STRIPS (Alford et al. 2016) uses the propositional state of the classical problem to encode a task network. As in the TO encoding, the size of the task network is bounded by pb. Each possible task ID ($task_1, \ldots, task_{pb}$) can either carry a task or not. The partial order $\prec$ of the task network is represented by memorising individual ordering constraints. For each pair of IDs i and j, we memorise whether we know that $task_i$ is ordered before $task_j$ in a variable $constr_{i<j}$. HTN2STRIPS uses a mechanism to avoid computing the transitive closure of the memorised ordering after each decomposition: Ensure that every method has a unique last task, i.e. a task that is ordered strictly after all other tasks in that method. If no such task exists, we add a no-op as an artificial last task. In Fig. 2, we show TNs encoded in this manner. The left column depicts the Base encoding. The top row encodes a TN with the tasks A, B, c, and D, were $c \prec B$ and $B \prec A$. We encode the order with $constr_{3<2} \mapsto yes$ and $constr_{2<1} \mapsto yes$. From the first to the second row, we progress through the primitive c and set $constr_{3<2} \mapsto no$, thus allowing to decompose B. In doing so, we place the method's last task e at the ID at which B was previously. By transitivity, the new tasks F and G, who precede e, also must precede A.

One complication in this encoding is the fact that when executing a decomposition, we do not know which task IDs are used and which ones are free *a priori* – in contrast to the TO encoding where we knew that all IDs after the decomposed one are free. We thus have to generate actions representing methods for each possible combination of free IDs. E.g. in Fig. 2, we place the new tasks F and G at the IDs 3 and 6, but cannot use ID 5. Since the order of the free IDs does not matter, we can map them to the subtasks of the applied method in sorted order to reduce symmetries. To describe this selection, we use the function $\beta_k : \{1, \ldots, pb\} \times \{0, \ldots, \binom{pb-1}{k-1} - 1\} \to \{1, \ldots, pb\}^k$, which shall be an injective function such that for the sequence $\beta_k(i, j) = (a_0, a_1, \ldots, a_{k-1})$ it holds that: $a_0 = i$, $a_0 \neq a_j$ for $j > 1$, and $a_1 < a_2 < \cdots < a_{j-1} < pb$. As in the TO encoding, we treat methods whose task networks contain no tasks as actions without preconditions and effects. We thus omit their explicit description from here on.

- $\mathcal{V}^{pb} = F \,\dot\cup\, \{task_i \mid 1 \leq i \leq pb\} \,\dot\cup$
  $\{constr_{i<j} \mid 1 \leq i, j \leq pb, i \neq j\}$
  $D(task_i) = \{c \mid c \in C \,\dot\cup\, A \,\dot\cup\, \{noTask\}\}$
  $D(constr_{i<j}) = \{yes, no\}$
- $s_0^{pb} = s_0 \circ [task_1 \mapsto t_I] \circ [constr_{i<j} \mapsto no \mid 1 \leq i, j \leq pb] \circ [task_i \mapsto noTask \mid 2 \leq i \leq pb]$
- $g^{pb} = g \circ [task_i \mapsto noTask \mid 1 \leq i \leq pb]$
- $\mathcal{A}^{pb} = \{a_i \mid a \in A, 1 \leq i \leq pb\} \cup \{m_i^j \mid (c, (T, \prec, \alpha)) \in M, |T| = k, 1 \leq i \leq pb, 0 \leq j < \binom{pb-1}{k-1}\}$
  - $pre_{a_i} = pre_a \circ [task_i \mapsto a] \circ$
    $[constr_{j<i} \mapsto no \mid 1 \leq j \leq pb, j \neq i]$
    $\mathit{eff}_{a_i} = \mathit{eff}_a \circ [task_i \mapsto noTask]$
    $\circ [constr_{i<j} \mapsto no \mid 1 \leq j \leq pb]$
  - for the actions m_i^j with $m = (c, (T, \prec, \alpha)) \in M$, $|T| > 0$, let $T = \{t_0, \ldots, t_{|T|-1}\}$ be fixed for m such that $\forall 0 < l \leq |T| - 1 : t_l \prec t_0$ (i.e. t_0 is the last task), and $\beta_k(i, j) = (a_0, \ldots, a_{|T|-1})$ we set:
    - $pre_{m_i^j} = [task_i \mapsto a] \circ [constr_{k<i} \mapsto no \mid 1 \leq k \leq pb, k \neq i] \circ [task_l \mapsto noTask \mid l \in \{a_1, \ldots, a_{k-1}\}]$
    - $\mathit{eff}_{m_i^j} = [task_{a_l} \mapsto \alpha(t_l) \mid 0 \leq l \leq k-1] \circ [constr_{a_l < a_m} \mapsto yes \mid t_l \prec t_m]$

Encoding Size The Basic encoding for partially-ordered task networks generates one instance of each original action per value of the progression bound, i.e. $pb \cdot |A|$ in total. It further generates for every original method $m \in M$ actions of the type m_i^j. Here i is a value from 1 to pb. The second parameter j encodes the selection of a subset of size k of the remaining $pb - 1$ IDs, where k is the number of subtasks of that method. We generate one action for every such $j \in \{0, \ldots, \binom{pb-1}{k-1} - 1\}$. As such, this encoding generates in total $pb \cdot \binom{pb-1}{k-1}$ translated method actions for one ground method with k subtasks. For any fixed k, the value of $\binom{pb-1}{k-1}$ increases exponentially in pb. $\binom{pb-1}{k-1}$ is bounded from above by $pb!$, which in turn is by Stirling's approximation in the order of $\sqrt{pb}\left(\frac{pb}{e}\right)^{pb}$. So to simplify, we can state that for each method, we generate $\mathcal{O}(pb^{\frac{3}{2}+pb})$ actions. This leads to $\mathcal{O}(pb \cdot |A| + |M| \cdot pb^{\frac{3}{2}+pb})$ many encoded actions in total.

Push Encoding

In its original (lifted) variant, the size of the previous Base encoding was, as the TO encoding, linear in the size of the (lifted) model. The encoding, however, produces an exponential amount of ground actions representing methods – since $\binom{pb-1}{k-1}$ scales exponentially in pb. This causes the Base encoding to quickly become impossible to use.

The main problem with the base HTN2STRIPS encoding is that we do not know which IDs are free when applying a decomposition method and can thus be used for the tasks that are added by that method. To reduce the complexity of the model, we would like to assume that if the method adds $k-1$ new tasks (i.e. it has k subtasks), the highest $k-1$ IDs ($task_{pb-k+1}, \ldots, task_{pb}$) are always free and can be used for these new tasks. This way, we eliminate the choice of which IDs to use from the method actions. As such, we only need one encoded action per method and ID of the decomposed task – i.e. only a linear amount. To allow for this assumption, we add new *push actions* which allow for compressing the currently stored TN representation. Since we have to move the ordering constraints as well, we (unfortunately) either require an exponential blow up or – what we chose to do – use conditional effects as shown below. We only need $|A \cup C| \cdot pb$, i.e. linearly many, such push actions – one for each task and ID apart from ID 1. The idea of this encoding is shown in Fig. 2. The TN in the second row has ID 4 occupied by the task D. In the Base encoding, we can use IDs 3 and 6 for the tasks F and G. In the Push encoding, we are forced to use IDs 5 and 6. ID 5 however is not free, to we need to move the task D from ID 5 to ID 4, which we do prior to applying the method – as shown in the right column.

In adding the *push* actions, we have introduced a new source of ambiguity in plans: the order in which push actions are applied. Consider pushing a task from position i to $i-1$. In the next state, it may be possible to push this task again from $i-1$ to $i-2$, but it might now also be possible to push the task at $i+1$ to i. No matter in which order we apply these two actions, we end up in the same state, i.e. applying these actions is commutative. To avoid these redundancies, we perform in-model partial-order reduction. If we have pushed a task from position i to $i-1$ and if the next position $i-2$ is also free, then we should immediately push the task from $i-1$ to $i-2$. To ensure this, we mark the position $i-1$ to have *priority* and only allow a push from a position i if no other position has priority.

- $\mathcal{V}^{pb} = F \,\dot\cup\, \{task_i \mid 1 \leq i \leq pb\} \,\dot\cup\, \{constr_{i<j} \mid 1 \leq i, j \leq pb, i \neq j\} \,\dot\cup\, \{next_i \mid 1 \leq i \leq pb\}$
  $D(task_i) = \{c \mid c \in C \,\dot\cup\, A \,\dot\cup\, \{noTask\}\}$
  $D(constr_{i<j}) = \{yes, no\}$
  $D(next_i) = \{\mathit{indiff}, prio\}$
- $s_0^{pb} = s_0 \circ [task_1 \mapsto t_I] \circ [constr_{i<j} \mapsto no \mid 1 \leq i, j \leq pb] \circ [next_i \mapsto \mathit{indiff} \mid 1 \leq i \leq pb] \circ [task_i \mapsto noTask \mid 2 \leq i \leq pb]$
- $g^{pb} = g \circ [task_i \mapsto noTask \mid \forall i : 1 \leq i \leq pb]$
- $\mathcal{A}^{pb} = \{a_i \mid a \in A, 1 \leq i \leq pb\} \cup \{m_i \mid (c, (T, \prec, \alpha)) \in M, 1 \leq i \leq pb - |T| + 1\} \circ \{push_i^a \mid a \in A, 1 < i \leq pb\}$

– $pre_{a_i} = pre_a \circ [task_i \mapsto a] \circ$
$[constr_{j<i} \mapsto no \mid 1 \leq j \leq pb, j \neq i]$
$\mathit{eff}_{a_i} = \mathit{eff}_a \circ [task_i \mapsto noTask]$
$\circ [constr_{i<j} \mapsto no \mid 1 \leq j \leq pb]$

– for the actions m_i with $m = (c, (T, \prec, \alpha)) \in M$, $|T| > 0$, let $T = \{t_0, \ldots, t_{|T|-1}\}$ be fixed for m such that $\forall 0 < l \leq |T| - 1 : t_l \prec t_0$ (i.e. t_0 is the last task).
  * $pre_{m_i} = [task_i \mapsto a] \circ [constr_{j<i} \mapsto no \mid 1 \leq j \leq pb, j \neq i] \circ [task_l \mapsto noTask \mid pb - |T| + 1 \leq l \leq pb]$
  * $\mathit{eff}_{m_i} = [task_0 \mapsto \alpha(t_0)] \circ [task_{pb-l+1} \mapsto \alpha(t_l) \mid 1 \leq l \leq k-1]$
  $\circ [constr_{pb-l+1<pb-m+1} \mapsto yes \mid t_l \prec t_m, m \neq 0]$
  $\circ [constr_{i<pb-j+1} \mapsto yes \mid 1 \leq j \leq k-1]$

– $pre_{push_i^a} = [task_i \mapsto a] \circ [task_{i-1} \mapsto noTask] \circ [next_j \mapsto indiff \mid j \neq i]$
$\mathit{eff}_{push_i^a} = [task_i \mapsto noTask] \circ [task_{i-1} \mapsto a] \circ$
$[constr_{i<j} \mapsto yes \triangleright constr_{i-1<j} \mapsto yes \mid 1 \leq j \leq pb] \circ$
$[constr_{j<i} \mapsto yes \triangleright constr_{j<i-1} \mapsto yes \mid 1 \leq j \leq pb] \circ$
$[next_i \mapsto indiff] \circ [task_{i-2} \mapsto noTask \triangleright next_{i-1} \mapsto prio] \circ [constr_{i<j} \mapsto no, constr_{j<i} \mapsto no \mid 1 \leq j \leq pb]$

Encoding Size In the Push encoding, we generate three types of actions. As for all the previous encodings, we generate $pb \cdot |A|$ many actions for the original actions of the HTN model. We further generate one push action per original task and value of the progression bound except for 1, i.e. another $(pb-1) \cdot |A \cup C|$ actions. Lastly, for each method m, we generate actions m_i for $i \in \{0, \ldots, pb\}$. As we fix these actions to use the $|T| - 1$ highest IDs for new tasks, there is no variability in selecting the IDs for these subtasks. The only component that varies is the position of the task we decompose as expressed by the parameter i. We only generate pb many actions for each method. This leads to $pb \cdot (|A| + |M|) + (pb-1) \cdot |A \cup C|$ many actions in total. This is linear in pb and not exponential as for the Base encoding.

Theorem 2. *The Push encoding generates exponentially fewer actions than the Base encoding in pb.* □

Parallel Sequences

Höller and Behnke (2021) have observed that most of the available partially-ordered HTN domains – and in particular those of the IPC 2020 – have a very specific structure: Notably, all methods except one are totally-ordered. The sole partially-ordered one decomposes the initial task t_I and contains no ordering constraints at all. As such, any task network that can be derived from t_I consists of a set of sequences of actions that are fully parallel, i.e. that do not have any ordering constraints between them.

If the method decomposes the initial task into k tasks, we interpret the resulting progressions as k independent stacks. As such, we can apply the encoding for the totally-ordered case for each of these stacks individually by duplicating the stack positions k times. For simplicity, we scale in this encoding by the size of each individual stack – i.e. each stack has pb many positions. Apart from this, the encoding works exactly as for the totally-ordered case. The sole exception is that we directly apply the initial method to "prime" the stacks with the correct tasks.

Lastly, we observed that in practice we can also apply this encoding in cases where the domain is not of the parallel-sequences type. In this case we have to guess for each method that is partially-ordered a linearisation and replace the original method with this linearisation. Note that this transformation may in theory eliminate solutions (i.e., it is incomplete), but it seems to be an acceptable approximation in practice (measured by the IPC 2020 benchmark set).[1]

Encoding Size Since this encoding is just multiple totally-ordered encodings without any additional elements, we generate $pb \cdot (|A| + |M|)$ many actions per stack and thus $k \cdot pb \cdot (|A| + |M|)$ in total.

Empirical Evaluation

We have compared our proposed encodings, which we call HTN2SAS[2] with a wide variety of other planners, including the IPC 2020 competitors, on the IPC 2020 benchmark set (Behnke, Höller, and Bercher 2021). Each planner was given 8GiB of RAM and 30 minutes of single-core runtime on an Xeon Gold 6242 CPU per instance. Runtime includes the time for grounding, encoding, and solving by the back-end planner. We compared multiple classical planners as back-ends. We found that the best performing back-end was Fast Downward (Helmert 2006), first performing enforced hill climbing and then lazy greedy search, both with the FF heuristic (Hoffmann and Nebel 2001) and FF preferred operators. We denote this configuration with FF. We have also tested the runner-up of the agile track of the most recent (classical) International Planning Competition (IPC 2018): Saarplan (Fickert et al. 2018), denoted as Saar. We were not (yet) able to use the agile track winner LAPKT-DUAL-BFWS (Frances et al. 2018) as its off-the-shelf version does not allow for providing a SAS$^+$ input directly. We also tested the winner of the satisficing track Fast Downward Stone Soup 2018 (FDSS) (Seipp and Röger 2018). In its default satisficing configuration, FDSS will use all the time allotted to it to find the shortest possible solution. We abort FDSS's portfolio if either a plan has been found or one component planner has shown unsolvability. We did the same for LAMA (Richter and Westphal 2010), the winner of the IPC 2008 satisficing track.

We start with the TO benchmark set of the IPC 2020. We have compared our approach against the five IPC competitors (HyperTensioN (Magnaguagno, Meneguzzi, and de Silva 2021), Lilotane (Schreiber 2021), PDDL4J (Pellier and Fiorino 2021), SIADEX (Fernandez-Olivares, Vellido, and Castillo 2021), pyHiPOP (Lesire and Albore 2021)) as well as against the original encoding HTN2STRIPS, the translation-based approach by Höller (2021) (TOAD), and the most recent versions of progression search (Greedy RC (Höller et al. 2020; Höller and Behnke 2021)) and SAT-based HTN planning (pandaPisatt-1iB (Behnke 2021a)).

[1] See the evaluation for details. Also note that whenever a user chooses that technique of our planner, it outputs a warning that it is technically incomplete to make sure comparisons remain fair.

[2] The code is integrated into the pandaPI system and can be found at https://github.com/panda-planner-dev/pandaPIengine.

		HTN2SAS 2R C FF	Greedy RC ADD	Greedy RC FF	pandaPIsatt-1iB	**HTN2SAS** C FF	TOAD	**HTN2SAS** 2R FF	HyperTensioN	**HTN2SAS** 2R C Saar	**HTN2STRIPS** FF	Greedy RC LMC	Lilotane	**HTN2SAS** FF	**HTN2SAS** 2R C FDSS	**HTN2SAS** 2R C Lama	PDDL4J	SIADEX	pyHiPOP
Assembly	30	**30**	**30**	**30**	6	**30**	**30**	5	3	**30**	24	7	5	5	**30**	4	2	0	1
Barman-BDI	20	14	17	**20**	19	11	16	14	**20**	13	12	15	17	10	12	10	11	**20**	0
BW-GTOHP	30	27	**28**	**28**	25	21	22	25	16	22	21	25	23	23	16	20	16	13	1
BW-HPDDL	30	21	28	25	6	21	21	20	**30**	20	25	14	1	20	24	7	0	0	0
Childsnack	30	26	23	21	23	23	24	25	**30**	21	20	20	28	16	23	0	21	22	0
Depots	30	22	22	27	**28**	24	23	22	24	22	22	24	24	22	20	18	23	22	0
Elevator-Learned	147	**147**	146	146	**147**	**147**	**147**	142	**147**	**147**	117	146	**147**	142	108	**147**	2	11	2
Entertainment	12	**12**	**12**	**12**	**12**	**12**	**12**	**12**	8	**12**	4	**12**	4	**12**	**12**	10	5	0	1
Factories-simple	20	6	**9**	7	8	6	5	5	3	6	6	5	4	5	3	5	0	0	1
Freecell-Learned	60	0	18	**19**	10	11	0	0	0	0	0	0	12	4	0	0	0	0	0
Hiking	30	23	**25**	**25**	22	22	22	19	**25**	12	24	20	23	20	3	8	17	0	0
Logistics-Learned	80	79	48	49	**80**	35	48	31	22	38	52	75	45	26	29	33	0	0	0
Minecraft-Player	20	3	4	4	4	1	1	2	**5**	1	4	1	4	1	0	1	1	3	0
Minecraft-Regular	59	43	43	43	40	41	41	49	**58**	42	55	41	37	45	43	41	23	35	0
Monroe-FO	20	**20**	18	18	**20**	**20**	0	**20**	**20**	0	2	12	**20**	19	2	0	**20**	7	0
Monroe-PO	20	15	8	12	**20**	13	0	13	0	0	1	8	**20**	12	1	0	1	0	0
Multiarm-BW	74	72	72	27	19	69	**74**	47	8	60	**74**	17	4	37	70	20	0	1	0
Robot	20	**20**	**20**	**20**	11	**20**	**20**	17	**20**	**20**	**20**	19	11	17	**20**	11	6	0	1
Rover-GTOHP	30	18	26	22	24	18	9	17	**30**	17	10	18	23	17	8	8	27	**30**	6
Satellite-GTOHP	20	**20**	**20**	17	**20**	10	10	19	**20**	10	7	12	15	10	9	6	**20**	0	7
Snake	20	**20**	**20**	**20**	**20**	**20**	15	19	**20**	18	**20**	**20**	**20**	19	13	17	**20**	7	2
Towers	20	**16**	13	13	8	15	9	14	**16**	12	**16**	13	9	13	14	15	14	11	2
Transport	40	33	32	30	**40**	23	35	32	**40**	32	24	24	34	23	25	26	33	1	18
Woodworking	30	25	28	28	28	19	**30**	25	7	17	7	19	**30**	19	19	14	6	3	4
Coverage	892	**712**	710	663	640	632	614	594	572	572	567	567	560	537	504	421	268	186	46
Normalised Coverage		18.6	**18.8**	18.2	17.0	16.6	15.0	16.0	15.7	14.2	13.8	14.6	14.9	14.2	12.9	9.7	10.0	6.1	1.6
IPC Score		13.6	**15.0**	14.4	13.7	11.7	11.8	11.0	**15.0**	9.0	8.3	10.3	12.5	8.8	7.8	6.7	7.5	4.9	0.9

Table 1: Results on IPC 2020 TO-HTN benchmark set. 2R represents 2-Regularization and C represents compression.

In Tab. 1, we show standard and normalized coverage and the IPC score of the several TO-HTN planners on the IPC 2020 benchmark set. The base version HTN2SAS does – surprisingly – not outperform HTN2STRIPS (537 vs 567 in coverage). We suppose that this is caused by the different way we handle method preconditions. In HTN2STRIPS, they are compiled directly into method actions, while HTN2SAS compiles them into separate primitive actions. This increases the minimum progression bound by one and can make the problem harder to solve. However, the Base encoding of HTN2SAS already has a higher IPC score (8.8) than HTN2STRIPS (8.3). Since the IPC score is a time-score, this shows that the switch from lifted to grounded pays off: we can solve problems faster.

With respect to the three alternative back-end planners, Saarplan solves 572 instances with an IPC Score of 9.0 and a normalised coverage of 14.2. For FDSS, we get 504 solved instances, an IPC Score of 7.8, and a normalised coverage of 12.9, and for LAMA 421, 6.7, and 9.7. This comparably bad performance is interesting. It might open an avenue for future research for the classical planning community where it might help to improve heuristics and search techniques. With respect to the time needed for generating the encoding, out of the 3804 translations performed by R2 C FF, only 120 took longer than 10 seconds with a maximum of 15.981 seconds. The classical planner always took longer solving the translation than the time needed to generate it.

Performing either 2-Regularisation (2R) or method compression (C) already pays off significantly in coverage (567 vs 594/632) and even more so in the IPC score (8.3 vs 11.0/11.7). Combining both techniques leads to a higher coverage than any other planner (712). It is slightly worse in normalised coverage (18.6 vs 18.8 by Greedy RC ADD). In terms of IPC score (13.6), it is not yet one the level of the absolutely best planners (15.0), but would have scored 2nd place if it would have participated in the IPC 2020, being only beaten by HyperTensioN. In the supplement, we present a cactus plot of the planners' runtime. There we can see the reason for this behaviour: HTN2SAS has a relatively slow start – which is punished by the IPC score.

Next, we consider the influence of 2-Regularisation and method compression on the minimum progression bounds, i.e. the bounds actually needed to solve the problems. Method compression has only a small influence over the necessary progression bound, but decreases the value by at least one in 449 out of 536 cases. It decreases by at least two in 144, by at least three in 69, and by at least four in 35 cases. The maximum decrease is seven in eight instances. For 2-Regularisation (without method compression), out of the 532 instances solve by both variants, the bound decreases in 496. It (at least) halves the bound in 338 instances and reduces it to at most a third in 200 instances. The overall maximum is reduced from 129 to 26 and all but 20 (solved) instances can be solved with a bound of less than 10, while 381 previously required a bound of at least 10. On average the bound is reduced to 42.7% of the original, with a median of 40%. If we consider contracting methods in already 2-Regularised domains, we can still observe a decrease in the progression bound. Out of the 590 instances solved by both configurations, in 379 the necessary progression bound

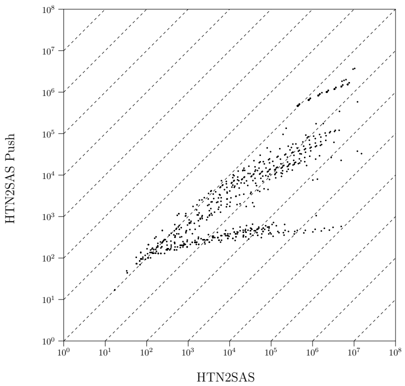

Figure 3: Comparison of the size of the partial-order encodings Base and Push. We plot the number of actions.

		GA* FF th+gi	**HTN2SAS** PSeq* FF	PANDA SAT	**HTN2SAS** Push FF	**HTN2SAS** PSeq FF	SIADEX	**HTN2SAS** FF	**HTN2STRIPS** FF	PYHIPOP
Barman-BDI	20	3	2	8	4	2	**20**	1	3	0
Monroe-FO	25	**24**	22	0	10	10	8	2	0	0
Monroe-PO	25	19	**20**	0	7	7	2	4	0	0
PCP	17	**14**	**14**	13	9	**14**	0	3	5	0
Rover	20	11	4	**16**	8	4	14	4	4	2
Satellite	25	**25**	24	**25**	19	24	**25**	24	23	6
Transport	40	12	8	**23**	15	8	1	8	12	3
UM-Translog	22	**22**	**22**	**22**	**22**	**22**	**22**	21	**22**	21
Woodworking	30	12	9	**17**	11	9	3	11	6	6
Coverage	224	**142**	125	124	105	100	95	78	75	38
Normalised Coverage		**5.9**	5.3	5.1	4.3	4.3	4.2	3.1	3.1	1.6
IPC Score		**4.9**	3.7	3.1	3.0	3.4	4.0	2.6	2.0	1.2

Table 2: Results on IPC 2020 PO-HTN benchmark set.

is still decreased by one and in one instance by two.

Next, we turn to the partially-ordered instances. Here, we compare only against HTN planners that can handle partial order: a progression planner (GA* FF th+gi, (Höller and Behnke 2021)), a SAT-based planner (Behnke, Höller, and Biundo 2019), and two IPC competitors (SIADEX and pyHiPOP). Out of the 224 IPC 2020 instances, 173 have – after grounding – the parallel-sequences structure. Of the remaining 51 instances with more complex partial order, one is an instance of the UM-Translog domain, while the remaining 50 are the instances of both Monroe domains. We present two versions of HTN2SAS employing the parallel sequences encoding. The first (PSeq*) always uses the encoding and is thus potentially incomplete. The second (PSeq) uses the parallel sequences encoding only if the instance actually has the property, else we use the Push encoding.

HTN2SAS's Base encoding already outperforms HTN2STRIPS's encoding both in terms of coverage (78 vs 75) and IPC score (2.6 vs 2.0), cf. Tab. 2. The Push encoding provides a further boost to 105 coverage and an IPC score of 3.0. The complete PSeq configuration lacks behind the Push encoding, but only in coverage (105 vs 100) while it increases the IPC score from 3.0 to 3.4. The incomplete configuration PSeq* performs still better, and notably takes the clear lead in the two Monroe domains which are not of the parallel sequences type. This is due to the fact that both domains are almost parallel sequences with only a few methods that introduce partial order, which in most cases can be handled by guessing a linearisation already in them model (which is exactly what PSeq* does). We do not fully reach the currently best performing planner GA* FF th+gi, but are close.

Lastly, we have investigated the impact of the Push and Parallel Sequences encodings on the size of the encoding. We have extracted for every encoding that was performed during the evaluation runs on the 2020 IPC domains the number of actions in the produced model. Note that this yields multiple data points per instances as each planner encodes the problems for multiple progression bounds. There were 839 formulae that were both constructed by the Base and the Push encoding. We show their sizes in Fig. 3. In 115 cases, the Push encoding generated more actions, but 65 out of these generated less than 7,500 actions in both models. The remaining 50 instances are the encodings for each of the 50 Monroe instances at progression bound $pb = 3$. Here the Base encoding generated between 437,220 and 480,366 actions, while Push encoding created between 468,220 and 512,928. This can be explained by the size of the grounded HTN model, which contains approx. 60,000 actions, 4,000 compound tasks, and 52,000 methods. In an encoding for a low progression bound (3 in this case), the number of actions dominates the encoding due to the additionally needed push actions. Of the remaining 717 cases, were we get a reduction, the highest absolute reduction is in Rover instances Nr. 11 at $pb = 24$, where we reduce from 15,121,320 actions to 32,938. In 216 cases we reduce by more than one order of magnitude, in 53 by more than two, and in 10 cases by more than three. Next, we compare the base and the PSeq* encodings, based on the 821 translations both performed. The PSeq* encoding is larger in only 28 cases, which a maximum number of actions of 23,760 in once instance and 3,740 for the remaining 27. Of the remaining 755 cases, the reduction is at least one order of magnitude in 142 cases, two in 40, and three in 10. If we compare Push and PSeq* directly, we can use 1,039 data points. In 637 cases the Push encoding was smaller and in 395 cases the PSeq* encoding. In 7 cases both models were exactly equal in size. The difference between both models was never larger than one order of magnitude.

Conclusion

We have presented translations of grounded HTN planning problems into (classical) SAS^+ planning problems and described several optimisations w.r.t. the current state of the art. With our contributions, translation-based HTN planners are now on par with the performance of dedicated HTN planners and the IPC 2020 planners, while lacking behind before. Future work might investigate (partial) 2-Regularisation for partially-ordered domains and more compact encodings for the partial-order case. The impact of different classical planning techniques should be studied further, which might lead to dedicated techniques and heuristics for the encodings.

References

Alford, R.; Behnke, G.; Höller, D.; Bercher, P.; Biundo, S.; and 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 2016)*, 20–28. AAAI Press.

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

Alford, R.; Kuter, U.; and Nau, D. 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 2009)*, 1629–1634. AAAI Press.

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. S. 2012. HTN Problem Spaces: Structure, Algorithms, Termination. In *Proceedings of the 5th Annual Symposium on Combinatorial Search (SoCS 2012)*, 2–9. AAAI Press.

Bäckström, C.; and Nebel, B. 1995. Complexity Results for SAS+ Planning. *Computational Intelligence*, 11(4): 625–656.

Behnke, G. 2021a. Block Compression and Invariant Pruning for SAT-based Totally-Ordered HTN Planning. In *Proceedings of the 31st International Conference on Automated Planning and Scheduling (ICAPS 2021)*, 25–35. AAAI Press.

Behnke, G. 2021b. Block Compression and Invariant Pruning for SAT-based Totally-Ordered HTN Planning: Supplement. Technical Report 297, University of Freiburg, Department of Computer Science.

Behnke, G.; Höller, D.; and Bercher, P., eds. 2021. *Proceedings of 10th International Planning Competition: Planner and Domain Abstracts – Hierarchical Task Network (HTN) Planning Track (IPC 2020)*.

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

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

Behnke, G.; and Speck, D. 2021. Symbolic Search for Optimal Total-Order HTN Planning. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI 2021)*, 11744–11754. AAAI Press.

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

Erol, K.; Hendler, J.; and Nau, D. 1996. Complexity results for HTN planning. *Annals of Mathematics and AI*, 18(1): 69–93.

Fernandez-Olivares, J.; Vellido, I.; and Castillo, L. 2021. Addressing HTN Planning with Blind Depth First Search. In *Proceedings of 10th International Planning Competition: planner and domain abstracts (IPC 2020)*.

Fickert, M.; Gnad, D.; Speicher, P.; and Hoffmann, J. 2018. SaarPlan: Combining Saarland's Greatest Planning Techniques. In *IPC2018 – Classical Tracks: Planner Abstracts for the Classical Tracks in the International Planning Competition 2018*, 10–15.

Fišer, D. 2020. Lifted Fact-Alternating Mutex Groups and Pruned Grounding of Classical Planning Problems. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI 2020)*, 9835–9842. AAAI Press.

Frances, G.; Geffner, H.; Lipovetzky, N.; and Ramirez, M. 2018. Best-First Width Search in the IPC 2018: Complete, Simulated, and Polynomial Variants. In *IPC2018 – Classical Tracks: Planner Abstracts for the Classical Tracks in the International Planning Competition 2018*, 22–26.

Geier, T.; and Bercher, P. 2011. On the Decidability of HTN Planning with Task Insertion. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, 1955–1961. AAAI Press.

Gnad, D.; and Hoffmann, J. 2018. Star-topology decoupled state space search. *Artificial Intelligence*, 257: 24–60.

Gnad, D.; Torralba, A.; Domínguez, M.; Areces, C.; and Bustos, F. 2019. Learning how to ground a plan–partial grounding in classical planning. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI 2019)*, 7602–7609. AAAI Press.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research (JAIR)*, 26: 191–246.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *Journal of the ACM*, 61(3).

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research (JAIR)*, 14: 2531–302.

Höller, D. 2021. Translating Totally Ordered HTN Planning Problems to Classical Planning Problems Using Regular Approximation of Context-Free Languages. In *Proceedings of the 31st International Conference on Automated Planning and Scheduling (ICAPS 2021)*, 159–167. AAAI Press.

Höller, D.; and Behnke, G. 2021. Loop Detection in the PANDA Planning System. In *Proceedings of the 31st International Conference on Automated Planning and Scheduling (ICAPS 2021)*, 168–173. AAAI Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language Classification of Hierarchical Planning Problems. In *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI 2014)*, volume 263, 447–452. IOS Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2016. Assessing the Expressivity of Planning Formalisms through the Comparison to Formal Languages. In *Proceedings of the*

26th International Conference on Automated Planning and Scheduling, (ICAPS 2016), 158–165. AAAI Press.

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

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

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

Lesire, C.; and Albore, A. 2021. pyHiPOP – Hierarchical Partial-Order Planner. In *Proceedings of 10th International Planning Competition: planner and domain abstracts (IPC 2020)*.

Magnaguagno, M. C.; Meneguzzi, F.; and de Silva, L. 2021. HyperTensioN: A three-stage compiler for planning. In *Proceedings of 10th International Planning Competition: planner and domain abstracts (IPC 2020)*.

Nau, D.; Cao, Y.; Lotem, A.; and Munoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI 1999)*, 968–973.

Pellier, D.; and Fiorino, H. 2021. Totally and Partially Ordered Hierarchical Planners in PDDL4J Library. In *Proceedings of 10th International Planning Competition: planner and domain abstracts (IPC 2020)*.

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

Richter, S.; and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39: 127–177.

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

Seipp, J.; and Röger, G. 2018. Fast Downward Stone Soup 2018. In *IPC2018 – Classical Tracks: Planner Abstracts for the Classical Tracks in the International Planning Competition 2018*, 72–74.