Block Compression and Invariant Pruning for SAT-based Totally-Ordered HTN Planning

Gregor Behnke
University of Freiburg
Freiburg im Breisgau, Germany
behnkeg@informatik.uni-freiburg.de

Abstract

Translations into propositional logic are currently one of the most efficient techniques for solving Totally-Ordered HTN planning problems. The current encodings iterate over the maximum allowed depth of decomposition. Given this depth, they compute a tree that represents all possible decompositions up to this depth. Based on this tree, a formula in propositional logic is created. We show that much of the computed tree is actually useless as it cannot possibly belong to a solution. We provide a technique for removing (parts of) these useless structures using state invariants. We further show that is often not necessary to encode all leafs of this tree as separate timesteps, as the prior encodings did. Instead, we can compress the leafs into blocks and encode all leafs of a block as one timestep. We show that these changes provide an improvement over the state-of-the-art in HTN planning.

Introduction

HTN planning problems specify the problem to be solved twofold: They (i) provide a description of how the execution of actions changes the (propositional) state of the world and (ii) describe the ways and means by which a plan must be derived. These derivations are performed using *decomposition methods* (or methods for short) that are akin to derivation rules of formal grammars. A solution to an HTN planning problem is a sequence of actions that is executable and that can be derived via application of decomposition methods to the initial abstract task. In HTN planning the initial abstract task describes the goal of the planning problem. For general HTN planning, each method provides a partially-ordered set of tasks which can replace an abstract task A. A significant body of research deals with a subclass of HTN planning: Totally-Ordered HTN planning (TOHTN). Notably, the IPC 2020 had a separate track solely dedicated to TOHTN planning with more participants and domains than the general HTN track. In TOHTN planing, methods are restricted to describing sequences of tasks which can replace abstract tasks. In contrast to general HTN planning, which is undecidable, TOHTN is only EXPTIME-complete (Erol, Hendler, and Nau 1996). In this paper, we consider TOHTN planning.

There are three prevalent approaches to solve HTN planning problems: Plan-space search, progression-search, and translations into propositional logic. We focus on the latter, namely SAT-based TOHTN planning. Recent work proposed two grounded encoding techniques (Schreiber et al. 2019; Behnke, Höller, and Biundo 2018a). We don't consider the lifted encoding by Schreiber (2021b) as we restrict ourselves to grounded planning. Both (grounded) techniques share the same basic ideas: (1) Restrict the depth of the allowed decompositions to some limit K, (2) construct a compact representation of possible decompositions up to depth K, (3) encode this along with state executability into a SAT formula, (4) use a SAT solver to determine whether a plan for the depth bound K exists, and (5) if not, increase K and try again. The core element of this technique is to compute a representation of all decompositions up to the given depth limit K. For TOHTN, the derivation of a plan via decompositions takes the form of a parse tree for a context-free grammar. As such, the propositional formula has to represent all possible parse trees that have a depth of at most K. For this, two isomorphic representations have been proposed: A common super tree of all possible parse trees (Path Decomposition Tree, PDT) (Behnke, Höller, and Biundo 2018a) and a tape-based layer structure (Schreiber et al. 2019).

Both structures share a problem: They do not consider the state-transition semantics of actions at construction time. Instead, state transitions are added afterwards as part of the SAT formula. They notably also represent decomposition trees whose plans are not executable. These trees might be pruned in order to decrease the size of the representation and therefore the formula. Presumably, such a reduction will help the SAT solver. We present a pruning method based on state invariants. One of the major differences between the two encodings is the handling of method preconditions. PANDA (Behnke, Höller, and Biundo 2018a) compiles method preconditions into additional actions – which must be represented in the encoding and increase the overall length of the plan. Tree-Rex (Schreiber et al. 2019) handles method preconditions separately by checking the method precondition in the state in which we apply the first derived action – which leads to a smaller SAT formula. We generalise this idea to the concept of *block compression*.

action	a	b	c	d	e	f	g
pre	$\emptyset$	$\{y\}$	$\emptyset$	$\emptyset$	$\{y,z\}$	$\{z\}$	$\emptyset$
add	$\{x\}$	$\emptyset$	$\{y\}$	$\{z\}$	$\emptyset$	$\emptyset$	$\emptyset$
del	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Table 1: Actions of the example TOHTN problem.

I	$\mapsto$	ABC	B	$\mapsto$	c	D	$\mapsto$	eg
I	$\mapsto$	BD	B	$\mapsto$	d	D	$\mapsto$	f
A	$\mapsto$	ab	C	$\mapsto$	g			

Table 2: Methods of the example TOHTN problem.

TOHTN Planning

Totally-Ordered Hierarchical Task Network Planning (TOHTN (Erol, Hendler, and Nau 1996)) shares structural similarities with context-free grammars (Erol, Hendler, and Nau 1996; Höller et al. 2014). We distinguish two types of tasks: Primitive actions and abstract task. In grammar terms, primitive actions are terminals and abstract tasks are non-terminals. We denote the set of primitive actions (or actions for short) with $\mathcal{P}$, while we denote the set of abstract tasks (or tasks) with $\mathcal{A}$. We call every element of $\mathcal{T} = (\mathcal{P} \cup \mathcal{A})^*$ a task sequence. A *decomposition method* (or method) is a rule of the form $A \mapsto \omega$ where $A \in \mathcal{A}$ and $\omega \in \mathcal{T}$. Given a task sequence $\lambda = \mu A \nu \in \mathcal{T}$ with $\mu, \nu \in \mathcal{T}$, we can apply the method $A \mapsto \omega$ to A in λ, leading to the task sequence $\lambda' = \mu\omega\nu$. If a task sequence λ' can be derived via a single application of a decomposition method to λ, we write $\lambda \rightarrow \lambda'$. We denote with $\rightarrow^*$ the transitive and reflexive closure of $\rightarrow$, i.e. $\lambda \rightarrow^* \lambda'$ holds if zero or more methods can be applied to λ to obtain λ'.

The objective of an TOHTN planning problem is given in terms of an initial abstract task I (in grammar terms: the initial non-terminal). A primitive task sequence $\omega \in \mathcal{P}^*$ is a derivable primitive task sequence if $I \rightarrow^* \omega$ holds. As in classical planning, actions have a state transition semantics. It is defined over a set of propositional state variables V. For every action, we are given three sets, $pre(a)$, $add(a)$, and $del(a)$. An action a is applicable in a state $s \subseteq V$ iff $pre(a) \subseteq s$. The state resulting from the application of a in s is $\gamma(a, s) = (s \setminus del(a)) \cup add(a)$. A primitive task sequence $\omega = a_1 a_2 \ldots a_n$ is called executable in s_0 iff states $s_1, \ldots s_n$ exist such that for all $i \in \{1, \ldots n\}$ a_i is applicable in s_{i-1} and $\gamma(a, s_{i-1}) = s_i$. A derivable primitive task sequence ω is a solution to an HTN planning problem, i.e., a plan, iff ω is executable in the problems initial state s_0.

A TOHTN planning problem $P = (\mathcal{P}, \mathcal{A}, M, V, I, s_0)$ consists of $\mathcal{P}$ (the primitive actions), $\mathcal{A}$ (the abstract tasks), M (the decomposition methods), V (the state variables), I (the initial abstract task), and s_0 (the initial state). A plan ω for an TOHTN planing problem P must (1) be derivable via decomposition from the initial task I, i.e., $I \rightarrow^* \omega$, and (2) must be executable in s_0. For notational purposes we define the set $M(a) = \{a \mapsto \omega \mid a \mapsto \omega \in M\}$ – the set of methods applicable to the abstract task a.

In order to illustrate the techniques in this paper, we will use a toy TOHTN planning problem. It has the abstract tasks I, A, B, C, and D. The initial abstract task is I. Furthermore, there are the primitive actions a, b, c, d, e, f, and g whose preconditions and effects are listed in Tab. 1. The domain has eight decomposition methods listed in Tab. 2

This TOHTN problem has six derivable primitive task sequences: $abcg$, $abdg$, ceg, deg, cf, and df. Of these, the first two are not executable as the precondition of b will not be satisfied. The next two are also not executable, as e requires that the two propositional variables y and z are both true, which can only be achieved by executing c and d. Lastly, cf is not executable, as it f requires y to be true, which is the case in the only plan df.

Given a plan ω with $I \rightarrow^* \omega$, we can interpret its derivation from I as a parse-tree for a context-free grammar. Such a tree is called a *decomposition tree* for ω (Geier and Bercher 2011). In our example domain, each of the six derivable primitive task sequences has a unique parse tree, i.e. a unique decomposition tree. In Fig. 1a we show the decomposition trees for the plans $abcg$ and df.

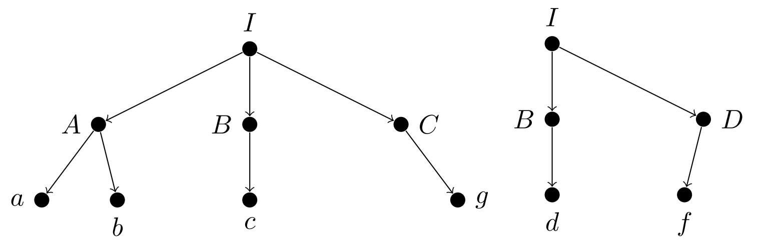
Definition 1. *A Decomposition Tree (DT) for a plan ω for a TOHTN planning problem P is a tree $\mathbb{T} = (N, E, \beta)$ with*

- *N – a set of nodes,*
- *$E : N \mapsto N^*$ – the edge function providing for every node an ordered list of children $\langle e_1, \ldots, e_k \rangle$, and*
- *$\beta : N \mapsto \mathcal{T}$ – the node labelling function which assigns a task to every node. For any inner node $\beta(n) \in \mathcal{A}$. For any leaf node either $\beta(n) \in \mathcal{P}$ or $\beta(n) \mapsto \varepsilon \in M$.*
- *For every inner node n, a method $\beta(n) \mapsto t_1, \ldots, t_k \in M$ must exists, such that $|E(n)| = k$ and $t_i = \beta(e_i)$ for all $e_i \in E(n) = \langle e_1, \ldots, e_k \rangle$.*
- *For the sequence of leafs $L = \langle n_1^\ell, \ldots, n_l^\ell \rangle$ with $\beta(n_i^\ell) \in \mathcal{P}$ it holds that $\omega = \beta(n_1^\ell) \cdots \beta(n_l^\ell)$.*

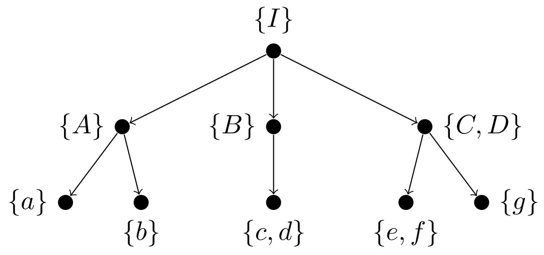
SAT-based TOHTN Planning

The core idea of SAT-based TOHTN planning is to create a formula $\mathcal{F}(P)$ that is satisfiable if and only if P has a solution – and that this solution can be extracted from the satisfying valuation of $\mathcal{F}(P)$. To avoid creating too large a formula, the typical approach is to create a sequence $\mathcal{F}_i(P)$ of formulae of increasing size and solve them one-by-one until a solution is found. In all currently existing encodings (Schreiber 2021b; Schreiber et al. 2019; Behnke, Höller, and Biundo 2018a), the parameter i bounds the maximum depth of decomposition. As such, the formula considers only plans that have a decomposition tree with a maximum depth of i. The formula $\mathcal{F}_i(P)$ is satisfiable if and only if a plan ω exists that has a decomposition tree with a depth of at most i. One could also consider formulae that can capture only some decomposition trees of a given depth, e.g. by allowing higher depths only for certain parts of the problem, while restricting other parts to a lower depth. Such uneven encodings have not yet been investigated and are thus an objective for future research. Further, they require extra care to ensure completeness, or optimality when used for current optimal SAT-based HTN planners (Behnke, Höller, and Biundo 2019b).

The formula $\mathcal{F}_i(P)$ has to represent all possible decomposition trees of depth at most i. To capture the set of all



(a) Two decomposition trees for the task networks $abcg$ and df.



(b) A PDT.

Figure 1: Decomposition Trees and Path Decomposition Trees (PDTs).

possible decomposition trees up to the depth i, the existing encodings use two – essentially isomorphic – data structures. We present only one of them, as our pruning techniques are based on it.

Definition 2. *A Path Decomposition Tree (PDT) for a TO-HTN planning problem P is a tree $\mathfrak{T} = (N, E, \alpha)$ with*

- *N – a set of nodes,*
- *$E : N \mapsto N^*$ – the edge function providing for every node an ordered list of children, and*
- *$\alpha : N \mapsto 2^{\mathcal{T}}$ – the node labelling function which assigns to every node a set of tasks.*

We denote the root node of the PDT with $r_{\mathfrak{T}}$. We call the PDT well-formed for depth i iff:

1. *$\alpha(r_{\mathfrak{T}}) = \{I\}$*
2. *For all $n \in N$ which are not at depth i or $\alpha(n) \subseteq \mathcal{P}$ and its children $E(n) = \langle c_1, \ldots, c_n \rangle$*
   - *(a) For every label $l \in \mathcal{A} \cap \alpha(n)$ and every method $l \mapsto t_1 \ldots t_m$, there are children $c_{p_1}, \ldots, c_{p_m}$ with $p_1 < \cdots < p_m$ such that $\forall j \in \{1, \ldots, m\} : t_j \in \alpha(c_{i_j})$.*
   - *(b) $\forall l \in \mathcal{P} \cap \alpha(n) \exists c_i \in E(n)$ with $l \in \alpha(c_i)$.*
3. *The PDT's label function contains no other tasks than those required by the previous two requirements.*

A well-formed PDT $\mathfrak{T}$ captures all possible decompositions tree up to a given depth (Behnke, Höller, and Biundo 2018a). Every decomposition tree $\mathbb{T} = (N_T, E_T, \beta)$ is (up to renaming of the nodes) a rooted sub-tree of $\mathfrak{T}$, i.e. $N_T \subseteq N$ and $E_T \subseteq E$. Further the labels of the decomposition tree $\mathbb{T}$ are reflected by the PDT $\mathfrak{T}$, i.e. $\beta(n) \in \alpha(n)$ for all $n \in N_T$. This can also be understood as a matching of the nodes N_T to nodes N, i.e. a homomorphism h from N_T to N. Consider an arbitrary decomposition tree $\mathbb{T}$. The PDT's root node $r_{\mathcal{T}}$ is always matched with the $\mathbb{T}$'s root node (i.e. $h(r_{\mathcal{T}}) = r_{\mathfrak{T}}$). Consider any arbitrary node $n' \in N_T$ that is matched to a node n in the PDT. Either the node n' is labelled with an action – then we have nothing to show – or it is labelled with an abstract task l. If so, in $\mathbb{T}$, some method $l \mapsto \omega$ is applied to it. Since n' can be matched to n, $l = \beta(n') \in \alpha(n)$. Thus there are children of n that have the tasks of ω in their label set in the correct order – which can be matched to the children $E_T(n')$ of n' in $\mathbb{T}$. Note that the leafs of the PDT $\mathfrak{T}$ form an ordered sequence where the order is imposed by the edge function E. There is one important detail in the definition of PDTs that adds a lot to its practicality: If there is a primitive action p in the label set of a node n, then this represents a leaf of a possible decomposition tree. This primitive action p is added to the label set of one of the children – as long as there are any. This pushes (or "copies") possible leafs of decomposition trees down towards the leafs of the PDT. As such, every derivable primitive task sequence is a subsequence[1] of the *leafs* of the PDT (and a corresponding selection of labels). Else we would have to consider inner nodes as potential parts of these task sequences. To illustrate the concept of a PDT, consider again the toy problem introduced in the previous section. A PDT of depth 2 for this problem is depicted in Fig. 1b. Note that it contains the two parse-trees of depth 2 shown in Fig. 1a as subtrees. Note that when constructing the PDT, there are still choice points. E.g., the root node of the PDT in Fig. 1b could have a fourth child labelled with $\{C\}$ (and this node one child labelled {g}) while we remove C from the label set of the third child. Both of these PDTs still capture all decomposition trees of depth 2 – they just differ in how. We assume that a PDT has already been build using the greedy method of Behnke, Höller, and Biundo (2018a).

We will not modify the actual SAT encoding of the PDT and as such will not present it. For details, we refer to Behnke, Höller, and Biundo (2018a) and Schreiber et al. (2019). The general idea of these encodings is that we encode the selection of a decomposition tree $\mathbb{T}$ as the selection of a rooted subtree of the PDT $\mathfrak{T}$. This selected decomposition tree $\mathbb{T}$ directly induces a primitive task sequence ω which is the plan we are looking for. To encode the selection of a decomposition tree $\mathbb{T}$, we have to select, for every node n of $\mathfrak{T}$ that should be a part of $\mathbb{T}$, the task it is labelled with in $\mathbb{T}$, i.e. $\beta(n)$. We do this via decision variables l^n for $n \in N$ and $l \in \alpha(n)$. A node $n \in N$ of the PDT is selected for the decomposition tree $\mathbb{T}$ if any of the l^n is true. We represent the method applied to the selected task l^n with decision variables m^n for $n \in N$ and $m \in \bigcup_{a \in \beta(n) \cap \mathcal{A}} M(a)$. The SAT formula we construct asserts that the selection represented by the decision variables forms a valid decomposition tree. For example, we enforce that for selected abstract tasks a

[1]In this paper, we refer with a *subsequence* of a sequence to any selection of elements of the sequence. If the selected elements must be contiguous, we call this a contiguous subsequence.

method is chosen ($l^n \to \vee_{m=l\mapsto\omega} m^n$), methods are only applied to the correct task ($m^n \to l^n$ for $m = l \mapsto \omega$), methods enforce their subtasks to be present ($m^n \to t_i^{n_i}$ for $m = l \mapsto t_1, \dots, t_k$ and appropriate children n_i), primitive actions are inherited towards the leafs, at most one method is selected, and each node is labelled with at most one task. The task variables l^n of the leafs of the PDT are used as the action variables in a SAT encoding for classical planning – this is only possible with propagating primitive actions towards the leafs. SAT encodings for classical planning divide the plan into a sequence of discrete time steps in which actions are applied. If we consider the ordered leafs $\langle \ell_1, \dots, \ell_n \rangle$ of the PDT $\mathfrak{T}$, each leaf ℓ_i constitutes the time step i in the classical encoding in which only the actions in $\alpha(\ell_i)$ can be applied. Currently, both planners use the Kautz-Selman encoding (Kautz and Selman 1996).

Pruning the PDT

Until now, a PDT $\mathcal{T}$ for a given TOHTN P and a given depth i must contain *all possible* decomposition trees. This is a significant drawback. This way, we also represent decomposition trees that do not lead to (1) primitive task sequences as well as to (2) non executable task sequences. It would be advantageous to construct a reduced PDT that does not contain these decomposition trees. Constructing a PDT that *only* contains the decomposition trees leading to executable primitive task sequences is not feasible as it would require enumerating all solutions of the planning problem we want to solve in the first place. As such, we are interested in an approximation. This approximation takes the form of removing labels (i.e. actions and tasks) from the label sets $\alpha(n)$ – which implicitly removes decomposition trees that label n with these removed labels. Normally, the PDT and its encoding into SAT also consider *all* methods applicable to the abstract tasks in $\alpha(n)$ for a node n (see Def. 2). In some cases, we cannot remove a label from $\alpha(n)$, but know that some method cannot be applied to it, because applying it will lead to an non-executable task sequence. To capture this case, we introduce the set of *impossible* methods $\mathcal{I}(n)$. Similar to the tasks removed from $\alpha(n)$, we add a method $m = A \mapsto \omega$ to $\mathcal{I}(n)$ if we can prove that applying m to the task A at the node n of any concretely selected decomposition tree will lead to a non-executable task sequence. Whether a method m is impossible – i.e. it can only lead to non-executable tasks sequences – depends on where in the PDT we want to apply it. It may be possible to derive an executable task sequence after applying m for a node n, but not for a node n'. Any method in $\mathcal{I}(n)$ will be ignored while encoding the PDT.

The basis of our pruning technique will be identifying labels that can safely be removed from a leaf of a PDT. We call this *leaf pruning*. Removing such a label may also imply that methods are impossible and that other labels can be removed from other nodes of the PDT. Thus, after pruning labels from leafs, we propagate this information through the PDT. If propagation prunes more labels from leafs, leaf pruning may be able to prune even more labels. We repeat leaf pruning and propagation until no further labels can be pruned.

Propagating Impossibility

Before discussing leaf pruning, we show how the information contained in pruned leaf labels can be propagated. Propagation always starts when a label l is removed from the label set $\alpha(n^\ell)$ for some leaf node n^ℓ (or a set of labels from some leafs, we treat all labels pruned by the leaf pruning in one step for efficiency). Since we know that the leaf n^ℓ cannot be labelled with l, it is also impossible to use the method(s) that caused l to be inserted into $\alpha(n^\ell)$ while constructing the PDT $\mathcal{T}$. If we were to use one of these methods on the parent of n^ℓ, we would assign l to n^ℓ. We already know that this assignment is impossible, as we pruned it.

To formalise this, we introduce the set $\mathcal{M}(n, l)$. It contains all methods that caused us to add a specific label l to $\alpha(n)$ for any node n of $\mathfrak{T}$. We can track this information while constructing the PDT (independent of the construction technique): By remembering the reason for adding the labels to $\alpha(n)$. More precisely, these are the methods applied to the parent of n that force adding l to $\alpha(n)$ due to condition 2a) of Def. 2. Consider the PDT in Fig. 1b as an example. When expanding the node n labelled with $\{A\}$, we apply the method $A \mapsto ab$. We then decide to use the left-most child n^ℓ of n to represent the subtask a and add a to its label set. Thus $\mathcal{M}(n^\ell, a) = \{A \mapsto ab\}$. We further denote with $\hat{n}$ the parent of n in the PDT $\mathfrak{T}$. If we removed the label l from $\alpha(n)$, we can mark all methods in $\mathcal{M}(n, l)$ as impossible for the parent $\hat{n}$. We thus add these methods to $\mathcal{I}(\hat{n})$.

Next, it is possible that there is a node n with an abstract task $a \in \alpha(n)$ in its label set, for which all applicable methods have been marked as impossible. Formally, this is the case if $M(a) \subseteq \mathcal{I}(n)$. If a decomposition tree would contain the node n labelled with a, no method could be chosen to apply be applied to a, since none can lead to an executable primitive task sequence. This is precisely why the methods have been marked as impossible. Thus, n cannot be labelled with a and we can remove a from $\alpha(n)$. Here, we can repeat the process: We have just removed a label from the label set, which is the fact that causes the propagation.

In addition to the just described upwards pruning, we can also propagate the pruning down the tree. We can mark a method as impossible for a node n (and add it to $\mathcal{I}(n)$) if one of its subtasks is impossible for one of n's children. This also implies that we do not need to add the other subtasks of this method to the label sets of the other children of n – except if there is another method that adds them. Formally, we remove the label l from a node n if $\mathcal{M}(n, l) \subseteq \mathcal{I}(\hat{n})$, i.e. if all methods that can assign l to n have been pruned.

Both pruning techniques are applied until we can remove no further labels or methods. The propagation of pruning information is sound by construction – provided that the initially pruning of labels was correct.

Leaf Pruning

We have discussed how we can propagate pruning information through the PDT, but not how we can obtain the initial *leaf pruning*. The simplest type of decomposition trees that can be removed are those that cannot even lead to a *primitive* task sequence. Consider a leaf n^ℓ of a PDT $\mathfrak{T}$. If there is an

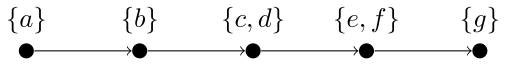


Figure 2: The leafs of the PDT in Fig. 1b.

abstract task $a \in A$ in $\alpha(n^\ell)$, this task cannot be selected as part of a decomposition tree – except if there is a method $a \mapsto \varepsilon \in M$ and n^ℓ has not depth i. In any other case, the task a occurs in a leaf whose depth is the depth limit i. To obtain a primitive task sequence we must apply at least one more method – exceeding the depth limit. As a consequence, we can remove such $a \in A$ from the label sets of leafs.

Pruning only abstract tasks from leafs does not take the state transition semantics of primitive actions into account – and thus wastes a lot of information. To consider the preconditions and effects of actions, we have to look at the meaning of the leafs of a PDT. The leafs form a sequence $L = \langle n^\ell_1, n^\ell_2, \dots, n^\ell_m \rangle$. Any derivable primitive task sequence $t_1 t_2 \dots t_k$ is represented by a subsequence of these leafs $\langle n^\ell_{i_1}, \dots, n^\ell_{i_k} \rangle$ s.t. $i_1 < \dots < i_k$ and $\forall j \in \{1, \dots, k\}$: $t_j \in \alpha(n^\ell_{i_j})$. A derivable primitive task sequence can be viewed as selecting a subsequence of the leafs and selecting *one* or zero action in every leaf's label set. We depict this structure for our example TOHTN domain in Fig. 2.

If we execute a derivable primitive task sequence, we can execute one or no action from each $\alpha(n^\ell_j)$, which restricts the possibly executable plans. E.g. we know that any action in n^ℓ_1 will be executed in the initial state s_0 and any action that is not applicable in it can be pruned. Furthermore, we know that any action in n^ℓ_2 will either be applied in s_0 (if no action from n^ℓ_1 is executed) or in a state resulting from applying *one* of the unpruned actions in $\alpha(n^\ell_1)$. Any action that is not applicable in any such state can be pruned. We use this type of reasoning for leaf pruning.

Considering the interdependency of the selection of actions for leafs is unfortunately computationally expensive ($\mathbb{NP}$-complete). As a relaxation, we will thus not consider such interdependencies. For example, we know that if we select a for the first leaf in Fig. 2, we must also select b for the second – as the method that assigns a to the first leaf also assign b to the second. Thus b cannot be executed in s_0 but only in $\gamma(a, s_0)$. We ignore these interdependencies. We mitigate this somewhat by propagating pruning information as described in the previous section.

The formal question that we have to answer in order to decide whether to prune an action l from the label set of a leaf $\alpha(n^\ell)$ is the following: Is it possible to select *any* combination of actions from the label sets of leafs before n^ℓ such that executing them in s_0 leads to a state in which l is executable, i.e. $prec(l)$ is satisfied. If not, we can safely prune l. Unfortunately, determining this exact pruning is NP-hard. In this theorem $prec(l)$ is represented by a goal formula s_g.

Theorem 1. *Given a sequence of sets of actions $\langle A_1, \dots A_n \rangle$ and an initial state s_0 and a goal state s_g. The problem* LEAFPRUNE *is to determine whether a subsequence $A_{i_1}, \dots, A_{i_m}$ and a selection of actions $\forall j : a_j \in A_{i_j}$ such that $\langle a_i, \dots, a_j \rangle$ is executable in s_0 and s_g is satisfied in the resulting state.* LEAFPRUNE *is* NP*-complete.*

Proof. Membership can be proven via a guess-and-check algorithm. Hardness follows from a reduction from SAT (Cook 1971). Let $\phi = \bigwedge_{i=1}^m \bigvee_j l_{i,j}$ be a SAT formula over variables $v_1, \dots v_n$. We define $2n$ actions a^+_i and a^-_i with no preconditions and delete effects. a^+_i has the add effect v^+_i and v^+_i. We assign $A_i = \{a^+_i, a^-_i\}$, which select the truth of the variables. For every clause $\bigvee_{j=1}^k l_{x,j}$, we create k actions $\ell_{x,j}$. If $l_{x,j}$ is the positive literal v_i, $\ell_{x,k}$ has the precondition a^+_i, else if $l_{x,j} = \neg v_i$, $\ell_{x,j}$ has the precondition a^-_i. Every $\ell_{x,j}$ has the add effect c_x. We set $A_{x+n} = \{\ell_{x,1}, \dots, \ell_{x,k}\}$. The goal is $\{c_1, \dots, c_m\}$. The goal is reachable by an executable selection of the actions of the A_i if and only if ϕ is satisfiable. □

Since we cannot expect to perform the pruning exactly in a reasonable time, we again have to approximate. Instead of considering the actually reachable states after the leaf n^ℓ, we consider properties that hold in *all* reachable states. We do this with disjunctive state invariants. A disjunctive state invariant is a disjunction of literals over the state variables $\bigvee_i l_i$. In this paper we restrict ourselves to two cases: (1) the size of the clauses is one, i.e. they are sole literals and (2) the size of the clauses is two, i.e. they are of the form $(l_1 \vee l_2)$. For the ith leaf, we will construct a set of invariants I_i that hold in every possible state that can be reached before it.

Definition 3. *Given a sequence of sets of actions $\langle A_1, \dots A_n \rangle$ and an initial state s_0 and an invariant ϕ. ϕ is valid after A_n iff for any subsequence $A_{i_1}, \dots, A_{i_m}$ and any selection of actions $\forall j : a_j \in A_{i_j}$ such that $\langle a_i, \dots, a_j \rangle$ is executable in s_0, $\gamma(\langle a_i, \dots, a_j \rangle, s_0) \models \phi$*

The set of invariants decreases monotonically with respect to the leafs: We can always choose not to execute an action and thus no invariant that was not in I_i can be in I_{i+1}. Computing I_1 is trivial: These are the invariants which hold in the initial state s_0. Let us now consider the general case. Assume that we have computed the set of invariants I_i. If we execute any action $a \in \alpha(n^\ell_i)$ of the ith leaf, it will be executed in a state satisfying I_i. If a's preconditions are incompatible with I_i, i.e. $prec(a) \not\models I_i$, then it is impossible to execute a at n^ℓ_i. Thus a can be pruned from $\alpha(n^\ell_i)$. Next, we have to determine the invariants I_{i+1} – those that hold after executing none or one of the actions in $\alpha(n^\ell_i)$. We have to distinguish between the two types of invariants we consider.

Unary Invariants Positive unary invariants, i.e. single state variables v, are irrelevant for pruning since we don't consider negative preconditions. Only a negative unary invariants, i.e. $\neg v$, can restrict the executability of actions. Such an invariant represents the fact that a state variable v cannot be made true. Checking whether an action's preconditions $prec(a)$ are compatible with a set of negative unary invariants is simple: Just check whether $\exists v \in prec(a)$: $\neg v \in I_i$. If we can execute an action a, its add effects $add(a)$ hold after executing it. We remove every $\neg v$ from I_{i+1} for which $v \in add(a)$ for some executable action a in $\alpha(n^\ell_i)$. Note that actions of one leaf cannot enable each other. If we are in a state in which $\neg x$ is an invariant and have two actions a and b in $\alpha(n_i^\ell)$ with $add(a) = prec(b) = x$, then a is executable but b is not – even though a enables b, since we can execute only one of the actions at this time we cannot use a to enable b. On an abstract level, unary invariant pruning is equivalent to the classical delete-relaxed reachability analysis when restricted to the actions in a PDT's leafs.

Binary Invariants While unary invariants only describe the reachability of facts, binary invariants can express more complex dependencies such as n-ary mutexes between state variables, i.e. SAS+ variables (Bäckström and Nebel 1995). Rintanen (1998) has presented a framework for handling such binary invariants which perfectly fits our purpose. He defines a function $F_a(I)$ which filters out binary invariants which can be violated by executing an actions a. One provides a set of binary, disjunctive invariants I which hold prior to executing the action a. $F_a(I)$ returns all invariants that still hold after executing the action a in any state that is compatible with I. Since we consider only positive preconditions, only an invariant $\neg x \vee \neg y$ s.t. $\{x, y\} \subseteq prec(a)$ can cause an action a to be inapplicable. Such actions are pruned from $\alpha(n_i^\ell)$. Based on the remaining actions we set $I_{i+1} = \bigcap_{a \in \alpha(n_i^\ell)} F_a(I)$, i.e. all invariants that hold after executing any of the actions of n_i^ℓ.

State Invariants Based on the two pruning techniques, we obtain for the state before each of the leafs of the PDT a set of invariants that will hold in that state. Provided we will actually encode this state (see Section on Block Compression), we can add these invariants as clauses to the state. As a general improvement to the encoding, we also compute the general invariants derivable using the method presented by Rintanen (1998). Since these invariants are true in all reachable states, we add them to every state. Adding invariants is generally advantageous – and usually provide a significant improvement in runtime – as it alleviates the SAT solving from having to re-infer them itself for every time step. Further SAT solver can use the invariants to cut branches of its search tree that violate invariants early.

Example To illustrate our pruning techniques, we use our example TOHTN in Fig. 1b. Further assume that the actions $a - g$ have the preconditions and effects shown in Tab. 1 and assume that $s_0 = \emptyset$. We start with unary invariant pruning. The (relevant) invariants of s_0 are $I_1 = \{\neg x, \neg y, \neg z\}$. We then process the first leaf labelled with $\{a\}$. Since a has no preconditions, it is executable and we thus remove the invariant $\neg x$ and obtain $I_2 = \{\neg y, \neg z\}$. We then proceed to the second leaf labelled with $\{b\}$. b has the precondition y but we know that the invariant $\neg y$ holds. It is therefore pruned, its effects are not applied, and thus $I_3 = I_2$. Next we process the third leaf labelled with $\{c, d\}$. Both have no preconditions and are not pruned. Their effects cause both $\neg y$ and $\neg z$ to be removed from the invariant set and thus $I_4 = I_5 = \emptyset$. No further actions can be pruned at later leafs. Using propagation, we can e.g. further deduce that a has to be pruned from the first leaf since the only method that can cause it is pruned as b is pruned from the second leaf.

If we use binary invariant pruning, we would derive that the invariant $\neg y \vee \neg z$ holds after the third leaf – since $\neg y$ and $\neg z$ hold in I_3 and neither c nor d can make both y and z true. This allows us to prune e from the fourth leafs label set. Using propagation, this leaves only the sole solution of the planning problem to be represented in the pruned PDT.

Block Compression

Both aforementioned grounded SAT encodings (Schreiber et al. 2019; Behnke, Höller, and Biundo 2018a) encode the actions assigned to the leafs of the PDT in a time-step-by-time-step fashion. The executability of the final plan ω is checked by executing its actions, i.e. those actions assigned to the leafs of a selected decomposition tree, over a sequence of time steps, where for each time step t the formula has variables expressing that the propositional state variable $v \in V$ is true at time step t. In the current encodings, per time step only one leaf is considered for execution and thus at most one action can be executed per time step. This requires that the state prior to every leaf n^ℓ is explicitly represented. This is done via a set of decision variables v^{n^ℓ} for all state variables $v \in V$. For many of these states, it is however not necessary to encode them as we will show in this section. Instead multiple leafs can share the same state that is represented with one set of decision variables.

Schreiber et al. (2019) already identified a case in which separate states are not necessary. Many modelled TOHTN domains do not conform to the simplistic formalism we introduced. They allow for specifying so-called *method preconditions*. A method precondition is a set of state variables $prec(m) \subseteq V$ associated with a method. The method precondition has to hold in the state in which the first primitive action resulting from the subtasks of the method is executed. Some planners (e.g. PANDA (Höller et al. 2021)) use a compilation-based representation of method preconditions. For a method $m := A \mapsto \omega$ with a method precondition $prec(m)$, they create a new primitive action m' with the precondition $prec(m)$ and no effects. The method m is then changed to $A \mapsto m'\omega$. Using the SAT-encoding technique of Behnke, Höller, and Biundo (2018a), this will lead to a lot of leafs that are labelled solely with method precondition actions. For each such leaf n^ℓ, decision variables v^{n^ℓ} representing the state before the leaf are introduced. The state before the next leaf $n^{\ell'}$ will be wholly identical to the state before n^ℓ – since no effects can be applied. In the SAT formula, this is asserted using the clauses $v^{n^\ell} \to v^{n^{\ell'}}$ and $\neg v^{n^\ell} \to \neg v^{n^{\ell'}}$ for all $v \in V$. The variables $n^{\ell'}$ and the additional clauses are useless.

Schreiber et al. (2019) handled method preconditions separately and do not compile them into actions. For every node n of the PDT to which a method can be applied, we can determine the first leaf n_1^ℓ that is below n. I.e. the node below n that occurs first in the total order of the leafs. The method precondition of any method m applied to n can be checked in the state before n_1^ℓ. $prec(m)$ is then asserted in this state, which reduces the complexity of the generated SAT formula.

The technique by Schreiber et al. (2019) is handling only the special case of method preconditions. We generalise their idea to the concept of *Block Compression*. For Block

Compression, we compile method preconditions into additional actions – to be able to handle actions and method preconditions uniformly. For a method m applied to an inner node n, we should execute the method precondition action m' in parallel with the action assigned to the first non-method precondition leaf n^ℓ_1 below n. The reason for this is that the leaf of the method precondition n^ℓ and n^ℓ_1 are immediate successors and there is no "interference" between the actions assigned to them – the method precondition action as no effects and thus cannot change the state. In effect, the method precondition and the action assigned to n^ℓ_1 will always be executed in the exactly same state. Block Compression groups the leafs into (contiguous) blocks. A leaf n^ℓ is allowed to belong to a block if all leafs prior to n^ℓ in that block cannot be labelled with an action that will "interfere" with the execution of any possible action n^ℓ. If we consider a contiguous subsequence of leafs $B = \langle n^\ell_1, \dots, n^\ell_n \rangle$ and their label sets $\langle \alpha(n^\ell_1), \dots, \alpha(n^\ell_n) \rangle$, we would like to omit the computation and explicit representation of the states between the execution of the leafs. Instead, we would like to consider only two states: the one before the first leaf of the block s^B_0 and the one after the last leaf s^B_1. We are then looking for a criterion s.t. the executability of an action $a \in \alpha(n^\ell_i)$ can be determine solely by checking whether its preconditions are met in s^B_0 – irrespective of the actions chosen for the other leafs. We formalise this strong notion of compatibility as follows. Note that it is possible to select no action for a particular leaf, i.e. leave the leaf empty.

Definition 4. *We call a sequence of sets of actions $\langle A_1, \dots, A_n \rangle$ block compatible, if for every state s^B_0 and selection of actions from the sequence $\omega = a_{i_1}, \dots, a_{i_m}$ with $a_j \in A_j$ and $1 \le a_{i_1} < \cdots < a_{i_m} \le n$, ω is executable in s^B_0 if and only if all a_{i_j} are executable in s_0.*

Checking block compatibility directly with Def. 4 is computationally expensive, since we have to enumerate all states s^B_0 and action selections. We instead check the simpler syntactic criterion of non-dependence, which we show to be equivalent to block compatibility.

Definition 5. *Given a sequence of sets of actions $B = \langle A_1, \dots, A_n \rangle$. A set of actions A_{n+1} is* non-dependent *on B iff $\forall a \in A_{n+1} \forall j \in \{1, \dots, n\} \forall a' \in A_j : prec(a) \cap (add(a') \cup del(a')) = \emptyset$.*

Theorem 2. *A sequence of sets of actions $A = \langle A_1, \dots, A_n \rangle$ is block compatible, if and only if $\forall i \in \{2, \dots, n\}$ A_i is non-depended on $\langle A_1, \dots A_{i-1} \rangle$.*

Proof. If $\forall i$ A_i is non-depended on $\langle A_1, \dots A_{i-1} \rangle$, then A is clearly block compatible. Assume that A is block compatible, but $\exists i$ s.t. A_i is non-depended on $\langle A_1, \dots A_{i-1} \rangle$. Then there would be actions $a_i \in A_i$ and $a_j \in A_j$ with $j < i$ and either $del(a_j) \cap prec(a_i) \neq \emptyset$ or $add(a_j) \cap prec(a_i) \neq \emptyset$.

If $del(a_j) \cap prec(a_i) \neq \emptyset$, consider $s_0 = V$. $\langle a_j, a_i \rangle$ is not executable, but a_j and a_i are executable in s_0 – since we have only positive preconditions. Thus, A would not be block compatible. We can further assume that $add(a_j) \cap prec(a_i) = \emptyset$, as any state variable added that is already a precondition is useless. We know that $del(a_j) \cap prec(a_i) = \emptyset$, since $j < i$. We can select $s_0 = V \setminus (add(a_j) \cap prec(a_i))$. $\langle a_j, a_i \rangle$ is executable in s_0, but a_i is not executable in s_0. Thus A would not be block compatible. $\square$

The notion of block compatibility is similar to the $\forall$-step semantics in classical SAT-based planning (Kautz and Selman 1996; Rintanen, Heljanko, and Niemelä 2006). The latter allows for a set of actions to be executed in parallel if they can be executed in any arbitrary order. Block compatibility on the other hand already has a fixed given order of actions – even though there is still variability as we can select different actions (or on at all) for each leaf. This way, we can a stronger compression than the $\forall$-step semantics. Further, the $\forall$-steps semantics only *allows* for actions to be executed in parallel, i.e. they still can be executed sequentially, we *force* the actions to be executed in parallel as their presence is required by the HTN structure, i.e. we cannot postpone an action to a next time step.

What remains to encode is computation the state s^B_1 resulting from the execution of all actions assigned to the leafs in B. This state can be determined by considering – per state variable $v \in V$ – only the last action that affects v, or if none exists the truth of v at the beginning of B. Determining this precisely in a SAT formula is complicated as the "last action that affects v" depends on the actual selection of actions. We instead opt for a simplistic approach: We perform block compression if it is not possible to select two actions a_i and a_j from two leafs n^ℓ_i and n^ℓ_j with conflicting effects. This allows us to assert the effects of all actions of the leafs in B directly to hold in the state s_1.

Definition 6. *Given a sequence of sets of actions $B = \langle A_1, \dots, A_n \rangle$. A set of actions A_{n+1} is* non-interfering *with B iff $\forall a \in A_{n+1} \forall j \in \{1, \dots, n\} \forall a' \in A_j$: $add(a) \cap del(a') = del(a) \cap add(a') = \emptyset$.*

We perform block compression as follows. We consider the leafs $n^\ell_1, \dots, n^\ell_n$ of the PDT in order and maintain a current block B starting with $B = \langle n^\ell_1 \rangle$. For each n^ℓ_i, we add n^ℓ_i to B if $\alpha(n^\ell_i)$ is non-dependent on and non-interfering with the labels of B. Else we finish the block B and start a new block $B = \langle n^\ell_i \rangle$. It might be possible to find a partitioning of the leafs into fewer block than with this greedy method. We use this greedy method as it can be computed efficiently.

For each computed block of leafs, we can omit the representation of the intermediate (or inner) states – only the state prior to the *whole* block and the one after the *whole* block must be encoded. For encoding any action inside of the block it suffices to refer to these two states for checking its preconditions and asserting its effects. Block Compression poses stronger requirements on the actions than the $\forall$-step semantics. As a result, we don't need to add further clauses to the formula (any selection of actions can be executed in parallel) – while $\forall$-steps does (it must assert that its conditions are met).

The effectiveness of Block Compression – measured in terms of number of blocks – is dependent on the label sets of the leafs. It is generally advantageous to have as small label sets as possible, since this reduces possibility for dependence and interference. Further Block Compression does not influence Leaf Pruning. Thus, if the planner performs both

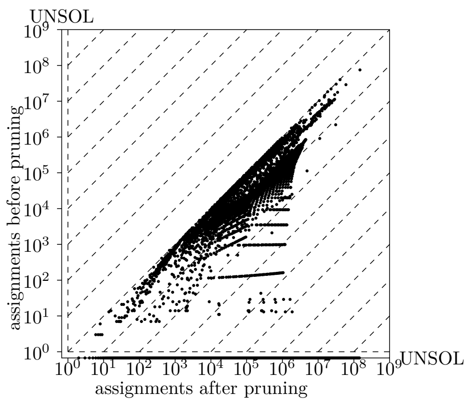


Figure 3: For every PDT created by SAT-1iB: The number of actions in the label sets of leafs before and after pruning.

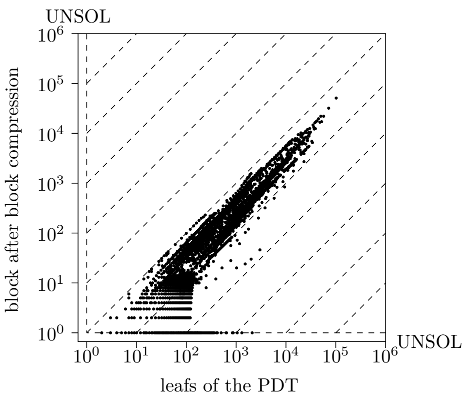


Figure 4: For every PDT created by SAT-MB: The number of leafs of the PDT and of blocks after block compression.

Leaf Pruning and Block Compression, we always perform Leaf Pruning first and Block Compression second.

Example Consider again the PDT depicted in Fig. 1b – without any pruning applied to it. We start Block Compression with a singleton block containing only the left-most leaf labelled with $\{a\}$. Next, we consider whether we can add the second leaf – labelled with $\{b\}$ – to that block. We can, since a and b are non-interfering (neither has delete effects) and b does not depend on a (b's only precondition is not an effect of a). Next, we consider whether to add the third leaf labelled with $\{c, d\}$ to the block. Both c and d are clearly non-interfering with a and b, and d is non-dependent on a and b. c is also non-dependent on a and b. In this situation b would be dependent on c, but b will necessarily be executed before c so this dependence is irrelevant – b can only be executed if y is already true in the initial state. Next, we try to add the fourth leaf labelled with $\{e, f\}$ to the block. Again e and f are non-interfering with the actions in the block a, b, c, and d. However both e and f are dependent on these actions. e is specifically dependent on c because it has the precondition y and on d because of z. Similarly, f is dependent on d. Thus, the so-far constructed block is finished and we start a new one whose first leaf is the fourth leaf. Next we add the last leaf of the PDT labelled with $\{g\}$ to this second block. As a result of Block Compression we get only two blocks for the five leafs. One block consists of the first three leafs, the second block of the latter two leafs. As a result, we have to encode only three states – while we would have to encode six states without Block Compression.

Evaluation

To determine whether leaf pruning and block compression are empirically effective, we have conducted the following evaluation. First, we compare our planner pandaPIsatt[2] against the two already existing SAT-based TOHTN planners Tree-REX (Schreiber et al. 2019) and PANDA-SAT (Behnke, Höller, and Biundo 2018a). Since Tree-Rex and PANDA-SAT do not accept the same input format, the first comparison uses only the domains by Schreiber et al. (2019), for which identical versions in both input languages are available. Second, we compare pandaPIsatt against the planners competing in the 2020 International Planning Competition (IPC) using the IPC domains[3] as well as PANDA-SAT and the heuristic-search planner by Höller et al. (2020). Further data is available (Behnke 2021).

Planners We have implemented pandaPIsatt in C++. As our SAT solver we use Cryptominisat 5.8.0 (Soos, Nohl, and Castelluccia 2009). We also used it for PANDA-SAT (Behnke, Höller, and Biundo 2018a) and Tree-Rex (Schreiber et al. 2019). Thus any differences in performance cannot be attributed to the SAT solver.

We consider 12 different configurations of pandaPIsatt. We have tested three versions of leaf pruning: only abstract pruning (no marker), unary invariant pruning (marked 1), and binary invariant pruning (2). We always propagate pruning information. We have tested both the encoding with and without added invariants (i) and with and without block compression (B). We e.g. denote with pandaPIsatt-1IB: unary invariant pruning, added invariants, and block compression. The base version pandaPIsatt uses exactly the same encoding as PANDA-SAT, it just differs in programming language and grounding. For grounding, we use the grounder of pandaPI (Behnke et al. 2020).

We compare our planner to a wide range of TOHTN planners. For the heuristic search planner (Höller et al. 2020) we use greedy search and the recently introduced taskhash loop detection (Höller and Behnke 2021). As the inner heuristics, we use ADD (Bonet and Geffner 2001), FF (Hoffmann and Nebel 2001), and LM-cut (Helmert and Domshlak 2009) heuristics. The respective planners are denoted with Greedy ADD, FF, and LMC. We have further included all planners of the International Planning Competition 2020: HyperTensioN (Magnaguagno, Meneguzzi, and de Silva 2021), Lilotane (Schreiber 2021a), PDDL4J (Pellier and Fiorino 2021), SIADEX (Fernandez-Olivares, Vellido, and Castillo 2021), and pyHiPOP (Lesire and Albore 2021).

[2]satt = *sat t*otal order. German satt means full (saturated).

[3]https://github.com/panda-planner-dev/ipc2020-domains

Note that Lilotane is a "successor" of Tree-REX (Schreiber et al. 2019) that also uses a SAT encoding, which encodes the HTN structure in a lifted fashion. Some of the IPC participants have fixed bugs after the IPC (notably HyperTensioN) or added further improvements to the planner (Lilotane (Schreiber 2021b)). We compare pandaPIsatt with these updated versions. Lastly, we include PANDA's SAT planner (Behnke, Höller, and Biundo 2018a) and a modified version (PANDA SAT+G) that uses the grounder of Behnke et al. (2020) – as PANDA's was shown to be very slow on certain domains (Wichlacz, Torralba, and Hoffmann 2019).

Results The comparison with Tree-REX is presented in Tab. 3. Three configurations of SAT (iB, 1i, 1iB) solve all 202 planning problems. The base configuration SAT already solves two instances more than Tree-REX, but there is no pronounced difference between the configurations of SAT with only a range of four instances. However, the difference is overall not very pronounced suggesting that this benchmark set is too small for a meaningful comparison. Since SAT and PANDA SAT use the same encoding and differ only in programming language and grounding, we can see the effect of these changes: an increase from 171 to 198. Grounding alone contributes only two additional instances.

The performance of the planners on the IPC 2020 domain set is shown in Tab. 4. We state the IPC score ($\min\{1, 1 - \log(t)/\log(1800)\}$ where t is the runtime in seconds), total raw coverage, and the normalised coverage of all planners. Both IPC score and normalised coverage divide the score per instance by the number of instance of that domain, i.e. the maximum score per domain is 1. All except for two of pandaPIsatt's configurations (base and 2B) have a higher coverage and normalised coverage than all IPC competitors. Note that these are the updated versions of IPC planners. If we were to consider the base IPC competitors, even the base variant would have a higher coverage than all competitors (the winner HyperTensioN solved 544 instances and Lilotane 540 in their competition versions, while our base version solves 548). The difference between pandaPIsatt and PANDA SAT is quite pronounced on the IPC benchmark set (548 vs 301 or 346 with grounding).

There is a pronounced difference between the configurations of pandaPIsatt. Binary invariant pruning does not seem to pay off when compared to unary invariant pruning. When comparing the two best configurations (1iB and 2iB) we see a difference of 43 instances. This can be attributed to the higher runtime of binary invariant pruning (quadratic in $|V|$) compared to unary invariant pruning (linear). No one new technique can solely be made responsible for the increased performance of the best configuration pandaPIsatt-1iB – which seems to benefit from synergy effects between the techniques. The three individual changes result in 575 (for 1), 587 (for B), and 592 (for i) solved instances, while the combination yields 640. The second best configuration (iB) still has only 624 solved instances. The increase of 16 instances from iB to 1iB is distributed over several domains, hinting at a general improvement.

Next, we consider the strength of unary invariant pruning. For this, we consider the 20088 PDTs constructed by 1iB. Of these, 10648 PDTs (53%) were fully pruned, i.e. binary invariant pruning could show that no solution exists. If we consider the cumulative number of primitives in the label sets $\alpha(\ell)$ for all leafs ℓ of the remaining 9440 PDTs, on average (per PDT) 38.8% of all primitives were pruned. Fig. 3 shows a scatter plot of the cumulative number of primitive leaf labels before and after pruning. If we consider all label sets of in the tree, on average 37.47% are pruned. For comparing the pruning strength of unary and binary invariants, we consider the 14676 PDTs constructed by both methods (1iB and 2iB). Of these 9119 were already shown to be unsolvable by unary invariant pruning. For 1015 PDTs, binary invariant pruning has exactly the same result as unary invariant pruning. Of the remaining 4542 PDTs, binary invariant pruning could show unsolvability of 942. For the then remaining PDTs, binary invariant pruning removed an additional 4.30% of primitive actions assigned to leafs over unary invariant pruning. Lastly, we discuss the effectiveness of block compression. To eliminate the additional effects of pruning (pruning removes actions from leafs which may allow for more block compression), we consider Block Compression in the configuration iB. In Fig. 4, we show for every constructed PDT its number of leafs and the number of blocks that were computed. On average, Block Compression reduced the number of time steps necessary to encode the PDT by 38.81%. The median reduction was 47.84%. There was no case where Block Compression could not perform any compaction.

Conclusion

We presented two new techniques for SAT-based TOHTN planning: leaf pruning and block compression. Leaf pruning removes methods and tasks from consideration when creating the SAT formula by showing that any decomposition that contains them at specific points cannot lead to an executable plan. Block compression reduces the number of time-steps necessary to encode executability of the resulting plan. We showed that both techniques improve the empirical performance of a SAT-based TOHTN planner. For future work, we may improve the pruning techniques further and extend them to also be able to handle partially-ordered HTN panning problems (Behnke, Höller, and Biundo 2019a, 2018b).

		pandaPIsatt-iB	**pandaPIsatt-1i**	**pandaPIsatt-1iB**	**pandaPIsatt-i**	**pandaPIsatt-B**	**pandaPIsatt-2iB**	**pandaPIsatt-1**	**pandaPIsatt-2i**	**pandaPIsatt-2B**	**pandaPIsatt-1B**	**pandaPIsatt**	**pandaPIsatt-2**	Tree-REX	PANDA SAT +G	PANDA SAT
Barman	20	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
Blocksworld	20	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	18	19
Childsnack	20	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	13
Depots	20	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
Entertainment	12	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**
Gripper	20	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
Hiking	20	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	19	12	16
Rover	20	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	9	9
Satellite	20	**20**	**20**	**20**	19	19	19	18	18	18	17	16	16	15	10	10
Transport	30	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
	202	**202**	**202**	**202**	201	201	201	200	200	200	199	198	198	196	171	169

Table 3: Coverage of SAT-based planners on the benchmark of Schreiber et al. (2019).

		Greedy RC ADD	Greedy RC FF	**pandaPIsatt-1iB**	**pandaPIsatt-iB**	**pandaPIsatt-1i**	**pandaPIsatt-1B**	**pandaPIsatt-2iB**	**pandaPIsatt-i**	**pandaPIsatt-B**	**pandaPIsatt-2i**	**pandaPIsatt-1**	**pandaPIsatt-2**	HyperTensioN	**pandaPIsatt-2B**	Greedy RC LMC	Lilotane	**pandaPIsatt**	PANDA SAT +G	PANDA SAT	PDDL4J	SIADEX	pyHiPOP
AssemblyHierarchical	30	**.93**	.91	.15	.15	.14	.14	.15	.15	.14	.15	.14	.14	.08	.14	.19	.13	.14	.11	.11	.06	.00	.02
Barman-BDI	20	.73	.86	.79	.77	.74	.70	.72	.73	.68	.71	.67	.67	**1.0**	.68	.54	.75	.67	.37	.00	.49	.92	.00
Blocksworld-GTOHP	30	**.88**	**.88**	.76	.63	.73	.79	.71	.60	.67	.69	.78	.78	.43	.73	.74	.71	.63	.24	.32	.43	.34	.01
Blocksworld-HPDDL	30	.72	.65	.12	.12	.11	.10	.11	.11	.10	.10	.10	.10	**.89**	.10	.26	.02	.10	.05	.00	.00	.00	.00
Childsnack	30	.68	.65	.70	.70	.71	.72	.72	.71	.72	.71	.71	.70	**1.0**	.70	.41	.87	.71	.35	.19	.47	.50	.00
Depots	30	.73	.85	.85	.78	.80	**.86**	.77	.74	.79	.75	.81	.81	.76	.77	.75	.74	.76	.40	.45	.60	.70	.00
Elevator-Learned	147	.63	.63	.91	.88	.85	.49	.81	.80	.49	.78	.47	.47	**1.0**	.49	.56	.76	.47	.24	.24	.01	.07	.01
Entertainment	12	**.95**	**.95**	**.95**	**.95**	**.95**	**.95**	.91	.94	**.95**	.88	**.95**	**.95**	.54	.94	**.95**	.16	**.95**	.65	.59	.27	.00	.07
Factories-simple	20	**.32**	.27	.29	.29	.28	.22	.29	.27	.22	.27	.21	.21	.14	.22	.21	.18	.21	.14	.14	.00	.00	.01
Freecell-Learned	60	.06	.08	**.10**	.09	.08	.09	.07	.07	.07	.06	.08	.08	.00	.07	.00	.08	.05	.00	.00	.00	.00	.00
Hiking	30	.72	.72	.65	.63	.65	.63	.63	.62	.62	.63	.62	.62	**.83**	.61	.32	.69	.60	.13	.21	.39	.00	.00
Logistics-Learned	80	.45	.48	**.69**	.62	.50	.52	.30	.43	.48	.26	.41	.41	.26	.31	.51	.34	.37	.13	.15	.00	.00	.00
Minecraft-Player	20	.07	.07	.09	.07	.09	.09	.09	.07	.09	.08	.09	.09	**.25**	.09	.02	.12	.07	.00	.00	.03	.13	.00
Minecraft-Regular	59	.58	.58	.49	.45	.43	.50	.38	.40	.47	.37	.43	.43	**.88**	.38	.45	.43	.40	.20	.14	.32	.33	.00
Monroe-FO	20	.49	.50	.72	.62	.66	.74	.44	.60	.65	.32	.72	.72	**.97**	.45	.24	.89	.64	.00	.02	.57	.25	.00
Monroe-PO	20	.22	.25	.58	.57	.54	.57	.31	.51	.58	.23	.57	.58	.00	.33	.16	**.84**	.54	.00	.02	.03	.00	.00
Multiarm-Blocksworld	74	**.83**	.33	.14	.14	.13	.13	.13	.13	.13	.12	.12	.12	.11	.12	.19	.04	.12	.05	.00	.00	.01	.00
Robot	20	.93	.94	.54	.54	.54	.51	.54	.54	.51	.54	.50	.50	**.96**	.51	.78	.52	.50	.39	.00	.27	.00	.05
Rover-GTOHP	30	.60	.52	.61	.57	.59	.55	.51	.59	.53	.50	.54	.53	**.92**	.50	.38	.57	.52	.20	.20	.62	.77	.14
Satellite-GTOHP	20	.71	.59	.75	.69	.66	.69	.63	.60	.72	.58	.67	.64	**1.0**	.62	.45	.62	.62	.28	.30	.73	.00	.19
Snake	20	.90	.89	.84	.76	.84	.83	.85	.77	.73	.84	.84	.83	**1.0**	.85	.71	.96	.77	.36	.30	.71	.29	.03
Towers	20	.50	.50	.33	.30	.33	.30	.34	.30	.28	.33	.31	.30	**.77**	.31	.46	.39	.28	.17	.00	.58	.47	.09
Transport	40	.70	.61	.90	.89	.89	.79	.85	.88	.78	.85	.79	.78	**1.0**	.81	.49	.77	.77	.50	.49	.70	.03	.23
Woodworking	30	.66	.66	.76	.72	.77	.76	.76	.73	.72	.76	.75	.76	.22	.74	.55	**.99**	.72	.39	.39	.17	.10	.09
Coverage	892	**710**	663	640	624	613	598	597	592	587	578	575	575	572	569	567	560	548	346	301	268	186	46
Normalised Coverage		**18.8**	18.2	17.0	16.5	16.5	16.1	16.1	15.9	15.8	15.5	15.7	15.8	15.7	15.5	14.6	14.9	15.1	9.8	8.1	10.0	6.1	1.6
IPC Score		**15.0**	14.4	13.7	12.9	13.0	12.7	12.0	12.3	12.1	11.5	12.2	12.3	**15.0**	11.5	10.3	12.5	11.6	5.3	4.3	7.5	4.9	0.9

Table 4: Performance of planner on the IPC 2020 domains. Each cell contains per planner and domain the number IPC-Score for that domain. The bottom three rows indicate the total coverage, total normalised coverage, and total IPC score. Maxima are indicated in bold.

References

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

Behnke, G. 2021. 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 Biundo, S. 2018a. totSAT – Totally-Ordered Hierarchical Planning through SAT. In *Proc. of the 32nd AAAI Conf. on AI (AAAI 2018)*, 6110–6118. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2018b. Tracking Branches in Trees – A Propositional Encoding for solving Partially-Ordered HTN Planning Problems. In *Proc. of the 30th Int. Conf. on Tools with Art. Int. (ICTAI 2018)*, 73–80. IEEE Computer Society. doi:10.1109/ICTAI.2018.00022.

Behnke, G.; Höller, D.; and Biundo, S. 2019a. Bringing Order to Chaos – A Compact Representation of Partial Order in SAT-based HTN Planning. In *Proc. of the 33rd AAAI Conf. on AI (AAAI 2019)*, 7520–7529. AAAI Press. doi: 10.1609/aaai.v33i01.33017520.

Behnke, G.; Höller, D.; and Biundo, S. 2019b. Finding Optimal Solutions in HTN Planning – A SAT-based Approach. In *Proc. of the 28th Int. Joint Conf. on AI (IJCAI 2019)*, 5500–5508. IJCAI. doi:10.24963/ijcai.2019/764.

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

Bonet, B.; and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 120(1-2): 5–33.

Cook, S. 1971. The Complexity of Theorem-proving Procedures. In *Proc. of the Third Ann. ACM Symp. on Theory of Computing (STOC 1971)*, 151–158. ACM.

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)*.

Geier, T.; and Bercher, P. 2011. On the Decidability of HTN Planning with Task Insertion. In *Proc. of the 22nd Int. Joint Conf. on AI (IJCAI 2011)*, 1955–1961. AAAI Press.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *Proc. of the 19th Int. Conf. on Autom. Plan. and Sched. (ICAPS 2009)*, 162–169. AAAI Press.

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.; and Behnke, G. 2021. Loop Detection in the PANDA Planning System. In *Proc. of the 31st Int. Conf. on Autom. Plan. and Sched. (ICAPS 2021)*. AAAI Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language Classification of Hierarchical Planning Problems. In *Proc. of the 21st Europ. Conf. on AI (ECAI 2014)*, volume 263, 447–452. IOS Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2021. The PANDA Framework for Hierarchical Planning. *KI – Künstliche Intelligenz* .

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

Kautz, H.; and Selman, B. 1996. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In *Proc. of the 13th Nat. Conf. on AI (AAAI 1996)*, 1194–1201.

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)*.

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)*.

Rintanen, J. 1998. A planning algorithm not based on directional search. In *Proc. of the 6th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 1998)*, 617–624. Morgan Kaufmann Publishers.

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

Schreiber, D. 2021a. Lifted Logic for Task Networks: TOHTN Planner Lilotane in the IPC 2020. In *Proceedings of 10th International Planning Competition: planner and domain abstracts (IPC 2020)*.

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

Schreiber, D.; Balyo, T.; Pellier, D.; and Fiorino, H. 2019. Tree-REX: SAT-based Tree Exploration for Efficient and High-Quality HTN Planning. In *Proc. of the 29th Int. Conf. on Autom. Plan. and Sched. (ICAPS 2019)*, 382–390. AAAI Press.

Soos, M.; Nohl, K.; and Castelluccia, C. 2009. Extending SAT Solvers to Cryptographic Problems. In *Proc. of the 12th Int. Conf. on Theory and App. of Sat. Testing (SAT 2009)*, 244–257. Springer. doi:10.1007/978-3-642-02777-2_24. URL https://doi.org/10.1007/978-3-642-02777-2_24.

Wichlacz, J.; Torralba, A.; and Hoffmann, J. 2019. Construction-Planning Models in Minecraft. In *Proc. of the 2nd ICAPS Workshop on Hierarchical Planning*, 1–5.