Tree-REX: SAT-based Tree Exploration for Efficient and High-Quality HTN Planning

Dominik Schreiber,[1] **Damien Pellier,**[2] **Humbert Fiorino,**[2] **Tomáš Balyo**[1]
[1]Karlsruhe Institute of Technology, Germany
[2]Université Grenoble-Alpes, France
{dominik.schreiber,tomas.balyo}@kit.edu, {damien.pellier,humbert.fiorino}@imag.fr

Abstract

In this paper, we propose a novel SAT-based planning approach to solve totally ordered hierarchical planning problems. Our approach called "Tree-like Reduction Exploration" (Tree-REX) makes two contributions: (1) it allows to rapidly solve hierarchical planning problems by making effective use of incremental SAT solving, and (2) it implements an anytime approach that gradually improves plan quality (makespan) as time resources are allotted. Incremental SAT solving is important as it reduces the encoding volume of planning problems, it builds on information obtained from previous search iterations and speeds up the search for plans. We show that Tree-REX outperforms state-of-the-art SAT-based HTN planning with respect to run times and plan quality on most of the considered IPC domains.

Introduction

HTN (*Hierarchical Task Network*) planning (Erol, Hendler, and Nau 1994) is one of the most efficient and widely used planning techniques. Being based on expressive languages, it allows to specify complex expert knowledge for real world domains. Unlike classical planning (Fikes and Nilsson 1971), the objective in HTN planning is expressed as a set of tasks to achieve. The search for a solution consists of reducing the initial tasks into subtasks satisfying sets of constraints until a set of primitive subtasks is found, which can be executed as classical planning actions. The recursive reduction of tasks into subtasks is performed by applying hierarchical planning operators named *methods*. Methods in HTN planning provide much more information about the problem to solve than in classical planning which only features primitive actions. The search space is usually significantly smaller in HTN planning, making the solving process faster and less resource-demanding.

On the other side, SAT solving is a generic problem resolution method which has already been successfully applied to classical planning before, e.g., (Kautz, McAllester, and Selman 1996). Given a propositional logic formula F, the objective of SAT solving is to find an assignment to all occurring Boolean variables such that F evaluates to *true*, or to report unsatisfiability if no such assignment exists. The general approach of applying SAT solving to planning problems has four steps: (1) enumerating and instantiating all the possible actions, (2) encoding the instantiated planning problem into propositional logic, (3) finding a solution with a SAT solver, e.g., (Eén and Sörensson 2003; Audemard and Simon 2009; Biere 2013), and (4) decoding the found variable assignment back to a valid plan. In conventional SAT planning (Kautz, McAllester, and Selman 1996), each fact and each action at each step is commonly represented by a Boolean variable. As the number of necessary actions is generally unknown in advance, the planning problem is iteratively re-encoded for a growing amount of steps, until a solution is found or the computation is cancelled. Recently, incremental SAT solving has been shown to improve the efficiency of classical SAT planning (Gocht and Balyo 2017). In incremental SAT solving, one single formula is maintained and extended over the whole iterative solving procedure, avoiding a complete re-encoding of the problem at each iteration and also letting the solver remember conflicts from past solving attempts.

In contrast to SAT-based classical planning, research on SAT-based HTN planning lay idle for nearly two decades after its initial proposal (Mali and Kambhampati 1998). Recently, the topic was revisited by (Behnke, Höller, and Biundo 2018a) and (Schreiber et al. 2019) who each proposed new, modern SAT encodings for HTN problems with totally ordered subtasks. However, these approaches do not fully exploit the potential of modern SAT solving yet, and they produce plans of improvable quality.

In this paper, we propose a new encoding called Tree-REX (Tree-like Reduction Exploration) which takes advantage of incremental SAT solving in order to rapidly explore a problem's hierarchy until an initial plan is found. Moreover, unlike the existing approaches, Tree-REX implements an anytime search that improves plan quality: when a solution is found and time can still be allocated, Tree-REX can continue its search to find shorter plans.

First, we introduce the concepts of HTN planning. Then, we present the Tree-REX encoding, compare it to previous encodings, and describe the integrated plan length optimization process. We then provide some relevant implementation

details and optimizations. A thorough evaluation of the features of our approach follows. Finally, related work is discussed and a conclusion is provided.

HTN Planning

This section introduces the foundations of HTN planning.

Operators, Methods and Tasks

A *fact* is an atomic logical proposition. A *state* s is a consistent set of positive facts. An *operator* o is a tuple $o = (\mathit{name}(o), \mathit{pre}(o), \mathit{eff}(o))$ where $\mathit{name}(o)$ is an expression featuring a set of parameters scoped over $\mathit{pre}(o)$ and $\mathit{eff}(o)$. $\mathit{pre}(o)$ defines the *preconditions* that must hold to apply the operator and $\mathit{eff}(o)$ defines the *effects* that hold in the state after the application of o.

An *action* is a ground operator, i.e. it has no free parameters. Action a is *applicable* to a state s if $\mathit{pre}(a) \subseteq s$. The resulting state s' of the application of a in a state s is defined as follows: $s' = \gamma(s, a) = (s \setminus \mathit{eff}^-(a)) \cup \mathit{eff}^+(a)$, where eff^+ (eff^-) represent the positive (negative) facts in *eff*. The application of a sequence of actions is recursively defined by $\gamma(s, \langle\rangle) = s$ and $\gamma(s, \langle a_0, a_1, \ldots, a_n\rangle) = \gamma(\gamma(s, a_0), \langle a_1, \ldots, a_n\rangle)$.

A *method* $m = (\mathit{name}(m), \mathit{pre}(m), \mathit{subtasks}(m))$ is a tuple where $\mathit{name}(m)$ features a set of parameters scoped over $\mathit{pre}(m)$ and $\mathit{subtasks}(m)$. $\mathit{pre}(m)$ defines preconditions, i.e., facts that must hold to apply m, and $\mathit{subtasks}(m)$ is the sequence of subtasks that must be executed in order to apply m. A *reduction* is a ground method. A reduction r is *applicable* in a state s if $\mathit{pre}(r) \subseteq s$.

A *task* t is a syntactic expression of the form $t(x_1, \ldots, x_n)$ where t is the task symbol and $x_1, \ldots, x_n$ its parameters. A task is *primitive* if t is the name of an operator; otherwise the task is *non-primitive*.

An action a *accomplishes* a primitive task t in a state s if $\mathit{name}(a) = t$ and a is applicable in s. Similarly, a reduction r accomplishes a non-primitive task t in a state s if $\mathit{name}(r) = t$ and r is applicable in s. We write $R(t)$ for the set of possible reductions of a non-primitive task t.

HTN Problems and Solutions

An *HTN planning problem* is a 5-tuple $P = (s_0, g, T, O, M)$ where s_0 and g are respectively the initial state and the goal defined by logical propositions, T is an ordered list of tasks $\langle t_0, \ldots, t_{k-1}\rangle$, O is a set of operators, and M is a set of methods defining the possible reductions of a task.

A *solution plan* for a planning problem $P = (s_0, g, T, O, M)$ is a sequence of actions $\pi = \langle a_0, \ldots, a_n\rangle$ such that $g \subseteq \gamma(s_0, \pi)$. Intuitively, a solution plan means that there is a reduction of T into π such that π is executable from s_0 and each reduction is applicable in the appropriate state of the world. The recursive formal definition has three cases. Let $P = (s_0, g, T, O, M)$ be an HTN planning problem. A *plan* $\pi = \langle a_0, \ldots, a_n\rangle$ is a *solution* for P iff:

Case 1. T is an empty sequence of tasks. Then the empty plan $\pi = \langle\rangle$ is the solution for P if $g \subseteq s_0$.

Case 2. The first task t_0 of T is primitive. Then π is a solution for P if there is an action a_0 obtained by grounding an operator $o \in O$ such that (1) a_0 accomplishes t_0, (2) a_0 is applicable in s_0 and (3) $\pi = \langle a_1, \ldots, a_n\rangle$ is a solution plan for the HTN planning problem:

$$P' = (\gamma(s_0, a_0), g, \langle t_1, \ldots, t_{k-1}\rangle, O, M)$$

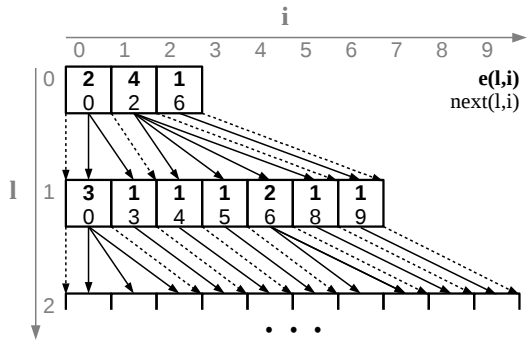

Figure 1: A sequence of hierarchical layers with $e(l, i)$ and $\mathit{next}(l, i)$ values, connected by fact transitions (dashed arrows) and action/reduction transitions (continuous arrows).

Case 3. The first task t_0 of T is non-primitive. Then π is a solution if there is a reduction r obtained by grounding a method $m \in M$ such that (1) r accomplishes t_0, (2) r is applicable in s_0 and (3) π is a solution for the HTN planning problem:

$$P' = (s_0, g, \langle \mathit{subtasks}(r), t_1, \ldots, t_{k-1}\rangle, O, M)$$

Tree-REX Encoding

The Tree-REX encoding is based on the idea of a breadth-first search over all possible reductions which can be chosen, beginning from the provided initial tasks as root nodes. Each reduction itself has a number of children each of which can either be a reduction or an action. When traversing the hierarchy layer from layer, we encode an *abstract plan* which gets more and more concrete until an actual plan (only containing actions) is reached. In the following, we define the used structures in such a way that the translation into propositional logic will be as direct and natural as possible.

Problem Hierarchies

A *hierarchical layer* is an array $L[0...n-1]$ which at each position contains a *set of elements*. An element is a fact, reduction or action. The *hierarchy* of an HTN planning problem P is a sequence of hierarchical layers $\langle L_0, \ldots, L_m\rangle$ which is computed incrementally, beginning with an initial layer that directly follows from the problem definition. A graphical representation of such a hierarchy is given in Fig. 1. When proceeding from one layer to another, some of the previous elements need to be propagated to the next layer. However, a single reduction at layer l can induce an entire *sequence* of elements at layer $l+1$, pushing all subsequent elements at layer $l+1$ further to the right. We define $e(l, i)$ as the maximum amount of subsequent positions which can be allocated by any element at (l, i) when propagating it to layer $l+1$. We can directly compute $e(l, i)$ when

we know the contents of position (l, i). Now, we can define *next*(l, i) for any position i at layer l as the first position in layer $l+1$ where the elements at (l, i) will be propagated to:

1. *next*$(l, 0) = 0$.
2. *next*$(l, i+1) =$ *next*$(l, i) + e(l, i)$.

We recursively define the hierarchical layers of the problem:

- L_0 is an array of size $k+1$. The i-th position of the array contains all the possible reductions of the initial task t_i and all of their preconditions. Additionally, position 0 contains all the facts in s_0, and position k contains all the facts in g.
- Assume that layer L_l is already defined. Then layer L_{l+1} is defined as follows:
  - For each fact p at (l, i), L_{l+1} contains p at position *next*(l, i).
  - For each action a at (l, i), L_{l+1} contains a and *pre*(a) at position *next*(l, i) and *eff*(a) at position *next*$(l, i)+1$.
  - For each reduction r at (l, i), let t_j be the j-th subtask of r. If t_j is primitive and accomplished by an action a, then L_{l+1} contains a at position *next*$(l, i) + j$. Else, L_{l+1} contains the reductions $R(t_j)$ and each of their preconditions at position *next*$(l, i) + j$.

  The size of layer L_{l+1} is defined as the highest position for which L_{l+1} contains an element.

The definition of hierarchical layers now includes all elements which *may occur* at the respective place. The actual reasoning of which combination of elements occurs at each place will be done by a SAT solver, after the hierarchical layers are encoded into propositional logic. Note that depending on the choice of which element occurs at a given place, some positions at the next hierarchical layer may also be empty in practice. For instance, suppose that an action and a reduction with three subtasks are possible at (l, i). If the action is selected to occur there, then the two places *next*$(l, i)+1$ and *next*$(l, i) + 2$ will remain empty. For this reason, a virtual action named *blank* will be incorporated into the encoding to enforce that such places remain effectively empty.

Simultaneously to picking consistent actions and reductions out of the possible sets of elements, the SAT solver also needs to find consistent assignments to all the present facts such that all the preconditions and effects of the occurring actions and reductions hold and that a fact only changes its logical value if an action or a reduction supports this change. In the following, we provide the encoding incorporating all such constraints.

Rules of Encoding

The initial state holds at the initial layer 0 at position 0:

$$\bigwedge_{p \in s_0} \textit{holds}(p, 0, 0) \wedge \bigwedge_{p \notin s_0} \neg \textit{holds}(p, 0, 0) \tag{1}$$

At each position j of the initial layer, the respective initial task reductions are possible. Let $T = \langle t_0, \ldots, t_j, \ldots, t_{k-1} \rangle$:

$$\bigwedge_{j=0}^{k-1} \bigvee_{r \in R(t_j)} \textit{element}(r, 0, j) \tag{2}$$

The last position of the initial layer contains a *blank* element:

$$\textit{element}(\textit{blank}, 0, k) \tag{3}$$

At the last position of the initial layer, all goal facts hold:

$$\bigwedge_{p \in g} \textit{holds}(p, 0, k) \tag{4}$$

The presence of an action at some position i implies its preconditions at position i and its effects at position $i+1$:

$$\textit{element}(a, l, i) \Rightarrow \bigwedge_{p \in \textit{pre}(a)} \textit{holds}(p, l, i) \tag{5}$$

$$\textit{element}(a, l, i) \Rightarrow \bigwedge_{p \in \textit{eff}^+(a)} \textit{holds}(p, l, i+1)$$

$$\textit{element}(a, l, i) \Rightarrow \bigwedge_{p \in \textit{eff}^-(a)} \neg \textit{holds}(p, l, i+1)$$

A reduction at some position i implies its preconditions at that position:

$$\textit{element}(r, l, i) \Rightarrow \bigwedge_{p \in \textit{pre}(r)} \textit{holds}(p, l, i) \tag{6}$$

Each action is primitive, and each reduction is non-primitive. The following rules eliminate the possibility of an action and a reduction to co-occur:

$$\begin{aligned} \textit{element}(a, l, i) &\Rightarrow \textit{primitive}(l, i) \\ \textit{element}(r, l, i) &\Rightarrow \neg \textit{primitive}(l, i) \end{aligned} \tag{7}$$

If a fact changes, then either this position does not contain an action yet or it contains an action which supports this fact change. Such constraints are also called "frame axioms".

$$\begin{aligned} &\textit{holds}(p, l, i) \ \wedge \ \neg \textit{holds}(p, l, i+1) \Rightarrow \\ &\Rightarrow \neg \textit{primitive}(l, i) \vee \bigvee_{p \in \textit{eff}^-(a)} \textit{element}(a, l, i) \\ &\neg \textit{holds}(p, l, i) \ \wedge \ \textit{holds}(p, l, i+1) \Rightarrow \\ &\Rightarrow \neg \textit{primitive}(l, i) \vee \bigvee_{p \in \textit{eff}^+(a)} \textit{element}(a, l, i) \end{aligned} \tag{8}$$

At each position, all possibly occurring actions are mutually exclusive. (Note that this also includes the *blank* action variable.) For each pair of actions a_1, a_2, we have:

$$\neg \textit{element}(a_1, l, i) \vee \neg \textit{element}(a_2, l, i) \tag{9}$$

A fact p holds at some position i if and only if it also holds at its first child position at the next hierarchical layer.

$$\textit{holds}(p, l, i) \Leftrightarrow \textit{holds}(p, l+1, \textit{next}(l, i)) \tag{10}$$

If an action occurs at some position i, then it will also occur at its first child position at the next hierarchical layer.

$$\textit{element}(a, l, i) \Rightarrow \textit{element}(a, l+1, \textit{next}(l, i)) \tag{11}$$

If an action occurs at some position i, then all further child positions at the next layer will contain a *blank* element.

$$\begin{aligned} &\bigwedge_{0 < j < e(l,i)} \textit{element}(a, l, i) \Rightarrow \\ &\Rightarrow \textit{element}(\textit{blank}, l+1, \textit{next}(l, i) + j) \end{aligned} \tag{12}$$

If a reduction occurs at some position i, then it implies some valid combination of its subtasks at the next layer. Let $\mathit{subtasks}(r) = \langle t_0, \ldots, t_{k-1} \rangle$ and $0 \leq j < k$. If t_j is primitive and accomplished by an action a:

$$\mathit{element}(r, l, i) \Rightarrow \mathit{element}(a, l+1, \mathit{next}(l, i) + j) \quad (13)$$

If t_j is non-primitive and $R(t_j)$ are its possible reductions:

$$\mathit{element}(r, l, i) \Rightarrow \bigvee_{r' \in R(t_j)} \mathit{element}(r', l+1, \mathit{next}(l, i) + j) \quad (14)$$

Any positions j at the next layer which remain undefined by an occurring reduction are filled with *blank* symbols.

$$\bigwedge_{k \leq j < e(l,i)} \mathit{element}(r, l, i) \Rightarrow \mathit{element}(\mathit{blank}, l+1, i+j) \quad (15)$$

To find a plan after n layers, we must ensure that all the positions of the last (i.e. the current) hierarchical layer n must be primitive. Let s_n be the size of the array at layer n:

$$\bigwedge_{0 \leq i < s_n} \mathit{primitive}(n, i) \quad (16)$$

The provided rules are instantiated and added incrementally to a solving procedure. At the beginning, the formula only consists of rules 1-4. Then, setting $l := 0$, rules 5-9 are instantiated and added to the problem, and rule 16 is added as an assumption, i.e. it is only part of the formula for one single solving attempt. The SAT solver is now executed on the formula for the first time. When the formula is not satisfiable, l is increased by one, rules 5-15 are added and rule 16 is assumed for the respective value of l. This procedure is repeated until a solution is found.

Comparison to Previous Encodings

The Tree-REX encoding lends itself to a comparison with the state-of-the-art approach totSAT (Behnke, Höller, and Biundo 2018a) because both approaches ultimately lead to an exploration of the problem's hierarchy by successively extending the SAT encoding along the hierarchy's depth. In particular, the hierarchical layers 1,...,n in our approach correspond to a Path Decomposition Tree (PDT) of depth n in totSAT. The propagation mechanism of actions from one layer to another as described in rules 11–12 resembles the *inheritPrimitive* clauses in totSAT. Similarly, the propagation of reductions from one layer to another (rules 13–15) is realized in totSAT with the clauses *applyMethod* and *selectMethod*. However, one significant difference is that totSAT features Boolean variables for possible tasks and for possible reductions (methods) at each position, whereas Tree-REX only encodes actions and reductions without any explicit notion of tasks.

Both approaches encode a problem's classical planning constraints based on the well-known encoding originally proposed by (Kautz, McAllester, and Selman 1996). However, totSAT enforces these constraints, $\mathcal{F}_E(P)$, only at the final depth, while Tree-REX features a generalization to actions and reductions (rules 5-6 and 8-9) at *every* hierarchical layer, and propagates the truth values of facts from layer to layer (rule 10). This allows our approach to enforce the consistency of any *partial plan* at a non-final layer with some non-primitive parts still remaining. In contrast, totSAT does not consider the problem's facts before the final layer: Primitive tasks are propagated to the final depth and only then seen as actions, and reduction preconditions are translated into virtual actions to be handled at the final depth as well.

To conclude, the two independently developed approaches share interesting structural similarities, but still differ by a significant margin regarding how logical constraints of the problem at hand are incorporated into the encoding.

Plan Length Optimization

When using the presented Tree-REX encoding in an incremental manner until a solution is found, it results in a solution to the planning problem at the first possible hierarchical layer. This implies an upper bound on the plan length equal to the maximum size of the final layer. However, in many cases, significantly shorter plans can exist, either on the same hierarchical layer with more empty spaces, or even on some later hierarchical layer which has not been computed yet. An HTN problem can, and often will, contain *no-operation* actions, which should not be taken into account regarding the plan length. The resulting hierarchy then essentially corresponds to a *non-shortening grammar* for which it can be difficult to find the overall shortest word. As a consequence, our planner will not try to find better plans by indefinitely increasing the amount of hierarchical layers, but instead by optimizing the plan length at the hierarchical layer where it was first found.

The procedure of the plan optimization is as follows: When the SAT solver reports satisfiability for the first time, a plan is extracted from the variable values and output as an initial solution. Then, the goal literals which previously were assumptions are added as permanent unit clauses. Afterwards, new clauses are added which provide a way to count the plan length using additional variables, similar to the Sequential Counter proposed in (Sinz 2005). The variable $\mathit{planAtLeast}(i, x)$ has the meaning: "At the final layer, the array read from position 0 up to position i contains at least x proper actions." We enforce the plan length to be at least zero at position 0, and that the plan length will never decrease when advancing to the next position:

$$\mathit{planAtLeast}(0, 0) \quad (17)$$

$$\mathit{planAtLeast}(i, x) \Rightarrow \mathit{planAtLeast}(i+1, x) \quad (18)$$

When a "proper" action a is at position i, then the plan length is increased by one:

$$\mathit{planAtLeast}(i, x) \wedge \neg\mathit{element}(\mathit{nop}, l, i) \wedge \\ \wedge \neg\mathit{element}(\mathit{blank}, l, i) \Rightarrow \mathit{planAtLeast}(i+1, x+1) \quad (19)$$

With these clauses, an assumption can now be added to prohibit any total plan length greater than some number $\hat{x}$:

$$\neg\mathit{planAtLeast}(s_n, \hat{x}) \quad (20)$$

With this mechanism, we are able to define a simple iterative optimization: When a SAT solver on the solved problem supplemented by assumption 20 reports satisfiability, then a

better plan of shorter length has been found. The optimization can then proceed with an assumption enforcing an even lower plan length. In contrast, when the solver reports unsatisfiability, this means that no plan of such length or any shorter length exists on the considered hierarchical layer. The optimization can then terminate with the last found plan.

The proposed plan optimization is an anytime algorithm: As soon as an initial solution has been found, the optimization can proceed until some limit on the computational resources is met or the plan cannot be optimized any further.

Complexity

In the following, we assess the complexity of the Tree-REX encoding in terms of clauses and variables.

Consider a problem for which after n hierarchical layers, each layer i with an array size of s_i, a plan π of length $|\pi| \leq s_n$ is found. Each action $a \in A$ is encoded at most $C := \sum_{i=1}^{n} s_i$ times (once for each position at each layer). Similarly, each fact p and each task reduction $r \in R$ has been encoded at most C times. The at-most-one constraints from rule 9 can be realized with a binary encoding (Sinz 2005) to avoid a clause complexity quadratic in the amount of actions. This technique adds at most $C \cdot \log(A)$ helper variables to the problem. Combined with exactly C variables of type *primitive*(l, i), this leads to a total variable complexity of $\mathcal{O}\big(C(A + F + R)\big)$, where F is the amount of facts in the problem.

Next, we analyze the amount of needed clauses for finding the mentioned plan: Let $E := \max\{\, |subtasks(r)| \,\big|\, r \in R \,\}$ be the maximum $e(l, i)$ which can occur in the problem. Then we require $\mathcal{O}(\sum_{i=1}^{n-1} s_i \cdot (R + A) \cdot E)$ clauses for reductions and the introduction of new *blank* symbols. Preconditions and effects add up to $\mathcal{O}(C(R + A))$ clauses. The definition of variables of type *primitive*(l, i) is included by this complexity measure as well. $\mathcal{O}(C \cdot F)$ clauses for frame axioms (rule 8) and at most $C \cdot A \log(A)$ clauses for the mentioned binary encoding of at-most-one constraints are needed. This leads to a total of $\mathcal{O}\big(\sum_{i=1}^{n-1} s_i \cdot (R + A) \cdot E + C(R + A \log(A) + F)\big)$ clauses.

Note that these complexity measures assume that at each position and at each layer, all the actions and all the reductions can occur, which is highly unlikely in practice.

For all HTN planning instances we considered, E is a small constant ($E \leq 9$ in all cases). Consequently, the determined complexities show that the size of the resulting encoding will be linear with respect to the problem size multiplied by the total amount of positions in the hierarchy.

This measure is similar to the clause complexity of the totSAT approach, where clauses are also added for each possible fact, task, and reduction at each potential step of the plan. However, note that our approach avoids to re-encode the entire problem at every iteration and instead adds new clauses to an already existing formula.

The amount of clauses and variables which are added for the means of optimizing the plan length is quadratic in s_n.

Implementation

In practice, our approach is designed as follows:

- The input instance, provided as PDDL files, is parsed and grounded into a compact representation of flat actions, reductions, and facts as proposed by (Ramoul et al. 2017).
- The application proceeds to encode the propositional logic rules in an abstract notation conceptually based on the separated DIMACS format (Gocht and Balyo 2017). Each constraint is encoded only once for each action and each reduction in the problem, and constraints which can be inferred from the sets of occurring elements are not included at all. This abstract encoding is then handed to a separate interpreter application.
- The interpreter application parses the abstract rules and assembles the initial hierarchical layer and transfers the resulting clauses to an incremental SAT solver. As long as the SAT solver reports unsatisfiability, the interpreter proceeds to calculate an additional hierarchical layer and add additional clauses, again calling the SAT solver afterwards.
- When the SAT solver finds a solution, it is transferred back from the interpreter to the main application, where the variable values are decoded into the original problem domain and returned to the user as a readable plan.

Optimizations

The presented encoding approach is designed to encode as few elements as possible in order to reduce the volume and complexity of the formulae. In the following, we briefly present the different improvements which we incorporated into Tree-REX to optimize the approach.

Sparse at-most-one constraints. At-most-one constraints as proposed in rule 9 make sure that only one action at a time is occurring at each given place. However, we have consciously refrained from encoding at-most-one constraints for pairs of reductions. It is true that it does not make sense for multiple reductions to co-occur at the same place. But when a SAT solver makes such a decision to switch two or more reduction variables to `true`, this will almost always lead to a logical conflict. The only exception is if all of these reductions result in the exact same sequence of actions, in which case the co-occurrence has no effect on the solution.

At-most-one constraints are quite expensive in a logical encoding. Common variants range from $\mathcal{O}(n^2)$ clauses without any additional variables ("pairwise encoding") over $\mathcal{O}(n)$ clauses and $\mathcal{O}(n)$ additional variables ("ladder encoding") up until $\mathcal{O}(n \log n)$ clauses and $\mathcal{O}(\log n)$ additional variables ("binary encoding") (Sinz 2005). In all cases, too many at-most-one constraints will slow down the solving procedure, either because of the large number of clauses or because the propagation of a variable assignment through the at-most-one constraint takes significant time. For this reason, we noticed that it is beneficial to instead allow multiple reductions at the same place and let the solver quickly find a conflict in such cases.

Re-usage of variables representing a fact. When a fact at some place (l, i) is assigned a truth value by the solver, then its corresponding value needs to be propagated to *next*(l, i), and to *next*$(l + 1,$ *next*$(l, i))$, and so on. Instead, we can just

use one single Boolean variable for the fact at each of these layers. This way, we do not need any clauses propagating the changes, and the total amount of needed variables is reduced.

Backwards propagation clauses. As the successors of reductions and the propagation of actions have been specified as logical clauses, the "forward" conditions for elements at the next position are fully defined. It can also help the solving process to consider the propagation in the other way, and to add the *necessary* conditions for an element to be at a certain position at the next layer. Intuitively, any action at the next layer either has already been at the corresponding parent position at the previous layer before, or it has been created by expanding an appropriate reduction at the parent position. Likewise, any reduction at the next layer must have been created by some appropriate reduction before.

Optimization and Parameter Tuning

In order to test whether the presented optimizations actually merit the total run times of the planning system compared to a more naïve approach, the tuning framework ParamILS (Hutter et al. 2009) has been employed. Given a program binary and a set of possible arguments for the program, ParamILS executes the problem on several sets of parameters and on different benchmarks in order to approximate an optimal set of parameters for which the program empirically performs best. The framework is popular especially in the field of SAT solving (Hutter et al. 2007) and automated planning (Alhossaini and Beck 2012).

For the purpose of tuning Tree-REX, each of the considered optimizations can be switched on and off separately by a parameter provided to the program. We avoided overfitting to the benchmarks of our main evaluation by only using a subset of those domains and by mixing them with domains from other benchmarks. Each execution of Tree-REX has been cut off after 3 minutes of execution time. The Tree-REX parameter tuning has been performed on a server with 24 cores of Intel Xeon CPU E5-2630 clocked at 2.30 GHz and with 264 GB of RAM, running Ubuntu 14.04.

In the best set of parameters to which ParamILS converged after over 24h of total run times, the previously proposed backwards propagation clauses are included as well as the sparse at-most-one constraints and the re-usage of variables representing facts, all of which helped speeding up the solving process.

Evaluation

In the following, we compare Tree-REX with the state-of-the-art SAT-based HTN planner totSAT (Behnke, Höller, and Biundo 2018a). The main evaluations have been conducted on a AMD Ryzen 7 1800X Eight-Core CPU with 64GB of RAM running Ubuntu 18.04.1 LTS. Each instance is cut off at five minutes of CPU time and 16GB of RAM usage. The developed software, all benchmark instances and the gathered experimental data are available online[1].

[1] https://gitlab.com/domschrei/htn-sat

Problem Benchmarks

We have used a set of problem benchmarks with a wide range of difficulty for comparing Tree-REX with totSAT. One general problem of HTN planning is that there is no standardized HTN extension for PDDL, so different planners are using varying PDDL adaptations. In our case, a translation between our benchmarks and the benchmarks used by (Behnke, Höller, and Biundo 2018a) is difficult due to various differences in the problem modelings: While our modeling features `(:method)` constructs with overloaded signatures and implicit parameters to account for multiple methods of a task, the totSAT modeling separates task definitions from methods, both with unique signatures. Moreover, the two modelings differ in which subset of advanced PDDL features they support. However, two domains previously exclusive to totSAT were successfully translated, namely Entertainment and Transport. Two further domains, Rover, and Satellite, are featured in both planners' benchmark sets. In addition, we have included six further domains from our benchmarks (Barman, Blocksworld, Childsnack, Depots, Gripper, Hiking) and translated them into the format used by totSAT.

SAT Solvers

As a backend, the totSAT configuration from AAAI 2018 (Behnke, Höller, and Biundo 2018a) uses the solver MiniSAT (Eén and Sörensson 2003). To keep the results comparable, we compiled our interpreter application with MiniSAT as well. Note that our application can be effortlessly linked with any SAT solver implementing a common and simplistic interface for incremental SAT solving called IPASIR (Balyo et al. 2016). As such, we have successfully plugged various solvers like PicoSAT (Biere 2008), Glucose (Audemard and Simon 2009), Lingeling (Biere 2013) and CryptoMiniSat (Soos 2016) into our planning system.

Results

Out of a total of 202 benchmark instances, totSAT was able to solve 164 instances while Tree-REX solved 198 instances. In particular, totSAT failed to solve the more difficult instances of Childsnack (12/20 instances solved), Hiking (11/20), Rover (8/20), Satellite (12/20) and Transport (29/30). Tree-REX only failed to find a plan for four instances, and it did not finish the plan length optimization in seven more cases. All these instances are from the Satellite and Rover domains which both feature complex coordination tasks and make heavy use of no-operations in some method subtasks. For all other instances, the plan length optimization terminated within the given computational constraints, reporting the shortest possible plan at the considered hierarchical layer.

An overview over all benchmarks is provided in Fig. 3. The label "Tree-REX-o" refers to our planning approach including the proposed plan length optimization, while the label "Tree-REX" refers to the same approach without any plan length optimization.

It can be seen that totSAT has higher run times in general. As expected, Tree-REX-o produces smaller plans than

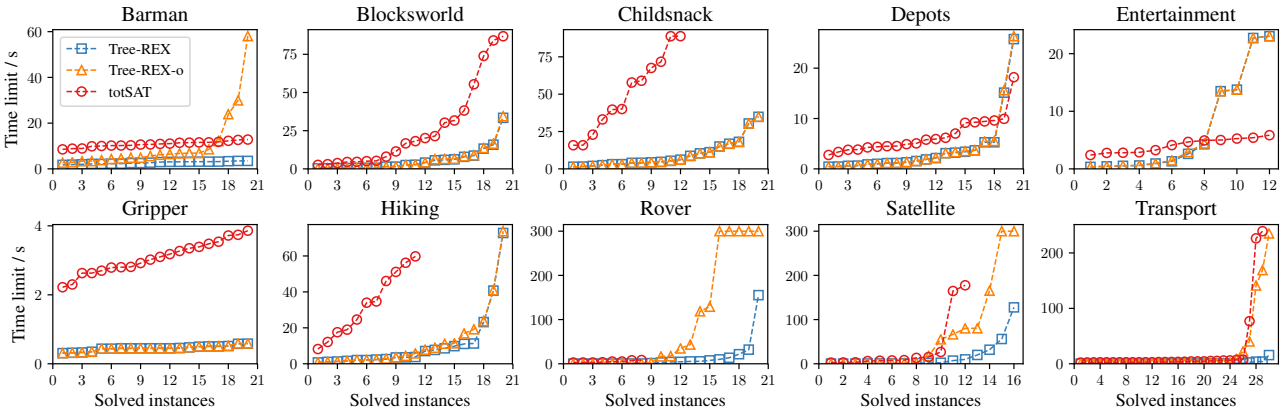

Figure 2: Cactus plots of run times for each considered benchmark domain.

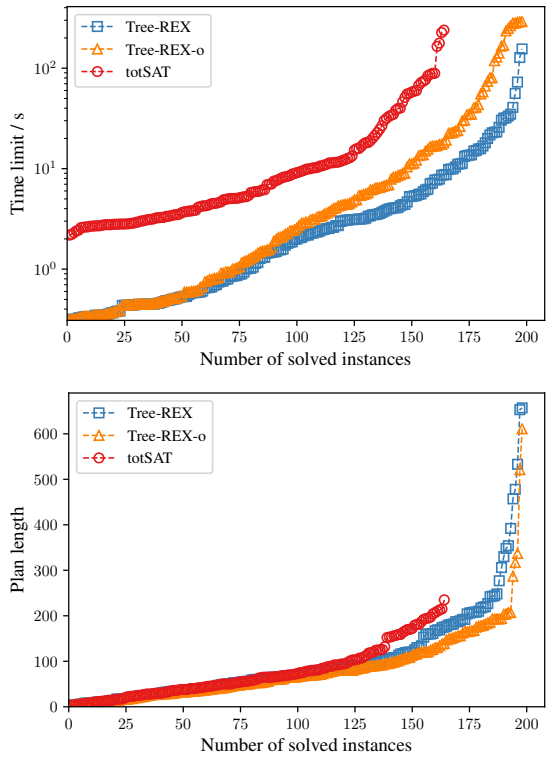

Figure 3: Run times (top, logarithmic scale) and found plan lengths (bottom, linear scale) over all benchmarks.

both totSAT and the non-optimizing Tree-REX variant. The search for better plans consequently leads to higher run times for some domains. This general picture is confirmed by Tables 1 and 2 showing that Tree-REX dominates the ICAPS scores regarding run times while Tree-REX-o dominates the scores regarding found plan lengths. Scores are calculated by adding a value of C^*/C to each competitor for each solved instance, where C is the run time (plan length) of that competitor and C^* is the minimum run time (plan length) among all competitors for this instance.

More detailed cactus plots for each domain are provided

Domain	totSAT	Tree-REX	Tree-REX-o
Barman	5.00	**20.00**	9.19
Blocksworld	4.02	**20.00**	19.22
Childsnack	0.91	**20.00**	19.99
Depots	7.81	**19.36**	18.95
Entertainment	6.73	**9.25**	8.94
Gripper	2.92	**20.00**	19.85
Hiking	0.93	**20.00**	17.93
Rover	0.87	**20.00**	10.12
Satellite	1.23	**16.00**	8.62
Transport	3.77	**30.00**	21.51

Table 1: Run time scores per domain

in Fig. 2 for run times and in Fig. 4 for plan lengths. Comparing Tree-REX to totSAT on a per-domain level, "Entertainment" is the only domain where totSAT clearly outperforms the Tree-REX variants. The problem instances of this domain lead to a heavily combinatorial amount of reductions to consider during grounding, which favors the preprocessing done inside totSAT in this case. On all other domains, totSAT performs comparably or worse than Tree-REX.

One noteworthy result is that on the domains where no plan optimization is possible at all, the optimization procedure recognizes this immediately and no additional time is wasted. In particular, the Childsnack and Gripper domains both feature a rigid hierarchical structure which enforces a fixed plan length for any valid solution. For these domains, both run times and reported plan lengths are virtually identical between Tree-REX and Tree-REX-o. In other cases such as the Rover and Satellite domains, the plan optimization procedure can take a considerable amount of time, but will also lead to significant improvements, even if the computation is cut off prematurely.

Overall, the experiments show that Tree-REX performs well on all considered benchmarks. Using the proposed plan optimization, our approach lead to better plans the more time was allotted, and almost always found the shortest possible plan at the final considered layer within the time limit.

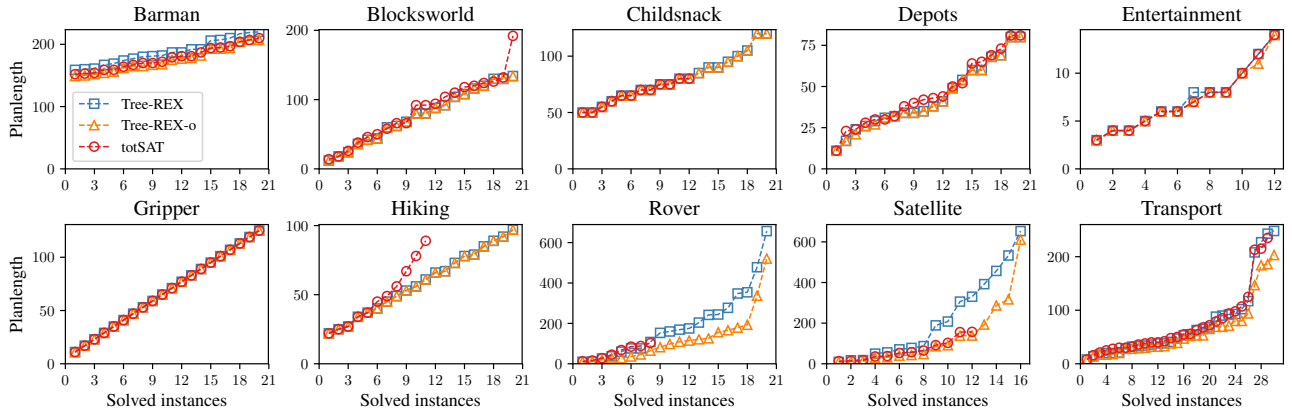

Figure 4: Cactus plots of found plan lengths for each considered benchmark domain.

Domain	totSAT	Tree-REX	Tree-REX-o
Barman	19.67	18.67	**20.00**
Blocksworld	18.36	**19.68**	**19.68**
Childsnack	12.00	**20.00**	**20.00**
Depots	18.59	19.74	**19.93**
Entertainment	11.92	11.79	**12.00**
Gripper	**20.00**	**20.00**	**20.00**
Hiking	11.00	**20.00**	**20.00**
Rover	4.83	13.08	**20.00**
Satellite	9.80	9.66	**16.00**
Transport	23.71	26.04	**30.00**

Table 2: Plan length scores per domain

Related Work

The idea to enrich a planning problem with additional knowledge of how certain tasks are realized ranges back to (Sacerdoti 1975), where a structure called procedural nets has been proposed. Since then, automated planning using Hierarchical Task Networks advanced with a range of proposed planning systems such as Nonlin (Tate 1976), O-Plan (Currie and Tate 1991) or SIPE (Wilkins 1984), with UMCP (Erol, Hendler, and Nau 1994) being the first proposed solving procedure which has been proven to be sound and complete. All of these algorithms have the common point of operating in a state-less manner; they do not maintain a set of facts at each planning step, instead they search the general space of possible plans in a unified manner.

The planner SHOP (Nau et al. 1999) and its enhancement SHOP2 (Nau et al. 2003) are among the most popular HTN planners used today (Nau et al. 2005). In contrast to previous approaches, SHOP and its enhancements are state-based and primitive actions are visited exactly in the order which they will have in the final plan. This leads to an easy identification of the applicability of reductions at some given point of the planning process. HTN planning is used in practice in various application domains such as web service compositions (Sirin et al. 2004), robot planning (Weser, Off, and Zhang 2010), and drone coordination (Bevacqua et al. 2015).

HTN planning with SAT techniques has been introduced in 1998 by (Mali and Kambhampati 1998). Their encodings share a clause and variable complexity cubic in the amount of tasks, rendering it not viable for today's HTN planning problems of non-trivial size. Current works on SAT-based totally ordered HTN planning include totSAT (Behnke, Höller, and Biundo 2018a) as well as the encodings proposed in (Schreiber et al. 2019) which introduced incremental SAT solving to hierarchical planning and served as a basis for the work at hand. A detailed comparison of Tree-REX to the latter can be found in (Schreiber 2018). Recently, novel SAT encodings for partially ordered HTN planning have been proposed (Behnke, Höller, and Biundo 2018b; 2019), expanding further on the potential use cases of SAT solving in modern planning.

Conclusion

In this paper, we presented a novel SAT-based approach for high quality HTN planning. We proposed an encoding of totally ordered HTN planning problems into SAT called Tree-like Reduction Exploration (Tree-REX). The approach results in a rapid breadth-first exploration of the problem's hierarchy by making use of incremental SAT solving, which allows to optimize the general performance and to reduce the volume of the resulting encodings. Furthermore, we presented an anytime plan length optimization stage within our approach; this is the first SAT-based hierarchical planner featuring such a technique. We have evaluated our approach and compared it against the state-of-the-art in SAT-based HTN planning, showing that Tree-REX outperforms it on most domains regarding both run times and plan lengths.

As for future work, we want to explore whether a SAT-based HTN planner could directly operate on a lifted (uninstantiated) problem and thus achieve better performance.

Acknowledgements

We would like to thank the reviewers for their detailed feedback. We also thank Gregor Behnke for his helpful cooperation regarding the comparison with totSAT, and Arnaud Lequen for preparing various HTN benchmark domains.

References

Alhossaini, M., and Beck, J. C. 2012. Macro learning in planning as parameter configuration. In *Proceedings of the Canadian Conference on Artificial Intelligence*, 13–24.

Audemard, G., and Simon, L. 2009. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 399–404.

Balyo, T.; Biere, A.; Iser, M.; and Sinz, C. 2016. SAT race 2015. *Artificial Intelligence* 241:45–65.

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

Behnke, G.; Höller, D.; and Biundo, S. 2018b. Tracking branches in trees–a propositional encoding for solving partially-ordered HTN planning problems. *Hierarchical Planning 2018* 40.

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 AI (AAAI 2019), AAAI Press*.

Bevacqua, G.; Cacace, J.; Finzi, A.; and Lippiello, V. 2015. Mixed-initiative planning and execution for multiple drones in search and rescue missions. In *Proceeding of the International Conference on Automated Planning and Scheduling*, 315–323.

Biere, A. 2008. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation* 4:75–97.

Biere, A. 2013. Lingeling, plingeling and treengeling entering the SAT competition 2013. In *Proceedings of SAT competition*, 51.

Currie, K., and Tate, A. 1991. O-Plan: the open planning architecture. *Artifical Intelligence* 52(1):49–86.

Eén, N., and Sörensson, N. 2003. An extensible SAT-solver. In *Proceedings of the International conference on theory and applications of satisfiability testing*, 502–518.

Erol, K.; Hendler, J.; and Nau, D. 1994. UMCP: A sound and complete procedure for hierarchical task-network planning. In *Proceedings of the Artificial Intelligence Planning Systems*, volume 94, 249–254.

Fikes, R., and Nilsson, N. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3-4):189–208.

Gocht, S., and Balyo, T. 2017. Accelerating SAT based planning with incremental SAT solving. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 135–139.

Hutter, F.; Babic, D.; Hoos, H. H.; and Hu, A. 2007. Boosting verification by automatic tuning of decision procedures. In *Proceeding of the conference on Formal Methods in Computer Aided Design*, 27–34.

Hutter, F.; Hoos, H.; Leyton-Brown, K.; and Stützle, T. 2009. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research* 36:267–306.

Kautz, H.; McAllester, D.; and Selman, B. 1996. Encoding Plans in Propositional Logic. In *Proceedings of the International Conference on Knowledge Representation and Reasoning*, 374–384.

Mali, A., and Kambhampati, S. 1998. Encoding HTN planning in propositional logic. In *Proceedings International Conference on Artificial Intelligence Planning and Scheduling*, 190–198.

Nau, D.; Cao, Y.; Lotem, A.; and Munoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proceedings of the international joint conference on Artificial intelligence*, 968–973.

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

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Wu, D.; Yaman, F.; Munoz-Avila, H.; and Murdock, J. W. 2005. Applications of SHOP and SHOP2. *IEEE Intelligent Systems* 20(2):34–41.

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.

Sacerdoti, E. 1975. A structure for plans and behavior. Technical report, DTIC Document.

Schreiber, D.; Balyo, T.; Pellier, D.; and Fiorino, H. 2019. Efficient SAT encodings for hierarchical planning. In *Proceedings of the 11th International Conference on Agents and Artificial Intelligence, ICAART 2019*, volume 2, 531–538.

Schreiber, D. 2018. Hierarchical task network planning using SAT techniques. Master's thesis, Grenoble Institut National Polytechnique, Karlsruhe Institute of Technology.

Sinz, C. 2005. Towards an optimal CNF encoding of boolean cardinality constraints. In *Proceedings of the International conference on principles and practice of constraint programming*, 827–831.

Sirin, E.; Parsia, B.; Wu, D.; Hendler, J.; and Nau, D. 2004. HTN planning for web service composition using SHOP2. *Web Semantics: Science, Services and Agents on the World Wide Web* 1(4):377–396.

Soos, M. 2016. The CryptoMiniSat 5 set of solvers at SAT Competition 2016. In *Proceedings of SAT Competition*, volume 2016, 28.

Tate, A. 1976. *Project planning using a hierarchic nonlinear planner*. Department of Artificial Intelligence, University of Edinburgh.

Weser, M.; Off, D.; and Zhang, J. 2010. HTN robot planning in partially observable dynamic environments. In *Proceedings of the International Conference on Robotics and Automation*, 1505–1510. IEEE.

Wilkins, D. 1984. Domain-independent planning representation and plan generation. *Artifical Intelligence* 22(3):269–301.