Journal of Artificial Intelligence Research 70 (2021) 1117-1181 Submitted 11/2020; published 03/2021

Lilotane: A Lifted SAT-Based Approach to Hierarchical Planning

Dominik Schreiber DOMINIK.SCHREIBER@KIT.EDU
Karlsruhe Institute of Technology, Kaiserstraße 12
76131 Karlsruhe, Germany

Abstract

One of the oldest and most popular approaches to automated planning is to encode the problem at hand into a propositional formula and use a Satisfiability (SAT) solver to find a solution. In all established SAT-based approaches for Hierarchical Task Network (HTN) planning, grounding the problem is necessary and oftentimes introduces a combinatorial blowup in terms of the number of actions and reductions to encode. Our contribution named Lilotane (Lifted Logic for Task Networks) eliminates this issue for Totally Ordered HTN planning by directly encoding the lifted representation of the problem at hand. We lazily instantiate the problem hierarchy layer by layer and use a novel SAT encoding which allows us to defer decisions regarding method arguments to the stage of SAT solving. We show the correctness of our encoding and compare it to the best performing prior SAT encoding in a worst-case analysis. Empirical evaluations confirm that Lilotane outperforms established SAT-based approaches, often by orders of magnitude, produces much smaller formulae on average, and compares favorably to other state-of-the-art HTN planners regarding robustness and plan quality. In the International Planning Competition (IPC) 2020, a preliminary version of Lilotane scored the second place. We expect these considerable improvements to SAT-based HTN planning to open up new perspectives for SAT-based approaches in related problem classes.

1. Introduction

Over the last decades, an advanced paradigm in domain-independent automated planning titled Hierarchical Planning has gained traction and high popularity among researchers and users alike (Georgievski & Aiello, 2015; Bercher, Alford, & Höller, 2019). Hierarchical planning and, specifically, its most noteworthy realization called Hierarchical Task Network (HTN) planning enrich a planning domain with hierarchical expert knowledge which results in better guidance for planners and in well-structured and intuitive plans. In 2020 this collective interest in hierarchical planning resulted in the first International Planning Competition (IPC) for HTN Planning taking place (Behnke, Bercher, & Höller, 2020b).

In addition to *operators* known from classical planning which are templates for valid atomic manipulations of the world state, HTN planning additionally features *tasks* which provide an abstract notion of something that needs to be achieved, and *methods* which provide conditional "recipes" for decomposing a specific task into smaller tasks. In an HTN planning problem, a number of *initial tasks* are successively decomposed by applicable methods under the notion of stepwise refinement until the resulting tasks can be achieved by primitive operators. In this work we focus on a highly popular subclass where the subtasks of every method are totally ordered, named Totally Ordered HTN (TOHTN) planning.

Among various established algorithms to resolve HTN planning problems as efficiently as possible, one popular approach is to reduce the problem to propositional satisfiabiliy (SAT). First, the HTN planning problem at hand is transformed into an easier to handle representation through a preprocessing stage called *grounding* which instantiates all possible (i.e., reachable) argument combinations of each operator and each method. Then, the structurally simpler ground problem is encoded into a sequence of propositional logic formulae and handed to a SAT solver. Eventually, if the problem is solvable then at some point a satisfying assignment will be found and can be decoded into a plan. This planning approach, after two decades of inactivity since its initial proposal (Mali & Kambhampati, 1998), recently received a significant amount of attention. On the one hand, new techniques for much more compact SAT encodings were found (Behnke, Höller, & Biundo, 2018, 2019a, 2019b; Schreiber, Pellier, Fiorino, & Balyo, 2019a; Schreiber, Pellier, Fiorino, et al., 2019b) while on the other hand new grounding approaches (Ramoul, Pellier, Fiorino, & Pesty, 2017; Behnke, Höller, Schmid, Bercher, & Biundo, 2020) serve as a catalyst to improve on SAT-based approaches, not only speeding up the processing of a planning description but also leading to smaller encodings and therefore better performance.

Wherever SAT solving has been employed for HTN planning before, the procedure of grounding was considered an obvious, unavoidable and essential stage to be done in advance. Encoding the problem on the basis of a ground representation has in fact various merits: The structures which result from grounding are logically much simpler than the lifted representation, and invalid parts of search space can be detected beforehand, hence they do not need to be encoded.

However, as a correct grounding procedure must enumerate all instantiations of operators and methods that may be part of a plan, grounding implies a combinatorial blowup in the worst case: Some planning problems inevitably result in a huge ground representation if all possibly useful parameter combinations for operators are collected. As such, grounding can also be a very heavyweight task and become a bottleneck for the whole planning procedure in terms of run time and memory footprint (e.g., Wichlacz, Torralba, & Hoffmann, 2019). For these reasons, planners which rely on a ground representation may fail to scale to large problems when compared to lifted planners which do not perform any grounding, especially on logically simple problems which lead to a huge ground representation.

With our contributions, we circumvent the problems tied to grounding for TOHTN planning by designing a lifted SAT-based approach that omits this phase of the planning procedure. To our knowledge we are the first to explore a lifted SAT encoding for hierarchical planning: Our approach *Lilotane* ('lı·lo·teın, Lifted Logic for Task Networks) generates an incremental sequence of propositional formulae from a lifted TOHTN problem description by instantiating operators and methods in a lazy manner and keeping free arguments where appropriate. As such, Lilotane defers any non-trivial argument substitution choices up until the stage of SAT solving and therefore follows the well-known *least-commitment principle* which suggests to defer a planner's decisions for as long as possible (Weld, 1994).

This greatly affects our planning approach as a whole: In addition to the obvious merit of omitting the stage of grounding, our encoding does not suffer from any combinatorial blowup with respect to the input size, promising much smaller encodings. This reduction of the encoding size, again, has an immediate effect on run times – fewer clauses need to be computed and handed to the SAT solver – and leads to shorter solving times and a reduced memory footprint. In terms of plan quality, we improve on the anytime plan improvement approach proposed by Schreiber et al. (2019b): We employ incremental SAT solving to iteratively tighten the bounds on possible plan lengths at a certain depth and obtain successively shorter plans in the process.

In extensive evaluations we find that our planning approach outperforms state-of-the-art SAT-based HTN planners by more than one order of magnitude on the majority of instances and in most cases produces much smaller SAT encodings of TOHTN planning problems which, in turn, leads to a considerably smaller memory footprint. Lilotane also compares favorably to other state-of-the-art HTN planners: A preliminary version of Lilotane participated in the IPC 2020 and scored the second place. We analyze the results and evaluate the current version of Lilotane against the IPC winner, concluding that Lilotane often requires more time to find solutions but nevertheless is an appealing engine for TOHTN planning based on its robustness and effective quality awareness. Moreover, by showing in theory and practice that SAT-based HTN planning does not necessarily rely on grounding in order to be efficient, we open up new perspectives for scalable SAT-based planning.

The paper is structured as follows: First, in Chapter 2 we provide a problem definition and introduce important concepts of HTN planning and SAT solving. We then discuss previous work with a focus on grounding and SAT-based approaches in Chapter 3. Chapter 4 provides an overview on our planning approach and describes its different components as well as a number of optimizations and improvements. Chapter 5 features the complete propositional logic encoding we employ as well as a proof of correctness and a worst-case analysis of the number of variables and clauses in the encoding. We present an anytime plan improvement technique for our approach in Chapter 6. In Chapter 7 we evaluate our approach, first in the scope of SAT-based approaches and then in the scope of HTN planning in general. We also discuss the results of the IPC in this chapter. A conclusion and an outlook follow in Chapter 8.

2. Preliminaries

In this chapter we introduce the necessary preliminaries for presenting our contributions.

2.1 TOHTN Planning

In the past, several formalisms for HTN planning of differing expressive power have been introduced (see Erol, Hendler, & Nau, 1996; Alford, Bercher, & Aha, 2015). We adapt and refine the model used by Schreiber et al. (2019b) as their work provides the foundation for our approach. In terms of expressiveness, this notation is equivalent to TOHTN planning with variables as specified by Alford et al. (2015) and compatible with the TOHTN formalism used in the International Planning Competition 2020 (Behnke et al., 2020b).

2.1.1 HTN Structures

We begin with some basic definitions. A *constant* $c \in C$ is an atomic symbol from some domain C. A *signature* $\sigma(a_1, \dots, a_k)$ is a syntactical construct consisting of a name σ and a list of $k \geq 0$ *arguments*. Thereby k is fixed for each σ; we call k the *arity* of σ. Each argument a_i is either a constant or a *variable*; a variable acts as a placeholder for a constant.

The possible domain of each a_i of σ is limited to some fixed subset $\tau_i \subseteq C$ of constants, the *type* of the i-th argument of σ. We call a signature *ground* if all of its arguments are constants, i.e., $\forall i \in \{1, \dots, k\} : a_i \in C$. We call a signature *lifted* if it is not ground. We use the term *free arguments* to refer to any variables left in a signature.

Predicates are special signatures which represent propositions, i.e., logical atoms, of the world state of our problem. A *literal* is a predicate supplied with a polarity (positive or negative). For any set s of literals, we define $s^+ := \{p \in s \mid p \text{ is positive}\}$ and $s^- := \{\neg p \mid p \in s,\ p \text{ is negative}\}$.

A *fact* is a ground literal. A *state* s is a set of positive facts. When we interpret a state s as the world state of a planning problem, in accordance to the closed-world assumption (see Reiter, 1981) we postulate that every fact *not* contained in s is negative.

A *task* is a non-predicate signature $t(a_1, \dots, a_k)$ and, in itself, a purely syntactical footprint of something that needs to be achieved. An *operator* $o = (\mathit{sig}(o), \mathit{pre}(o), \mathit{eff}(o))$ is a tuple of a task $\mathit{sig}(o)$ and two sets $\mathit{pre}(o)$, $\mathit{eff}(o)$ of literals whose arguments are arguments of $\mathit{sig}(o)$. An *action* is an operator o where $\mathit{sig}(o)$ is ground.

A *method* is a tuple $m = (\mathit{sig}(m), \mathit{task}(m), \mathit{pre}(m), \mathit{subtasks}(m))$ where $\mathit{sig}(m)$ is a non-predicate signature, $\mathit{task}(m)$ is a task, $\mathit{pre}(m)$ is a set of literals, $\mathit{subtasks}(m) = \langle t_1, \dots, t_j \rangle$ is a sequence of $j \geq 0$ tasks, and all arguments in $\mathit{task}(m), \mathit{pre}(m)$, and $\mathit{subtasks}(m)$ are arguments of $\mathit{sig}(m)$. A *reduction* is a method m where $\mathit{sig}(m)$ is ground. Note that in general HTN planning, $\mathit{subtasks}(m)$ is not a sequence but rather a set of tasks supplied with a precedence relation. We will not pursue this more general case any further but restrict subtasks to be totally ordered, hence the name Totally Ordered HTN planning.

We use the term *operation* to refer to an object that is either an action or a reduction.

Operators and methods are "recipes" to achieve an existing task, either by applying a matching operator that alters the world state or by replacing the task with the subtasks of a matching method. In both cases, the structures' preconditions $\mathit{pre}(\cdot)$ need to hold in the world state immediately before this refinement is performed. Given a method m for some task t such that $\mathit{task}(m) = t$, we say that m *matches* t and that t is *compound.* Given an operator o for some task t such that $\mathit{sig}(o) = t$, we say that o matches t and that t is *primitive.* Every task we consider in a problem is *either* compound *or* primitive.

2.1.2 Problem Definition

A *TOHTN domain* (Totally Ordered Hierarchical Task Network) $D = (C, P, O, M)$ consists of constants C, predicates P, operators O, and methods M. The domain's actions A and reductions R are defined as the result of exhaustively substituting the arguments in each operator/method with all possible combinations of constants.

A *TOHTN problem* $\Pi = (D, s_I, T)$ consists of a TOHTN domain D, *initial state* s_I, and *initial task network* T. Thereby s_I is a state and T is a list of ground tasks. In the following, let "$\circ$" denote the concatenation of two sequences.

Definition 1. *A sequence of actions π is a* solution *to a TOHTN problem $\Pi = (D, s_I, T)$ iff one of the following cases holds and the resulting recursion is well-defined.*

1. ***(Base case.)*** $\pi = \langle\rangle$ *and* $T = \langle\rangle$*.*

2. ***(Applying a reduction.)*** *$T = \langle t \rangle \circ T'$, $t = sig(r)$ for some $r \in R$, $pre^+(r) \subseteq s_I$, $pre^-(r) \cap s_I = \emptyset$, and π is a solution to $\Pi' := (D, s_I, subtasks(r) \circ T')$.*

3. ***(Applying an action.)*** *$T = \langle t \rangle \circ T'$, $t = sig(a)$ for some $a \in A$, $pre^+(a) \subseteq s_I$, $pre^-(a) \cap s_I = \emptyset$, and $\pi' := \langle a \rangle \circ \pi$ is a solution to $\Pi' := (D, (s_I \setminus eff^-(a)) \cup eff^+(a), T')$.*

Note that this definition directly provides a recursive algorithm to resolve a TOHTN problem – popular progression search planners such as SHOP (Nau, Cao, Lotem, & Munoz-Avila, 1999) are essentially refinements of this algorithm. Alternative 1 solves the empty problem where there is nothing to achieve ($T = \langle\rangle$) hence no actions are performed ($\pi = \langle\rangle$). In alternative 2, a reduction r is applied which matches the current first task and whose preconditions hold in s: The matched task is replaced with the subtasks of r. In alternative 3, an action a is applied which matches the current first task and whose preconditions hold in s: a is appended to the plan, its effects are applied to the current state, and the matched task is removed from the list of tasks yet to achieve.

The decisions left to a planner which follows the above algorithm are limited to picking a reduction whenever the second case is encountered. This includes both the decision for a particular method and the choice of substitutions that ground the method into a fitting reduction. The third case does not induce any decisions: the tasks in T are invariantly ground, so there can only be one particular action matching any given task $t \in T$.

To understand, reproduce, and verify a solution, it should not only contain a flat sequence of actions but the full trace leading to this plan, which we define as follows:

Definition 2. *A directed tree $H = (V, E)$ with a total node ordering relation $\prec \subseteq V \times V$ is a* hierarchical solution *to a problem Π iff (1)–(3) hold.*

(1) Each leaf node v corresponds to some action $a_v \in A$, and each inner node v corresponds to some reduction $r_v \in R$. In particular, the root node $\hat{v}$ corresponds to the initial reduction, *i.e., a reduction r_0 with $subtasks(r_0) = T$.*

(2) If an inner node u has k outgoing edges $(u, v_1), \dots, (u, v_k)$, sorted such that $v_i \prec v_j$ if $i < j$, each v_i corresponds to an operation which matches the i-th subtask of r_u.

(3) Let $\Omega := \langle o_1, o_2, \dots, o_k \rangle$ be a sequence of operations which results from a depth-first traversal of H beginning from $\hat{v}$ and using $\prec$ as an ordering. Specifically, if node v is visited, its corresponding operation is appended to Ω and all children of v are added according to "$\prec$" to the frontier of nodes to visit (i.e., v_1 is visited before v_2 if $v_1 \prec v_2$). Then there is a sequence of actions π such that (A) holds:

(A) π is a solution for Π according to Def. 1 where either case 1 applies or case 2 with $r := o_1$ applies or case 3 with $a := o_1$ applies; and in the two latter cases, (A) holds recursively for the resulting π', the resulting Π', and for $\Omega' := \langle o_2, \dots, o_k \rangle$.

Essentially, Def. 2 defines data structure H as a witness for a particular "path" to take through Def. 1 to obtain the given classical solution π. In particular, π can be read from H just by enumerating all leaf nodes in H according to $\prec$. The structure of H closely resembles the plan output format (Behnke, Bercher, & Höller, 2020a) required for the International Planning Competition 2020, and H can be transformed easily into this desired output.

Erol, Hendler, and Nau (1994) showed that general HTN planning is a strictly semi-decidable problem. By contrast, their findings also include that TOHTN planning is decidable due to its more rigid and predictable structure. Notably, the property that makes TOHTN planning decidable is that it prevents arbitrary interleavings of subtasks. However, Alford et al. (2015) have shown that TOHTN planning in our setting, i.e., with variables, is 2-EXPTIME-complete and as such conjectured to be strictly more difficult than classical automated planning (which Bylander, 1994 showed to be PSPACE-complete).

2.1.3 Input Definition

Our planner accepts planning problems in the form of HDDL (Hierarchical Domain Description Language) files. HDDL has been proposed recently to consolidate a common input language for hierarchical planning (Höller, Behnke, Bercher, Biundo, Fiorino, Pellier, & Alford, 2020) and was used for the IPC 2020 (Behnke et al., 2020b).

The TOHTN model we just described is a subset of HDDL. Most importantly, we require the subtasks of each method to be totally ordered. Furthermore, HDDL models can feature a propositional goal, i.e., a set g of facts which must hold in the end when T is empty. We emulate this constraint within our model by appending an additional "goal operator" o_g with $pre(o_g) = g$ and $eff(o_g) = \emptyset$ to T. Thirdly, HDDL allows to specify (in)equality constraints for pairs of arguments. We treat these as additional preconditions.

One more point to note is that we alter the problem such that exactly one (virtual) reduction r_0 is the hierarchy's root which then features the initial tasks T as its subtasks (see Def. 2). This transformation originates from pandaPIparser (Behnke et al., 2020), a parser for HDDL problems which we make use of. Therefore, we adopted this simplification.

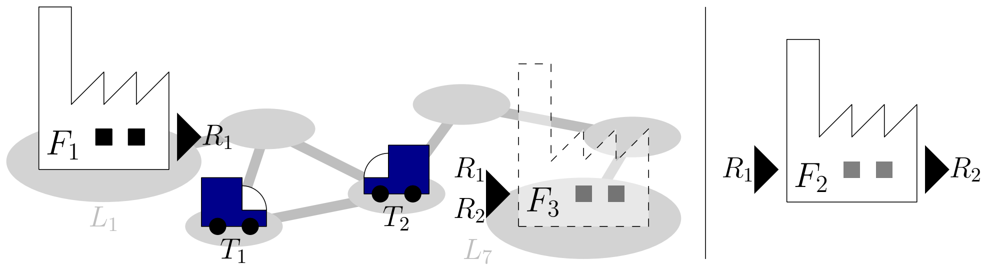
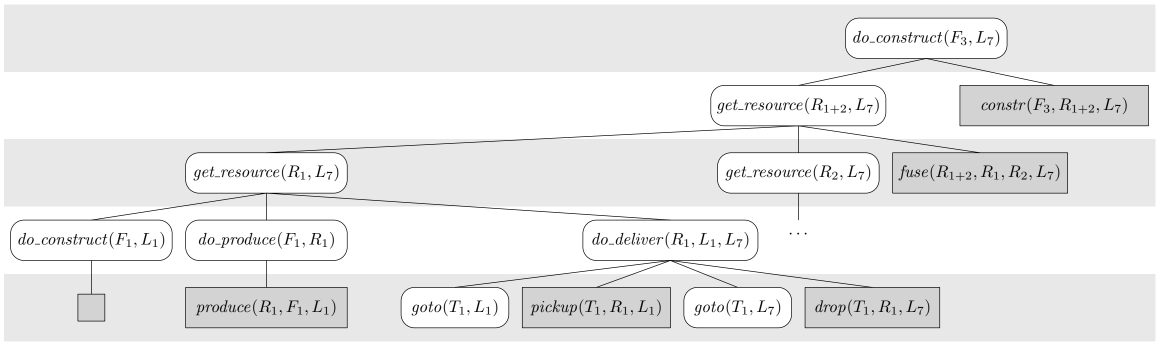


Figure 1: Example instance of the Factories planning domain

2.1.4 Example

Throughout the paper we will use the domain "Factories" by Sönnichsen and Schreiber (2020) as an example for our formalism. In the planning instance illustrated in Fig. 1, two trucks T_1, T_2 can transport resources from one location to another and a factory F_1 can indefinitely produce resource R_1. The objective is to construct factory F_3 at location L_7, for which resources R_1 and R_2 are required. In addition we have a blueprint for factory F_2 which is able to produce resource R_2. However, one unit of resource R_1 is consumed in order to construct F_2 and also for each unit of resource R_2 that F_2 produces.

In a planning domain for this scenario we have predicates such as $at(o, l)$ (Is object o at location l?), $requires(f, r)$ (Does factory f require resource r to be built?) and $free(l)$



$do\_construct(F_3, L_7)$
$get\_resource(R_{1+2}, L_7)$
$constr(F_3, R_{1+2}, L_7)$
$get\_resource(R_1, L_7)$
$get\_resource(R_2, L_7)$
$fuse(R_{1+2}, R_1, R_2, L_7)$
$do\_construct(F_1, L_1)$
$do\_produce(F_1, R_1)$
$do\_deliver(R_1, L_1, L_7)$
...
$produce(R_1, F_1, L_1)$
$goto(T_1, L_1)$
$pickup(T_1, R_1, L_1)$
$goto(T_1, L_7)$
$drop(T_1, R_1, L_7)$

Figure 2: Selected parts of a task network for above Factories planning instance

(Can a factory be built at l?), and operators such as $move(t, l_1, l_2)$ to move truck t in between connected positions, *pickup(t, r, l)* and *drop(t, r, l)* for picking up and dropping resources, *construct*, *produce*, and so on. In addition we define tasks and methods. Fig. 2 depicts a partially expanded task network that may result from such a hierarchical model. Primitive tasks are rectangular and colored gray, compound tasks have rounded corners. For instance, $do\_construct(F_3, L_7)$ is the only initial task of our problem and is achieved through a method $m(f, r, l)$ for $f := F_3$, resource $r := R_{1+2}$ (an object representing the union of R_1 and R_2), and $l := L_7$ with preconditions $\{requires(f, r), free(l), \neg constructed(f)\}$ and subtasks $\langle get\_resource(r, l), construct(f, r, l)\rangle$. In words, we can construct F_3 at L_7 by bringing one unit of R_{1+2} to L_7 and then, atomically, performing the actual construction. Similarly, we get resource R_{1+2} to L_7 by getting both R_1 and R_2 to L_7 and then fusing the two resources; we get R_1 to L_7 by ensuring that F_1 is constructed at L_1, then producing R_1 and finally delivering it to L_7; and so on. In the bottom left, the chosen method for task $do\_construct(F_1, L_1)$ leads to an empty sequence of subtasks: It has a precondition $at(F_1, L_1)$ which ensures that F_1 is already present, so nothing must be done.

When we repeatedly apply such methods and instantiate more and more *layers* of the network, we successively decompose the task network into more and more concrete tasks until only actions remain. If this sequence of actions is executable from left to right beginning with initial state s_I, then we found a solution to our problem.

2.2 SAT Solving

We briefly explain the basics of Satisfiability (SAT) solving. A Boolean variable v can only take two values, `true` and `false`. A Boolean literal is a Boolean variable v or its negation $\neg v$. A clause $c = (l_1 \vee l_2 \vee \ldots \vee l_k)$ is a disjunction (logical OR) of Boolean literals. A propositional formula given in CNF (Conjunctive Normal Form), $F = (c_1 \wedge c_2 \wedge \ldots \wedge c_n)$, is a conjunction (logical AND) of clauses. The propositional satisfiability (SAT) problem is the decision problem of whether there exists a consistent assignment to all Boolean variables in F such that F evaluates to `true`. In practice, we do not consider SAT a pure decision problem but also require a satisfying assignment to be reported if such an assignment exists.

Application-specific problem solvers can profit from SAT solving by encoding their specific problem into propositional logic, executing a SAT solver on the resulting formula, and decoding a satisfying assignment back into the problem domain. Yet, in some applications such as in PSPACE-complete classical automated planning, a theoretical gap between the complexity of the problem to solve and the complexity of SAT (which Cook, 1971 showed to be NP-complete) arises. In this case we cannot hope to find a general polynomially sized encoding of our entire problem into a single formula. Instead, a sequence of SAT formulae of increasing "problem horizon" is encoded until at some point a formula is found to be satisfiable. All relevant SAT-based classical planners have been using this kind of procedure (Kautz & Selman, 1998; Kautz, Selman, & Hoffmann, 2006; Rintanen, 2014), using the maximum number of considered steps as the problem horizon and sometimes allowing for several non-conflicting actions to be executed in a single step.

Gocht and Balyo (2017) introduced incremental SAT solving to SAT-based planning as an improvement. An incremental SAT solver can be queried multiple times with a growing set of clauses and a set of *assumption* literals. The latter are considered axiomatic for a single solving attempt and dropped afterwards. As such, the solver can preserve its knowledge base, reuse conflict clauses from earlier solving iterations, and avoid to repeatedly parse and preprocess similar sets of clauses. Schreiber et al. (2019b) were the first to exploit this technique for hierarchical planning.

3. Related Work

In this chapter we discuss important scientific work related to our approach. We omit a reiteration of the history of the first hierarchical planning approaches leading up to today's established HTN planners because such reviews have been done extensively many times before, for example by Georgievski and Aiello (2015). Instead we focus on the most relevant topics for our means: We discuss grounding of HTN planning problems and approaches to SAT-based HTN planning, and touch on lifted encodings for SAT-based planning.

3.1 Grounding

We now provide some context for grounding HTN planning domains and share some of our own considerations regarding the topic of lifted vs. ground HTN planning.

While many HTN planners operate on the lifted problem description (e.g., Nau et al., 1999; Magnaguagno, Meneguzzi, & de Silva, 2020), ground approaches operate on a simplified and "flattened" representation of the problem. Through grounding, all facts, actions, and reductions which may be relevant to solve the problem are enumerated and compressed into compact data structures. Finding this subset of relevant facts and operations is a difficult problem in itself: For general HTN planning it can even be shown that it is undecidable whether a given action can be part of a plan (Behnke et al., 2020).

Grounding procedures in automated planning generally perform graph-based reachability analyses and only accumulate instantiations which may be reachable during planning (Helmert, 2009). The science of grounding HTN domains is comparably young as a designated area of research with only two notable publications to date (Ramoul et al., 2017; Behnke et al., 2020). Two different analyses that are employed in HTN grounding are (i) top-down reachability analyses where only the operations reachable from the problem's initial tasks over transitive subtask relationships are instantiated, and (ii) bottom-up reachability analyses. The latter perform a state-based reachability analysis on the classical planning problem, for example a delete-relaxed reachability analysis based on planning graphs

(Helmert, 2009), obtain an upper bound on the set of reachable world states and actions, and discard any operations (and potentially their parent operations in the hierarchy) which are never applicable regarding their preconditions.

Naturally, the grade of success of such approaches varies depending on the problem at hand. In many cases grounding is successful in practice and can be beneficial for the subsequent search algorithm (Ramoul et al., 2017). For instance, grounding procedures can be able to prune an operation o because some unavoidable (transitive) child of o is impossible to achieve while lifted planners may get lost in search space without such knowledge.

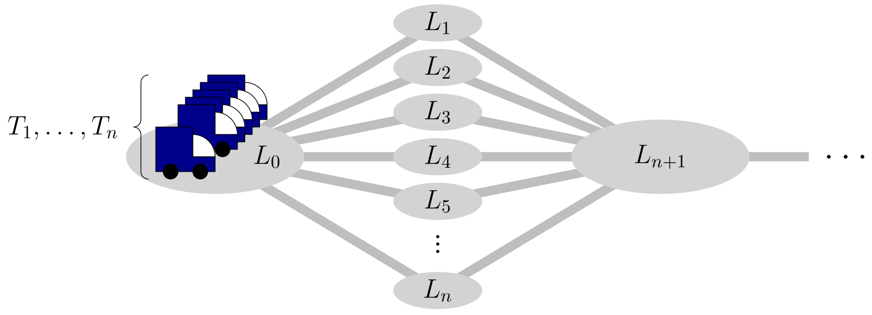


Figure 3: Excerpt of an artificial planning instance from the domain Factories

However, on some planning domains, grounding suffers from intrinsic scaling problems. Fig. 3 illustrates a simple example from the domain Factories. There are n trucks at L_0 and the objective of the problem (i.e., resources to be transported) is located at the far right beyond L_{n+1}. Assume that only a single truck is required for this objective. Any complete grounding procedure is required to instantiate operations for each of the n trucks to traverse any of the locations $L_1, \ldots, L_n$ in order to get to L_{n+1} and achieve the actual task: None of the $\mathcal{O}(n^2)$ operations can be omitted because every operation is reachable and can be part of a plan.[1] This example can be generalized to make the argument that the minimum number of operations produced through grounding can be a high order polynomial in the problem size. The maximum arity of any operation signature provides an upper bound on the polynomial's order. By contrast, a lifted progression search planner can simply make an ad-hoc decision on the truck and the route to take.

3.2 SAT-Based HTN Planning

The first propositional logic encodings for HTN planning problems have been introduced by Mali and Kambhampati (1998). These encodings, however, were restricted to non-recursive (or acyclic) domains. An HTN domain is non-recursive when the graph of all subtask relationships is acyclic. For such domains there is a fixed maximum number of actions any given task can induce, which renders problems relatively easy to solve. Non-recursive HTN planning is unpopular; for instance, in the IPC 2020, 40 TOHTN planning domains but only three non-recursive domains were submitted, and an advertised Acyclic Track (Behnke et al., 2020b) was cancelled due to lack of participants.

1. An interesting direction of research could be to explore incomplete grounding approaches which detect symmetries in the problem and only instantiate some sufficient subset of operations.

After these initial encodings, two decades passed until Behnke et al. (2018) revisited the topic and developed a novel encoding approach named totSAT and designed for TOHTN planning, showing that it outperformed several prior HTN planning algorithms. The authors of totSAT refined this approach to support general HTN planning (Behnke et al., 2019a) and optimal planning regarding the number of actions (Behnke et al., 2019b). The employed encoding is expanded iteratively, but handed to a non-incrementally operating SAT solver at each iteration. All these techniques have been integrated into the PANDA planning system, so we will refer to this branch of approaches as "PANDA-SAT".

Independently and almost simultaneously, Schreiber et al. (2019a) developed a SAT encoding for TOHTN planning problems which exploits incremental SAT solving by simulating a stack machine of tasks and using the number of stack machine transitions as the problem horizon to increase. An enhancement of this approach resulted in the Tree-REX planner (Schreiber et al., 2019b) which was shown to be much more efficient than its precursor while capable to find the shortest possible plan at the first solvable layer.

Although their authors were unaware of each another, Tree-REX and PANDA-SAT converged regarding their encoding structure to some degree: The encodings are iteratively extended not along the length of a final plan (as is the case for encodings for classical planning) but instead along the depth of the hierarchy, essentially leading to an iterative deepening search of the task network at hand. The main difference between the encodings is that Tree-REX encodes states, preconditions, and effects at every layer of the problem while PANDA-SAT only encodes them at the currently final layer, propagating method preconditions as virtual actions instead of encoding them natively.

Furthermore, both approaches rely on a grounding stage prior to the encoding and solving stage. Tree-REX uses the grounding procedure presented by Ramoul et al. (2017) and enhances it by a top-down reachability analysis. The PANDA planning system uses a separate grounding procedure which was recently used as a basis for pandaPIgrounder, the most efficient HTN grounder to date (Behnke et al., 2020); yet, the original grounder of PANDA is significantly slower and of lower quality. A direct comparison of PANDA-SAT and Tree-REX by Schreiber et al. (2019b) suggested that Tree-REX mostly finds plans significantly faster and of comparable or better quality.

The work we present is named Lilotane (Lifted Logic for Task Networks). It is based on the ideas of Tree-REX but omits grounding and uses a substantially reworked encoding. As we successfully avoid a combinatorial blowup of the problem input which is intrinsic to all previous SAT-based approaches, we improve on state-of-the-art HTN planning. Lilotane comes with two limitations which we elaborate on in the following.

First, our approach is limited to TOHTN planning. As a matter of fact, the total ordering of subtasks and consequently the exact knowledge of each operation's position in the hierarchy enables us to perform significant optimizations and greatly simplifies the resulting encoding. We argue that totally ordered HTN models are highly popular and, where possible, desirable over partial orderings: Imposing a total order on the task network reduces the decisions a planner needs to make and helps to achieve a plan that is well-structured and simple to interpret for a human. The recent International Planning Competition suggests that TOHTN planning is highly popular, as the Total Order track had twice the number of competitors of the Partial Order track. Furthermore, 40 totally ordered planning domains from nine different groups but only eleven partially ordered planning domains from two groups were submitted (Behnke et al., 2020b). That being said, we do expect that our lifted encoding approach can be transferred to general HTN planning in the future.

Secondly, while we do consider quality-awareness in our approach, Lilotane is not generally optimal with respect to the number of actions in a plan. We believe that the problem-dependent bound by Behnke et al. (2019b) for how deeply the hierarchy must be explored until an optimal plan can be found is expensive to obtain with no access to the ground problem. As such, our approach does not knowingly find optimal plans but offers tools to indefinitely improve the plan, guaranteeing that eventually the optimal plan will be output if sufficient time and memory are available. Our evaluations suggest that our quality-aware approach is highly practical while far more efficient than existing optimal approaches.

3.3 Lifted SAT Encodings for Automated Planning

The idea of reducing the number of encoded actions in SAT-based automated planning ranges back to Kautz, Selman, et al. (1992) who proposed to "factorize" actions by splitting each signature into several shorter signatures. Based on this idea, various factorized encodings have been established which encode arguments explicitly and can therefore be considered lifted SAT encodings (Ernst, Millstein, & Weld, 1997). Executing multiple non-interfering actions at a single step can be problematic for these encodings and requires non-trivial adjustments (Robinson, Gretton, Pham, & Sattar, 2009; Williams, 2020). To further reduce grounding overhead and encoding size, Cashmore, Fox, and Giunchiglia (2013) presented an approach of Quantified Boolean Formula (QBF) based planning. Most recently, Bonet and Geffner (2020) described a fully lifted "meta-encoding" of planning domains in order to infer a first-order symbolic representation from the structure of state space.

Our hierarchical encoding shares some of the issues of previous lifted encodings such as more complex frame axioms (Ernst et al., 1997). Yet, due to the rigid layout of operations induced by the given problem hierarchy, we do not consider parallel action execution and are faced with new challenges and opportunities, as we elaborate in the following chapters.

4. Planning Approach

In this chapter we present our overall planning algorithm.

In order to solve a TOHTN planning problem, we partition the hierarchy into a sequence of *hierarchical layers* $\{L_0, L_1, \ldots\}$. The first layer L_0 only contains the initial reduction r_0. For $i > 0$, layer L_i contains all operations which match a subtask of some operation at layer L_{i-1}. Intuitively, the index or *depth* of a layer can be seen as the degree of refinement of the planning problem. Furthermore, each layer is subdivided into *positions* to account for the total ordering of all operations. Scanning a layer from left to right chronologically traverses possible plans at the respective degree of refinement.

As illustrated in Algorithm 1, we begin to construct the first two hierarchical layers L_0, L_1 of the problem, encode them into propositional logic, and then perform a first solving attempt of the formula in line 24 under the logical assumption that all operations chosen at the currently final layer are primitive. As long as the SAT solver reports unsatisfiability, we construct the next layer, extend our formula by that layer's encoding, and attempt to solve it in the same way. When satisfiability is reported, we either directly decode and return a plan from the satisfying assignment or we employ a plan improvement procedure

Algorithm 1: Lilotane Planning Procedure

```
   Input: $\Pi = (D, s_I, T)$
   Result: Plan $\pi$
1  Preprocess $\Pi$;            // parsing, simplification, setup of data structures
2  $H := \langle\rangle$;
3  $L_0 := \langle$ CreateInitialPosition$(T, s_I)$ $\rangle$;
4  $H := H \circ \langle L_0 \rangle$;
5  $F := \emptyset$;                                        // global relevant facts
6  for $l = 0, 1, \dots$ do
       // instantiate new layer
7      $L_{l+1} := \langle\rangle$;
8      $S^0_{l+1} := (s_I, \emptyset)$;                          // reachable facts at this layer
9      $x' := 0$;
10     for $x = 0, \dots, |L_l| - 1$ do
11         $e_{l,x} := \max\{1, \max\{|subtasks(r)| \mid r \in P_{l,x}\}\}$;     // max. expansion size
12         for $z = 0, \dots, e_{l,x} - 1$ do
13             $P_{l+1,x'} :=$ Instantiate$(P_{l,x}, z, S^{x'}_{l+1})$;
14             $L_{l+1} := L_{l+1} \circ \langle P_{l+1,x'} \rangle$;
15             $S^{x'+1}_{l+1} := S^{x'}_{l+1} \cup \mathit{possibleFactChanges}(P_{l+1,x'})$;
16             $F := F \cup \mathit{relevantFacts}(P_{l+1,x'})$;
17             $x' := x' + 1$;
18         end
19     end
       // encode new layer
20     for $x' = 0, \dots, |L_{l+1}| - 1$ do
21         Encode$(P_{l+1,x'}, F)$;
22     end
       // finalize layer, attempt to solve
23     $H := H \circ L_l$;
24     $\mathit{result}$ := Solve$(H)$;
25     if $\mathit{result}$ is SAT then
           // optional anytime plan length optimization
26         while further plan improvement is desired do
27             OptimizeCurrentPlan$(H)$;
28             if plan is depth-optimal break;
29         end
30         return Decode$(H, \mathit{result})$;
31     end
32 end
```

as long as the plan is improvable and the user permits it (see Chapter 6). This general procedure has been introduced by Schreiber et al. (2019b) and is conceptually based on the approach of (classical) planning via incremental SAT solving by Gocht and Balyo (2017) where assumptions are used to enforce a completed plan at the current problem horizon.

4.1 Instantiation

Let us now take a closer look at how hierarchical layers are defined and constructed.

Let $P_{l,x}$ denote the x-th position of the l-th layer L_l. The initial layer L_0 is obtained from the problem definition: The only position $P_{0,0}$ of L_0 only contains the initial reduction r_0. Given a layer $L_l = \langle P_{l,0}, P_{l,1}, \dots, P_{l,x}, \dots \rangle$, we compute the possible children of the operations at each $P_{l,x}$ and append respective new positions to the subsequent layer L_{l+1}.

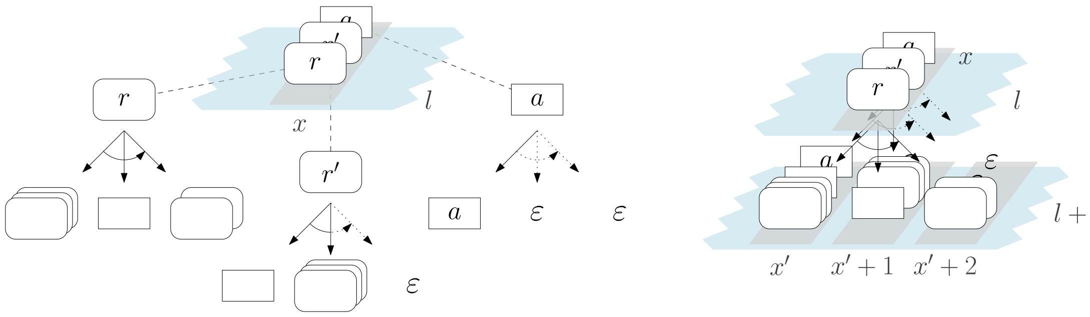


Figure 4: Construction of hierarchical layers. Rectangles are actions, rounded rectangles are reductions. Stacked operations denote a set of alternatives ("or"). Left: $P_{l,x}$ contains several operations each of whose expansions is normalized to the maximum expansion size of three. Right: The child operations are aggregated into three new positions.

Consider the situation illustrated in Fig. 4: We are in the process of instantiating layer $l+1$. Positions $P_{l,0}, \dots, P_{l,x-1}$ of layer L_l have already been processed and resulted in new positions $P_{l+1,0}, \dots, P_{l+1,x'-1}$ at layer L_{l+1} for some x'. Consequently, the children of $P_{l,x}$ begin at index x'. We refer to this index x' as $s_l(x)$, the first *successor position* (or child position) of $P_{l,x}$. In Fig. 4, $P_{l,x}$ features three possible operations: reductions r and r' and action a. Assume that by their definition, r has three subtasks and r' has two subtasks.

Each subtask of an operation may be achieved by any of several operations, depending on whether an operator matches the subtask or, otherwise, how many distinct methods match the subtask. For the final plan, only a single operation from each position will be chosen. We denote the set of operations which can result from the z-th subtask of an operation o as $\mathit{children}(o, z)$ for $z \geq 0$. As we know for every operation how large its induced sequence of children will be (1 for an action and $|\mathit{subtasks}(r)|$ for a reduction r), we can easily compute $e_{l,x}$ as the *maximum expansion size* of any operation at $P_{l,x}$.

As a consequence, we know that position $P_{l,x}$ induces $e_{l,x}$ child positions beginning from $s_l(x)$. To keep each child position well-defined for each parent operation, we define $\mathit{children}(o, z) := \{\varepsilon\}$ if o is a reduction and $z \geq |\mathit{subtasks}(o)|$ if o is an action and $z \geq 1$. We define ε as a special action with $\mathit{pre}(\varepsilon) = \mathit{eff}(\varepsilon) = \emptyset$ which is treated as a normal action in the encoding but omitted in the final plan. As such, in Fig. 4 we have constructed three new positions each of which contains the union of all children at the respective offset.

4.1.1 Example

We now illustrate this approach with the Factories domain introduced in Chapter 2.1.4. For the sake of simplicity, we use a very simple planning task as pictured in Fig. 5: The goal is to construct factory F_2 at location C which requires one unit of resource R. This resource can be produced without any prerequisites either by F_0 at A or by F_1 at B and must be transported to C by truck T_1 or T_2.

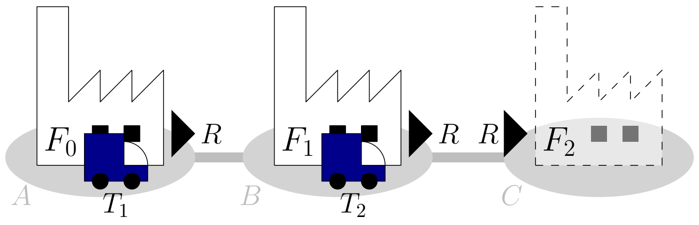


Figure 5: Simple planning example from the Factories domain

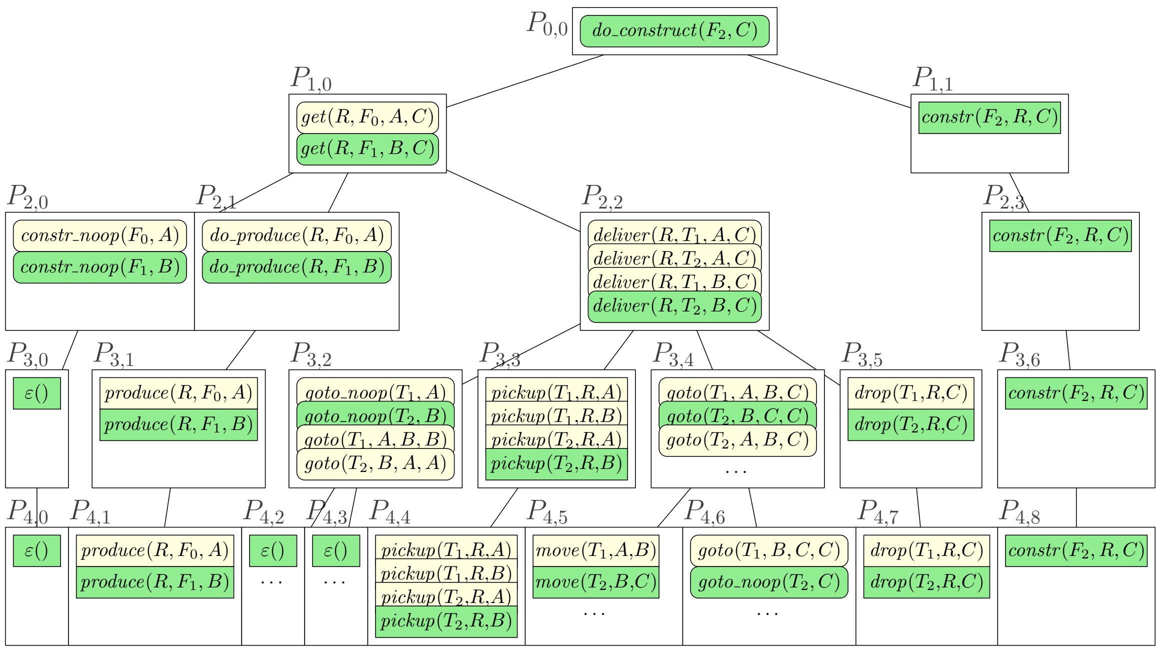

$P_{0,0}$ $do\_construct(F_2, C)$
$P_{1,0}$ $get(R, F_0, A, C)$ $get(R, F_1, B, C)$
$P_{1,1}$ $constr(F_2, R, C)$
$P_{2,0}$ $constr\_noop(F_0, A)$ $constr\_noop(F_1, B)$
$P_{2,1}$ $do\_produce(R, F_0, A)$ $do\_produce(R, F_1, B)$
$P_{2,2}$ $deliver(R, T_1, A, C)$ $deliver(R, T_2, A, C)$ $deliver(R, T_1, B, C)$ $deliver(R, T_2, B, C)$
$P_{2,3}$ $constr(F_2, R, C)$
$P_{3,0}$ $\varepsilon()$
$P_{3,1}$ $produce(R, F_0, A)$ $produce(R, F_1, B)$
$P_{3,2}$ $goto\_noop(T_1, A)$ $goto\_noop(T_2, B)$ $goto(T_1, A, B, B)$ $goto(T_2, B, A, A)$
$P_{3,3}$ $pickup(T_1,R,A)$ $pickup(T_1,R,B)$ $pickup(T_2,R,A)$ $pickup(T_2,R,B)$
$P_{3,4}$ $goto(T_1, A, B, C)$ $goto(T_2, B, C, C)$ $goto(T_2, A, B, C)$ …
$P_{3,5}$ $drop(T_1,R,C)$ $drop(T_2,R,C)$
$P_{3,6}$ $constr(F_2, R, C)$
$P_{4,0}$ $\varepsilon()$
$P_{4,1}$ $produce(R, F_0, A)$ $produce(R, F_1, B)$
$P_{4,2}$ $\varepsilon()$ …
$P_{4,3}$ $\varepsilon()$ …
$P_{4,4}$ $pickup(T_1,R,A)$ $pickup(T_1,R,B)$ $pickup(T_2,R,A)$ $pickup(T_2,R,B)$
$P_{4,5}$ $move(T_1,A,B)$ $move(T_2,B,C)$ …
$P_{4,6}$ $goto(T_1, B, C, C)$ $goto\_noop(T_2, C)$ …
$P_{4,7}$ $drop(T_1,R,C)$ $drop(T_2,R,C)$
$P_{4,8}$ $constr(F_2, R, C)$

Figure 6: Five hierarchical layers of the Factories instance from Fig. 5. Each white rectangle is a position. Black lines connect a position to its child positions. Actions are displayed as rectangles, reductions are displayed as rounded rectangles. A set of operations leading to a valid plan is colored green.

Fig. 6 illustrates the layers instantiated by Lilotane for solving this planning problem. The tree-like structure is similar to the illustration of a task network in Fig. 2. However, note two important differences: First, while the nodes in Fig. 2 represent compound and primitive tasks, the nodes in the above tree feature reductions and actions instead. Secondly, the task network in Fig. 2 represents one particular (partial) expansion of a problem whereas above structure represents all possible expansions (abbreviated where necessary).

Layers L_0 through L_4 are displayed from top to bottom. L_0 only contains a single initial reduction *do_construct*(F_2, C) at position $P_{0,0}$[2]. This reduction induces two child positions at layer L_1, $P_{1,0}$ and $P_{1,1}$, with all possible operations which match the first and second

2. For the sake of simplicity, the illustration deviates from our definition where the initial reduction is a virtual operation which features the actual initial reduction(s) as its children (see Chapter 2.1.3).

subtask of *do_construct*(F_2, C) respectively. Further down, we omitted some operations at positions marked with "…".

An "almost complete" hierarchical plan for the problem at hand is colored green. This plan involves producing R at factory F_1 and using truck T_2 to transport it in a single move action to location C where R is used to construct F_2. The plan is not entirely finished: Our SAT encoding up to L_4 will not be satisfiable because operation *goto_noop*(T_2, C) at $P_{4,6}$ is not primitive and still needs to be concretized. However, it is clear that this reduction at $P_{4,6}$ will decompose into an ε-action at the next layer L_5. Then all chosen operations at the final layer are primitive and the highlighted plan can be found and reported.

4.1.2 Transformation of Reductions into Actions

The inability of our basic approach to find a plan at L_4 in the above example can be corrected by treating certain reductions as actions, as introduced by Schreiber et al. (2019b).

First, all reductions r with $\mathit{subtasks}(r) = \langle\rangle$ are treated as actions throughout the planning procedure. In Fig. 6, our plan contains such an empty reduction as its only remaining non-primitive operation at position $P_{4,6}$. If we treat this operation as an action, we can find a plan at layer L_4 and do not need to construct and encode another layer. If an extracted plan contains such an "action" at the final layer, we omit it in the output of the primitive plan but keep the corresponding reduction as a part of the hierarchical solution.

Secondly, given a reduction r with $\mathit{subtasks}(r) = \langle a \rangle$ where a is a primitive task, we replace r with a new action a' with $\mathit{pre}(a') := \mathit{pre}(r) \cup \mathit{pre}(a)$ and $\mathit{eff}(a') := \mathit{eff}(a)$. In our example in Fig. 6, reduction $r = \mathit{do\_produce}(R, \phi, \lambda_1)$ at position $P_{2,1}$ would be replaced in this manner. When a plan contains such a surrogate a', we replace it with r and its child a.

In some cases, these simplifications shortcut necessary expansions and hence decrease the depth required to find a plan by one, which can make a notable difference if the size of layers grows exponentially in the depth of a problem.

4.1.3 Pseudo-Constants

Unlike previous SAT-based approaches which perform a complete grounding, Lilotane lazily instantiates each operation from a parent's definition just when needed. In addition, as we explain next, this instantiation is done minimalistically: We do not fully instantiate child operations with free arguments but instead keep them lifted.

Consider position $P_{1,0}$ in Fig. 6. On an intuitive level, this position features the production and transportation of resource R to location C. Both F_0 and F_1 are able to produce R. In addition, at position $P_{2,2}$ truck T_1 or T_2 must be chosen to transport R. The combination of these two decisions leads to four different deliver operations at position $P_{2,2}$. More generally, the full instantiation of such argument combinations can lead to a concerning increase of operations at each position (see Chapter 3.1).

To alleviate this issue, we replace free arguments in methods with new symbols instead of fully instantiating them. Let ϕ be the picked factory, λ the location of ϕ, and θ the picked truck. Then we can express both operations at $P_{1,0}$ as $\mathit{get}(R, \phi, \lambda, C)$ and all four operations at $P_{2,2}$ as $\mathit{deliver}(R, \theta, \lambda, C)$. We call the new argument symbols *pseudo-constants*: At a later point, ϕ must be substituted with either F_0 or F_1, θ must be substituted with either

T_1 or T_2, and so on. With our encoding presented in Chapter 5 we will let a SAT solver decide which of the possible substitutions to apply for each pseudo-constant.

In general, for each free argument a_i of operation o, we initialize the *effective domain*, $dom(\alpha_i)$, of pseudo-constant α_i with the argument type τ_i. We then remove any constants from $dom(\alpha_i)$ for which some precondition of o becomes impossible at the current position. (We explain in Chapter 4.2 how this can be checked.) Only if $|dom(\alpha_i)| = 1$ we do not introduce a pseudo-constant but rather directly substitute a_i with the only valid constant.

Essentially, instead of introducing $\prod_i |dom(\alpha_i)|$ operations, we introduce one lifted operation with $\sum_i |dom(\alpha_i)|$ different pseudo-constant values to handle. However, we will still need to enumerate and encode all (ground) preconditions and effects that can result from such a lifted operation. This concept is not unlike the lifted successor generation recently proposed by Corrêa, Pommerening, Helmert, and Frances (2020) for the case of classical planning, where actions are kept lifted while states are maintained in a ground representation. Just like their approach, our idea is built upon the assumption that there are fewer ground facts than there are reachable ground operations in the problem. In Chapter 7, we show by benchmark analyses and experiments that this assumption is reasonable in general.

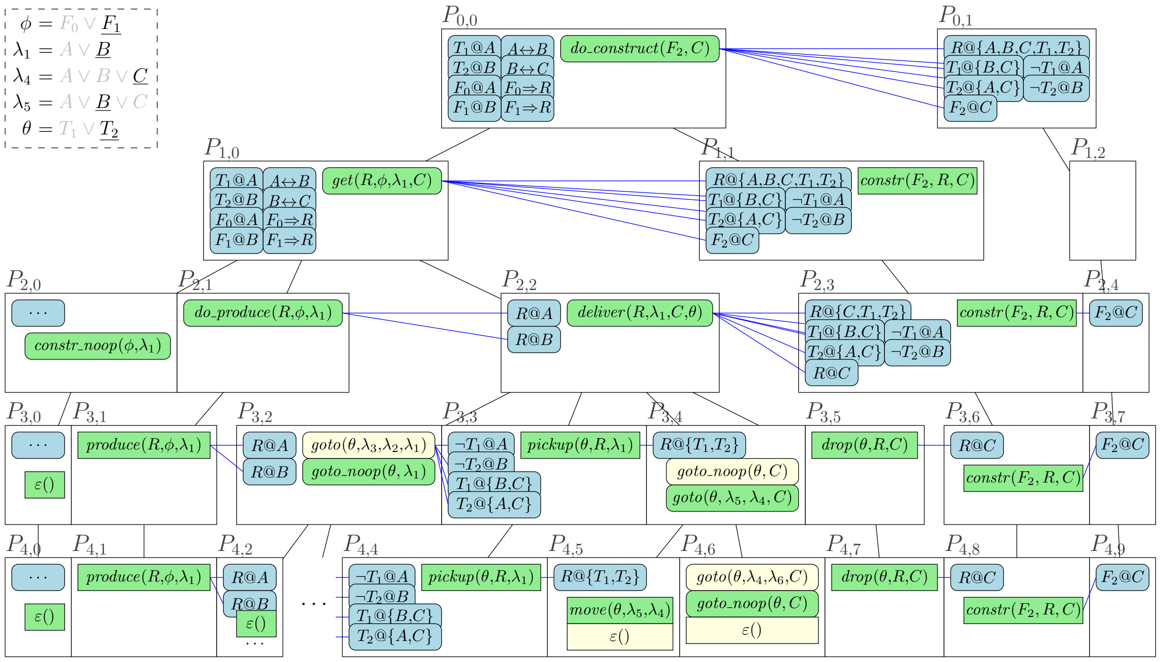


Figure 7: Hierarchical layers as in Fig. 6 but with pseudo-constants and reachable facts. Facts occurring for the first time in a layer are displayed as blue boxes with rounded corners. Blue lines (horizontal) connect operations with any “new” facts they may cause.

Fig. 7 applies the example from Fig. 6 to the use of pseudo-constants. This figure serves as a running example throughout the following sections and hence also introduces certain fact collections at each layer which we will discuss in Chapter 4.2. For now, let us concentrate on the operations that occur within the layers. At $P_{1,0}$ we introduce pseudo-constant ϕ for the factory producing R and at $P_{2,2}$ we introduce pseudo-constant θ for the truck to transport R to C. At positions $P_{1,0}, P_{3,2}, \ldots$ we introduce various pseudo-constants

$\lambda_1, \lambda_2, \ldots$ to represent particular locations. It can be seen that pseudo-constants, just like normal arguments, are propagated down to the (transitive) children of the operation they originated from. In the top left corner the chosen substitution for each relevant pseudo-constant is displayed: This information is essential to decode a valid plan from the chosen operations. A pseudo-constant is relevant if and only if the operation it originates from is part of the (hierarchical) solution.

4.2 Reachability Analysis for Facts and Operations

In order to minimize the number of added operations, we perform a reachability analysis at each layer where we take into account the possible world states at each position.

For each operation o we instantiate, we define $\mathit{pfc}(o)$ as the *possible fact changes* of o – an over-approximation of all positive and negated facts which may be caused by o or by any transitive child of o. With the use of this function, we construct and successively update S_{l+1}, which represents all positive and all negative facts which may have been effected so far by earlier operations, at each layer L_{l+1}. We define the x-th update of S_{l+1} as follows:

$$S^x_{l+1} := (+S^x_{l+1}, -S^x_{l+1}) := \Big(\bigcup_{i=0}^{x-1} \bigcup_{o \in P_{l+1,i}} \mathit{pfc}(o)^+,\ \bigcup_{i=0}^{x-1} \bigcup_{o \in P_{l+1,i}} \mathit{pfc}(o)^-\Big).$$

As such, $+S^x_{l+1}$ ($-S^x_{l+1}$) consists of all positive (negative) facts that may be produced by some operation up to the x-th position.

For any position $P_{l,x}$, we define a positive fact f to be *reachable* at position $P_{l,x}$ if $f \in s_I \cup +S^x_l$ holds. Similarly, a negated fact $\neg f$ is *reachable* at $P_{l,x}$ if $f \in -S_l$ or $f \notin s_I \cup +S^x_l$ holds. If a fact f is not reachable at $P_{l,x}$, then we call f *invariantly false* at $P_{l,x}$. If for a fact f its negation $\neg f$ is not reachable at $P_{l,x}$, then we call f *invariantly true* at $P_{l,x}$. Note that if some fact f is invariantly true (false) at $P_{l,x}$, then $\neg f$ is invariantly false (true) at $P_{l,x}$. We use S_l and these definitions to check the invariance of preconditions in the "Instantiate" procedure in line 13 of Alg. 1 to prune impossible operations.

In the following, we first illustrate our reachability analysis in an example, then describe how we compute S_l and $\mathit{pfc}(\cdot)$, and establish a proof of correctness for our analysis.

We visualized a number of facts in Fig. 7. In each position $P_{l,0}$ all facts in the initial state are displayed (or abbreviated, for $l \geq 2$). In each position $P_{l,x}$ for $x > 0$ all facts are displayed which were invariantly false at $P_{l,0}, \ldots, P_{l,x-1}$ and which become reachable at $P_{l,x}$. Facts have been abbreviated – for example, $F_0@A$ to denote that factory F_0 is present at location A, $A{\leftrightarrow}B$ to denote that there is a path between A and B, and $F_0 \Rightarrow R$ to denote that factory F_0 produces resource R. Some redundant facts are omitted: Each fact $(\neg)\phi@\lambda$ for factory ϕ and location λ also implies the fact $(\neg)\mathit{constructed}(\phi)$.

Blue horizontal lines indicate which facts are caused by which operation. In the case of $\mathit{get}(R, \phi, \lambda_1, C)$ at $P_{1,0}$, we can see that this operation may cause resource R to be located anywhere (due to *do_produce* and *deliver*) and both trucks T_1, T_2 to be located anywhere (because the *goto* subprocedure in *deliver* is recursive and may lead a truck to any location). These fact changes include the negative facts $\neg T_1@A$ and $\neg T_2@B$ as they were unreachable before (because $T_1@A$ and $T_2@B$ hold initially). According to $\mathit{pfc}(\cdot)$, the operation may even cause F_2 to be constructed – this false positive will be discussed further in Chapter 4.2.3.

Reduction $r = get(R, \phi, \lambda_1, C)$ at $P_{1,0}$ encompasses the construction of some factory ϕ at location λ_1 as its first subtask. Without incorporating any knowledge from preconditions, we would assume that $dom(\phi) = \{F_0, F_1, F_2\}$ and $dom(\lambda_1) = \{A, B, C\}$. However, r has a precondition $\phi \Rightarrow R$. According to S_1^0, this precondition is invariantly false for $\phi = F_2$. For this reason, ϕ is initialized with a smaller domain, $dom(\phi) = \{F_0, F_1\}$. Note that reducing the domain of a pseudo-constant is a form of pruning in our approach as it cuts the number of ground operations that are represented by the enclosing lifted operation.

Nevertheless it can also occur that an entire operation is pruned. The first subtask of r, namely the construction of ϕ at λ_1, is achieved at position $P_{2,0}$. This task can be matched by $constr\_noop(\phi, \lambda_1)$ (with a precondition $\phi@\lambda_1$) or $do\_construct(\phi, \lambda_1)$ (with a precondition $\neg constructed(\phi)$). As both F_0 and F_1 are already constructed according to S_2^0, we know that the precondition of $do\_construct(\phi, \lambda_1)$ is invariantly false for *all* substitutions of ϕ. As such, this operation is not included in position $P_{2,0}$.

4.2.1 Computation

We now describe how to efficiently compute our reachability analysis. S_{l+1}^0 is initialized with the initial state s_I and no negative facts in line 8 of Alg. 1: It would be redundant and expensive to add all syntactically possible facts to the set S_{l+1}^0 which are not in s_I. We can implicitly consider them invariantly false according to the closed-world assumption.

For $x' \geq 0$, we construct $S_{l+1}^{x'+1}$ via $possibleFactChanges(P_{l+1,x'})$ in line 15 of Alg. 1. In this procedure we collect all possible fact changes of $P_{l+1,x'}$, $PFC_{l+1,x'} := \bigcup_{o \in P_{l+1,x'}} pfc(o)$, and then update $S_{l+1}^{x'+1} := (+S_{l+1}^{x'} \cup PFC_{l+1,x'}^{+}, -S_{l+1}^{x'} \cup PFC_{l+1,x'}^{-})$. For any operation o, we compute $pfc(o)$ as follows:

- If o is primitive, then $pfc(o) = g(eff(o))$. Thereby $g(\cdot)$ is the *ground hull* of a set of facts: Any lifted fact in $eff(o)$ is fully instantiated into a set of ground facts.
- Otherwise, $pfc(o) = g(\bigcup_{z=0}^{e_{l,x}} \bigcup_{o' \in children(o,z)} pfc(o'))$, i.e., we recursively compute the possible fact changes of each possible child of o and ground the resulting facts.

To avoid infinite recursion, we remember each visited method together with its subset of ground arguments and break recursion when an equivalent signature occurs again. We employ memoization to compute the fact changes only once per operator and method.

4.2.2 Correctness

To make sure that our reachability analysis works as intended, we show a central (semi-formal) correctness property of S_l:

Theorem 1. *For $l \geq 0$ and $x \geq 0$, let $O_l := \langle o_0, \ldots, o_x \rangle$ be a sequence of operations where each o_i is a possible operation at position $P_{l,i}$ of some problem Π. Expand each $o_i \in O_l$ into some sequence $\mathcal{O}_i$ of actions such that $\mathcal{O} := \mathcal{O}_0 \circ \ldots \circ \mathcal{O}_x$ is executable from s_I.*
(1) If fact f holds after the execution of $\mathcal{O}$, then f is reachable at $P_{l,x+1}$ according to S_l^{x+1}.
(2) Similarly, if f does not hold after executing $\mathcal{O}$, then $\neg f$ is reachable at $P_{l,x+1}$.

The proof of this theorem is given in Appendix A. As a direct consequence, if some fact $(\neg)f$ is not reachable at some position, then there is no way how any execution of the operations before this position could lead to f being true (false). For this reason, we can safely prune any operations for which a precondition is not reachable according to S_l, as in this case the precondition is definitely impossible.

4.2.3 Relevant Facts and Retroactive Pruning

The procedure we just described helps to discard irrelevant operations similarly to how grounders analyze the problem in order to avoid instantiating unreachable operations. However, our analysis is done repeatedly on a successively refined sequence of operations. As such, it generally gains knowledge the deeper we explore the problem hierarchy: The more concrete our operations become, the more exact $pfc(\cdot)$ becomes.

For instance, in Fig. 7, $F_2@C$ is invariantly false at $P_{1,0}$ but reachable at $P_{1,1}$: As $pfc(\cdot)$ ignores the preconditions of children, it finds that operation $get(R, \phi, \lambda_1, C)$ has a possible child *do_construct*$(\cdot)$ which might achieve the construction of F_2 (even if we know that this cannot be the case). One layer later, this over-approximation is rectified: There is no operation before $P_{2,3}$ which can cause the construction of F_2.

One consequence of this stepwise refinement is that some facts which in practice are irrelevant for the planning task may be added to S_l. As we do not wish to encode facts that are not relevant at this layer (yet), we maintain a set of *relevant facts* F which grows monotonically with each further layer. A fact is *relevant* and consequently added to F if it occurs as a (positive or negative) action effect or as a (positive or negative) precondition of some operation added to the current layer. Then, when we encode the layer into propositional logic, we can check for each fact whether it is relevant and should be encoded. Otherwise, we know that the fact cannot be actively involved in the planning task so far and we omit it from our encoding.

Another consequence of our technique is that we may encode an operation o at layer L_l and then notice at some layer L_{l+k} that a precondition of some necessary transitive child o' of o is not reachable. In such a case, o retroactively turns out to be impossible to achieve. We mentioned in Chapter 3.1 how ground approaches are often able to prune such operations *a priori* while our approach does not have the necessary knowledge to do so.

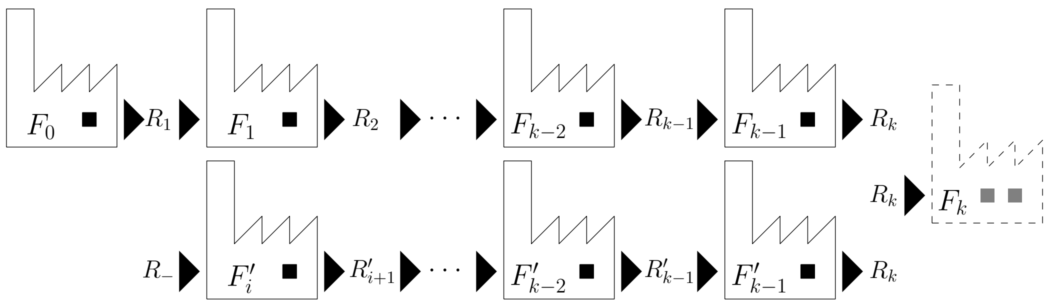


Figure 8: Dependency chains in an instance of the Factories domain

In Fig. 8 we adapted our Factories example to illustrate this issue. The initial reduction is *do_construct*(F_k, L) for some location L. Resource R_k is required to construct F_k and can be produced either by F_{k-1} or by F'_{k-1}. In both cases a certain production chain must be followed to meet the requirements for producing R_k. However, the production chain enabling F'_{k-1} is broken: For some $0 \leq i < k$, F'_i requires an unobtainable resource R_- in order to produce R'_{i+1}.

In such a planning problem, the task of acquiring R_k may – allegedly – involve either $\mathit{do\_construct}(F_{k-1}, \lambda)$ or $\mathit{do\_construct}(F'_{k-1}, \lambda')$ (for some λ, λ'). Both reductions cause a large subtree of operations. However, $\mathcal{O}(k-i)$ layers further below, it turns out that there is no valid reduction to acquire R_-: Operation $\mathit{do\_construct}(F'_i, \cdot)$ has no valid children for its first subtask and is therefore impossible to achieve. As a result, parent operation $\mathit{get}(R'_{i+1}, F'_i, \cdot, \cdot)$ has no valid children matching its first subtask any more and becomes unachievable, causing $\mathit{do\_construct}(F'_{i+1}, \cdot)$ to become unachievable, and so on. As such, the entire subtree of operations rooted at reduction $o := \mathit{do\_construct}(F'_{k-1}, \lambda')$ turns out to be irrelevant. The larger $k-i$, the more unnecessary work we already did; and the larger i, the more irrelevant positions and operations may still be instantiated.

We cannot undo the work we already did, and we cannot remove clauses which were already added to our incremental encoding. We can, however, remove the subtree of operations rooted at o from our layers such that no more irrelevant children are induced at a subsequent layer. Beginning from an operation found to be impossible, we traverse the hierarchy upwards and recursively mark each parent which has no valid child left for some of its subtasks. Then, beginning from the upmost marked operation(s), we traverse the hierarchy downwards and remove all operations which have been marked or which have no valid parent any more. We also add a unit clause to the encoding which forbids o to be used and hence essentially switch off the logical constraints associated with o.

Our retroactive pruning technique benefits the planning process on a situational basis: While many planning domains do not feature any retroactively pruneable operations, on some domains we observed thousands of operations being pruned (see Table 6, Appendix C).

4.3 Precondition Inference

The effectiveness of the state-based reachability analysis we discussed in the previous section crucially depends on expressive method preconditions: The pruning of impossible operations can have considerable effects on run times and encoding volume if pruned operations would otherwise make up for a large number of transitive children. Yet, some domains do not or very sparingly contain method preconditions. Our intention was to make Lilotane robust towards such "ill-conditioned" domains missing expressive preconditions such that the planner may still perform effective pruning.

We use a simple graph traversal of (lifted) subtask relationships in order to raise certain action preconditions up the hierarchy and hence infer method preconditions which are implied by the problem logic. Interestingly, Magnaguagno et al. (2020) have implemented a very similar technique for their TOHTN planner HyperTensioN which competed with Lilotane in the IPC 2020, although their planning approach itself is very different from ours. This indicates that efficient TOHTN planning algorithms oftentimes rely on expressive method preconditions and ideally infer them where missing.

Our basic idea is that each precondition that is common to all operations in $\mathit{children}(r, z)$ for some z is forcibly a precondition of r itself, except if it may be caused by a child of r at an earlier offset $z' < z$. For instance, in our Factories example, method $m := \mathit{do\_produce}(r, f, l)$ to produce resource r via factory f at location l has action $a := \mathit{produce}(r, f, l)$ as its only possible child at offset $z = 0$. As a result, $pre(a)$ can be added to $pre(m)$. By contrast, method $m' := deliver(r, l_1, l_2, t)$ to deliver resource r from l_1 to l_2 via truck t has action $a' := pickup(t, r, l_1)$ as its only child at offset $z = 1$ with a precondition $t@l_1$. However, as there are child operations of m' at offset $z' = 0$ which can cause $t@l_1$, this precondition cannot be added to $pre(m')$.

For each operation o, we write $pre^*(o)$ for the union of $pre(o)$ with all new preconditions that are found for o. If o is an action, $pre^*(o) := pre(o)$. Otherwise, o is a method with $k \geq 0$ subtasks and we compute $pre^*(o)$ recursively as follows:

We initialize $pre^*(o) := pre(o)$ and $E := \emptyset$. For each offset $z = 0, \dots, k-1$, we compute the set of preconditions $I := \bigcap_{o' \in children(o,z)} pre^*(o')$ common to all operations at offset z. We add the preconditions not yet achieved by an earlier subtask, $I \setminus E$, to $pre^*(o)$, and then we add the possible fact changes of each o' to E. If we recursively encounter method m as a child while computing $pre^*(m)$, we approximate $pre^*(m) := pre(m)$ for that child to avoid infinite recursion. As in the computation of possible fact changes (Chapter 4.2) we compute pre^* only once for each method.

In practice we have observed that this algorithm finds various preconditions in domains where they are originally missing (see Table 6, Appendix C). In our Factories example, the mentioned procedure will propagate precondition $f \Rightarrow r$ from action $produce(r, f, l)$ to method $do\_produce(r, f, l)$ and, consequently, to method $get(r, f, l_1, l_2)$ if these method preconditions are not specified explicitly.

4.4 Shared Pseudo-Constants and Dominated Operations

Our approach is based on the idea that every operation at some position will induce one new pseudo-constant κ for each of its free arguments, with domain $dom(\kappa)$ as described in Chapter 4.1.3, and that child operations at subsequent layers naturally inherit some of these pseudo-constants in addition to introducing new pseudo-constants themselves. There are scenarios where this leads to undesired behavior.

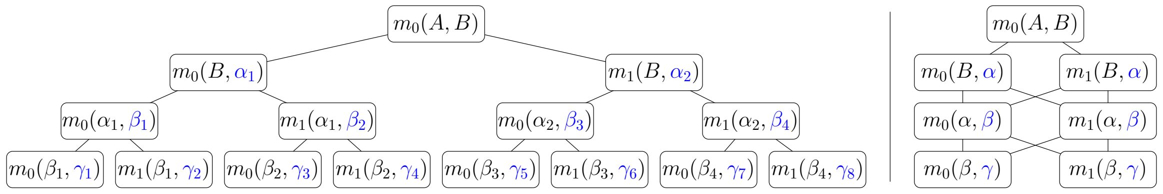


Figure 9: Naive (left) and true (right) recursive children of m_0. New pseudo-constants are colored blue. An edge from u to v denotes that v matches a subtask of u.

Consider methods $m_0(a, b)$ and $m_1(a, b)$ with $subtasks(m_0(a, b)) = subtasks(m_1(a, b)) = \langle t(b) \rangle$ where task $t(b)$ can be achieved both by $m_0(b, a)$ and $m_1(b, a)$. The transitive children of $m_0(a, b)$ will blow up in our approach as depicted on the left in Fig. 9: At each further layer, each operation branches into two new possible operations, and all produced operations syntactically differ due to the unique names of pseudo-constants for each new operation. As such, the number of operations grows exponentially in the explored depth whereas the true structure of m_0 only results in two distinct operations as depicted on the right in Fig. 9.

As only a single operation can be active at each position, we suggest to share the same new pseudo-constant among multiple operations: When two or more operations have a free argument which would lead to exactly the same effective domain of a pseudo-constant, we introduce the same pseudo-constant for these arguments. With this change, the recursive children of $m_0(a, b)$ are computed properly in our minimalistic example. Yet, it does not address the underlying problem in its entirety because, still, a new pseudo-constant will be created whenever the effective domain does not exactly match another pseudo-constant from the same position (e.g., due to different sets of invariant preconditions). As the number of distinct domains for a given base type τ is only bounded by the number of power sets over τ, we may again arrive at an intolerable number of distinct operations being produced.

As a pragmatic countermeasure, we unify certain operations after their instantiation. We define that an operation o *dominates* another operation o' if (i) $sig(o)$ and $sig(o')$ are syntactically equivalent except for any number of argument positions i where both arguments a_i of o and a'_i of o' are pseudo-constants, and (ii) both a_i and a'_i originate from the same position and $dom(a_i) \supseteq dom(a'_i)$ for each such i. After instantiating a position P, we identify operations $o' \in P$ dominated by another operation $o \in P$. In such a case, we remove each dominated operation o' from P and update the possible parents of o' to feature o as a child instead. In addition, we logically restrict the pseudo-constants of o to be equivalent to those of o' whenever o becomes active as a child of one of the parents of o'.

We found these techniques to be an effective countermeasure against an exponential blowup of the encoding as described above: On several domains, a significant number of operations are dominated and subsequently removed this way (see Table 6, Appendix C).

5. Encoding

We now present our encoding $\mathcal{L}_l(\Pi)$ of layers $L_0, \ldots, L_l$ of a TOHTN planning problem Π into propositional logic. We first provide succinct definitions for all clauses to encode. Thereafter we explain how to decode a plan from a satisfying assignment, provide a proof of correctness, and present a worst-case complexity analysis.

5.1 Base Encoding

Some parts of the propositional logic encoding we now present, namely those specified in Chapter 5.1.1, are taken from the Tree-REX approach by Schreiber et al. (2019b). The fundamental difference between Tree-REX and our new encoding is that we must now handle lifted actions and reductions and, consequently, lifted fact constraints.

In the following, we call the operations in our hierarchy actions and reductions regardless of whether they contain pseudo-constants or not. Similarly we use the term *fact* for any literal with constants and/or pseudo-constants. We use the term *ground fact* for a fact without pseudo-constants and the term *pseudo-fact* for a fact with pseudo-constants.

We use a Boolean variable o^l_x for each occurring operation o and a variable f^l_x for each ground fact per position x of each layer l of the problem. Variables $prim^l_x$ represent whether position x at layer l features a primitive operation, i.e., an action and not a reduction. In addition, for each pseudo-constant κ introduced to the problem we introduce variables $[\kappa/c]$ for each $c \in dom(\kappa)$ which represent that κ is substituted with constant c.

5.1.1 Basic Constraints

We begin with enforcing the initial reduction to hold at the only position of the first layer:

$$(r_0)_0^0 \tag{1}$$

To avoid encoding superfluous facts, we make use of the set F_l of relevant facts (see Chapter 4.2.3). We introduce a Boolean variable for each relevant fact and enforce it to assume a polarity according to the initial state at the zeroth position:

$$\begin{aligned} \forall f \in F_l \cap s_I &: \quad f_0^l \\ \forall f \in F_l \setminus s_I &: \quad \neg f_0^l \end{aligned} \tag{2}$$

If an action a occurs at position x at layer l, then we define the respective position as primitive. Similarly, if a reduction occurs, we define the position as non-primitive.

$$\begin{aligned} a_x^l &\Rightarrow \mathit{prim}_x^l \\ r_x^l &\Rightarrow \neg \mathit{prim}_x^l \end{aligned} \tag{3}$$

At most one action and at most one reduction may occur at the same position. Together with Eq. 3, this enforces that at most one operation is active at each position.

$$\begin{aligned} \forall a \neq a' \in P_{l,x} &: \quad \neg a_x^l \vee \neg (a')_x^l \\ \forall r \neq r' \in P_{l,x} &: \quad \neg r_x^l \vee \neg (r')_x^l \end{aligned} \tag{4}$$

This constraint adds $\mathcal{O}(n^2)$ clauses if there are n actions, or reductions, at the same position. In our case this quadratic measure is not as problematic as for Tree-REX because we generally instantiate significantly fewer operations. Still, if n does become too large, we use a logically equivalent encoding which introduces $\log(n)$ helper variables but only encodes $\mathcal{O}(n \log(n))$ clauses. Based on preliminary experiments we use the latter encoding if $n \geq 50$. For a specification of this encoding we refer to Schreiber (2018, Appendix A).

Any operation o at $P_{l,x}$ enforces its preconditions at $P_{l,x}$:

$$o_x^l \;\Rightarrow \bigwedge_{f \in \mathit{pre}(o)^+} f_x^l \;\wedge \bigwedge_{f \in \mathit{pre}(o)^-} \neg f_x^l \tag{5}$$

Similarly, any action a enforces its effects at $P_{l,x+1}$:

$$a_x^l \;\Rightarrow \bigwedge_{f \in \mathit{eff}(a)^+} f_{x+1}^l \;\wedge \bigwedge_{f \in \mathit{eff}(a)^-} \neg f_{x+1}^l \tag{6}$$

Later on in Chapter 5.1.2 we describe so-called frame axioms which complement the correct enforcement of action effects. Also note that each f in above formulae may contain pseudo-constants. For now we treat such pseudo-facts as we treat ground facts and encode each of them with a Boolean variable.

To logically connect subsequent layers with each another we use the following clauses. First, a ground fact f that holds at $P_{l,x}$ must be logically equivalent to the same ground fact at the first successor position $P_{l+1,s_l(x)}$ of $P_{l,x}$:

$$f_x^l \Leftrightarrow f_{s_l(x)}^{l+1} \tag{7}$$

Note that in practice these clauses do not occur in our encoding; instead we use exactly the same Boolean variable for f^l_x and $f^{l+1}_{s_l(x)}$ in the first place.

Next, we describe the sufficient and necessary conditions for operations at a new layer: When a parent o is active at $P_{l,x}$, then for each offset z one of its children o' at offset z must be active at $P_{l+1,s_l(x)+z}$.

$$\forall z \in \{0, \dots, e_{l,x} - 1\} \ : \ o^l_x \Rightarrow \bigvee_{o' \in \mathit{children}(o,z)} (o')^{l+1}_{s_l(x)+z} \tag{8}$$

Remember that $\mathit{children}(o, z)$ is well-defined for all such z: For each action a and $z > 0$, $\mathit{children}(a, z) = \{\varepsilon\}$, and for each reduction r and $z \geq |\mathit{subtasks}(r)|$, $\mathit{children}(r, z) = \{\varepsilon\}$.

Schreiber et al. (2019b) found that it is beneficial for the SAT solving performance to also redundantly enforce the opposite direction: When a child o' is active at $P_{l+1,s_l(x)+z}$, then any of its possible parents o must be active at $P_{l,x}$.

$$\forall z \in \{0, \dots, e_{l,x} - 1\} \ : \ (o')^{l+1}_{s_l(x)+z} \Rightarrow \bigvee_{o \ \mid \ o' \in \mathit{children}(o,z)} o^l_x \tag{9}$$

To conclude the set of basic constraints which were already established in a similar form by Schreiber et al. (2019b), we enforce the currently deepest layer $L_{l'}$ to be fully primitive, what implies a fully expanded hierarchical task network, before attempting to find a valid plan. The following unit clauses are added as *assumptions*, i.e., they are considered axioms by the SAT solver for the upcoming solving attempt and discarded afterwards.

$$\forall x \in \{0, \dots, |L_{l'}| - 1\} \ : \ \mathit{prim}^{l'}_x \tag{10}$$

5.1.2 Pseudo-Constants and Pseudo-Facts

Next we define the semantics of pseudo-constants and the constructs containing them.

For each pseudo-constant κ introduced by some operation o, κ must be substituted with at most one constant from its possible domain, $\mathit{dom}(\kappa)$, and if o is active then exactly one such substitution must hold:

$$\bigwedge_{c_1 \neq c_2 \in \mathit{dom}(\kappa)} \neg[\kappa/c_1] \vee \neg[\kappa/c_2] \tag{11}$$

$$o^l_x \Rightarrow \bigvee_{c \in \mathit{dom}(\kappa)} [\kappa/c] \tag{12}$$

As in Eq. 4, we employ an asymptotically better encoding instead of Eq. 11 if κ has at least $n = 50$ substitutions.

Consider a pseudo-fact f_p with pseudo-constants $\kappa_1, \dots, \kappa_k$ $(k \geq 1)$. Assume that substituting each such pseudo-constant κ_i with a particular constant $c_i \in \mathit{dom}(\kappa_i)$ yields ground fact f. Then we define:

$$\big([\kappa_1/c_1] \wedge [\kappa_2/c_2] \wedge \ldots \wedge [\kappa_k/c_k]\big) \Rightarrow \big((f_p)^l_x \Leftrightarrow f^l_x\big) \tag{13}$$

In words, we enforce a pseudo-fact to be equivalent to the ground fact it corresponds to when performing particular substitutions.

In most automated planning encodings, so-called *frame axioms* logically specify the necessary conditions for a change of polarity of a fact in between two adjacent time steps. In other words, frame axioms are necessary to prevent a SAT solver from arbitrarily changing the world state without executing a supporting action. In our encoding, frame axioms are needed for ground facts only, as the pseudo-facts are well-defined by Eq. 13. We define a fact's support, $\mathit{supp}((\neg)f)$, as the set of actions which have f as a positive (negative) effect. Also, we define a fact's indirect support, $\mathit{isupp}((\neg)f)$, as the set of actions which are not in $\mathit{supp}((\neg)f)$ but which have some pseudo-fact f_p as a positive (negative) effect that can be unified with f. We add two types of clauses to specify frame axioms:

(i) If a fact f changes its value, then either some reduction is responsible for the change (logically represented by $\neg prim^l_x$), or some action *directly* supports this fact change, or some action *indirectly* supports the fact change.

$$\begin{aligned} f^l_x \land \neg f^l_{x+1} &\Rightarrow \neg prim^l_x \lor \bigvee_{a \in \mathit{supp}(\neg f)} a^l_x \lor \bigvee_{a \in \mathit{isupp}(\neg f)} a^l_x \\ \neg f^l_x \land f^l_{x+1} &\Rightarrow \neg prim^l_x \lor \bigvee_{a \in \mathit{supp}(f)} a^l_x \lor \bigvee_{a \in \mathit{isupp}(f)} a^l_x \end{aligned} \tag{14}$$

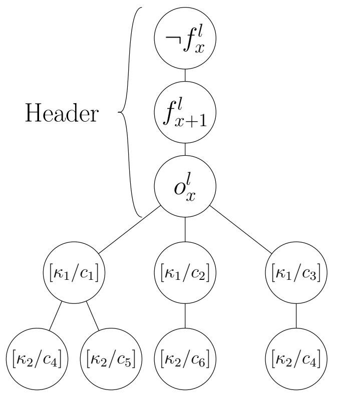
(ii) If fact f changes its value and some action $a \in \mathit{isupp}((\neg)f)$ is applied, then some set of substitutions must be active which unifies an effect f_p of a with f.

$$\begin{aligned} f^l_x \land \neg f^l_{x+1} \land a^l_x &\Rightarrow \bigvee_{\substack{f_p \in \mathit{eff}(a)^-, \\ f_p[\kappa_1/c_1]\ldots[\kappa_k/c_k]=f}} \Big(\bigwedge_{i=1}^{k} [\kappa_i/c_i] \Big) \\ \neg f^l_x \land f^l_{x+1} \land a^l_x &\Rightarrow \bigvee_{\substack{f_p \in \mathit{eff}(a)^+, \\ f_p[\kappa_1/c_1]\ldots[\kappa_k/c_k]=f}} \Big(\bigwedge_{i=1}^{k} [\kappa_i/c_i] \Big) \end{aligned} \tag{15}$$

5.1.3 Literal Trees for Sets of Substitutions

Frame axioms (ii) in the above rule are not in Conjunctive Normal Form (CNF). As such, for their encoding we require a transformation of Disjunctive Normal Form (DNF) into CNF when a features multiple effects which can be unified to f. We perform a simple transformation over a tree of Boolean literals which we can encode easily into CNF.

We build a tree rooted at the sequence of implicants from Eq. 15 which then branches over the possible substitution choices. Fig. 10 illustrates such a tree and the resulting clauses. We insert each valid set of substitutions into the tree by transforming the set into a sorted sequence and appending it to the header as a new branch. This is not necessarily the optimal construction of the tree in terms of its size (another ordering of the substitutions may lead to fewer nodes and branches overall), yet we believe that in practice this simple strategy is an acceptable compromise between construction speed and encoding size, especially considering that the arity of predicates is commonly very small in planning domains (see Table 3, Chapter 7.3.1). We can then encode equivalent CNF clauses by traversing the tree and adding a clause for each node which has substitutions as child nodes: The clause consists of the negated literals of the current branch up to the parent (as implicants)



$$f^x_l \vee \neg f^{x+1}_l \vee \neg o^l_x \vee \underline{[\kappa_1/c_1] \vee [\kappa_1/c_2] \vee [\kappa_1/c_3]}$$

$$f^x_l \vee \neg f^{x+1}_l \vee \neg o^l_x \vee \neg[\kappa_1/c_1] \vee \underline{[\kappa_2/c_4] \vee [\kappa_2/c_5]}$$

$$f^x_l \vee \neg f^{x+1}_l \vee \neg o^l_x \vee \neg[\kappa_1/c_2] \vee \underline{[\kappa_2/c_6]}$$

$$f^x_l \vee \neg f^{x+1}_l \vee \neg o^l_x \vee \neg[\kappa_1/c_3] \vee \underline{[\kappa_2/c_4]}$$

Figure 10: Exemplary literal tree (left) and corresponding clauses in CNF (right) where the intended consequence of each clause (when written as an implication) is underlined.

and the children in positive form (as possible consequences). No additional variables are required.

5.1.4 Argument Type Restrictions

Assume in the Factories domain that we can transport resources not only with trucks but with airplanes as well, and that trucks and airplanes share a common "vehicle" type τ. While some operation $\mathit{deliver}(r, \lambda, l, \nu)$ to transport resource r from λ to l may introduce ν as a pseudo-constant of type τ, some child operations $\mathit{drive\_to}(\nu, \cdot)$ and $\mathit{fly\_to}(\nu, \cdot)$ may force ν to be of the type "truck" or "airplane". More generally, it can happen that a child restricts the valid domain τ of a pseudo-constant from an earlier layer to some $\tau' \subset \tau$.

We explicitly deal with argument type restrictions by either forbidding all illegal substitutions or enforcing one of the valid substitutions:

$$\forall c \in \tau \setminus \tau' \ : \ o^l_x \Rightarrow \neg[\kappa/c] \tag{16}$$

$$o^l_x \Rightarrow \bigvee_{c \in \tau'} [\kappa/c] \tag{17}$$

We dynamically decide on whether to encode Eq. 16 or Eq. 17 based on which of the sets induces a smaller overall number of Boolean literals. Empirically we found that oftentimes one of the two sets is very small.

5.1.5 Actions with Contradictory Effects

In PDDL and, by extension, in HDDL we allow an operator o to have both f and $\neg f$ as an effect. Seeming contradictory at first glance, it is indeed consistent with the common semantics of applying an action in automated planning: First all negative effects are deleted from the state and then all positive effects are added to the state (e.g. Ghallab, Nau, and Traverso (2004), p. 30 Def. 2.7). To demonstrate the use of such a construct, consider a simple operator $\mathit{goto}(v, x, y)$ where v is a vehicle and x and y are waypoints, with a precondition $\mathit{at}(v, x)$ and two effects $\neg \mathit{at}(v, x)$, $\mathit{at}(v, y)$. When we instantiate some action $\mathit{goto}(V, L, L)$ from this operator, the two effects become contradictory. Yet, depending on the intention of the domain modeler, the action may still be important because at some point it may be required to execute the action of V going from L to L, i.e., staying there.

In encodings generated from ground representations, each action can be trivially preprocessed by deleting each effect $\neg f$ for which f is also an effect. In our lifted encoding, whether an action contains contradictory effects generally depends on which substitutions are applied. Our encoding specified so far may then logically imply both f_x^l and $\neg f_x^l$ at the same time, leading to a contradiction and rendering the formula unsatisfiable.

Our solution to this problem is to encode effects in the following way: Each positive effect of action a is encoded normally. For each negative effect $f \in \mathit{eff}(a)^-$ we collect all positive effects $f' \in \mathit{eff}(a)^+$ such that f and f' share the same predicate. We then compute the set Σ of all substitution sets which unify f with such an f'. We distinguish three cases:

1. If $\Sigma = \emptyset$, then there are no conflicting positive effects for negative effect $\neg f$ and the effect will be encoded normally as in Eq. 6.

2. If $\emptyset \in \Sigma$, i.e., f is already unified with some $f' \in \mathit{eff}(a)^+$ without applying any substitution, then f and f' are syntactically equal: The negated effect is discarded because it is always overridden by the positive effect.

3. Otherwise, $\Sigma := \{\Sigma_1, \ldots, \Sigma_m\}$ where each Σ_i unifies some $f' \in \mathit{eff}(a)^+$ with f. We enforce that either the negative effect holds or one of these unifications is active:

$$o_x^l \Rightarrow \neg {f_p}_{x+1}^{\,l} \vee \bigvee_{i=1}^{m} \bigwedge_{[\kappa/\alpha] \in \Sigma_i} [\kappa/\alpha] \tag{18}$$

Let us take a closer look at the above literals of the form $[\kappa/\alpha]$. Whenever one effect's argument κ is a pseudo-constant while the other's α is a constant, $[\kappa/\alpha]$ is a substitution variable. Also, at least one of κ and α must be a pseudo-constant: For constants $c \neq c'$, substitutions of the form $[c/c']$ are invalid and substitutions of the form $[c/c]$ are redundant and hence omitted. What remains is the special case of unifying a pair of pseudo-constants, i.e., $\kappa' := \alpha$ is a pseudo-constant as well. For each such case we introduce a new Boolean variable and give it the meaning: "κ and κ' are equal."

To have a variable $[\kappa/\kappa']$ assume this meaning, we introduce additional clauses: First the intersection of both domains, $I = \mathit{dom}(\kappa) \cap \mathit{dom}(\kappa')$, and the respective differences, $D := \mathit{dom}(\kappa) \setminus I$ and $D' := \mathit{dom}(\kappa') \setminus I$, are computed. If I is empty, then the pseudo-constants cannot be equal: $[\kappa/\kappa']$ is `false`. Otherwise, we encode clauses to guarantee that $[\kappa/\kappa']$ holds if and only if both pseudo-constants are substituted with the same constant:

$$\begin{aligned}
\forall c \in I \; &: [\kappa/\kappa'] && \Rightarrow ([\kappa/c] \Leftrightarrow [\kappa'/c]) \\
\forall c \in I \; &: ([\kappa/c] \wedge [\kappa'/c]) && \Rightarrow [\kappa/\kappa'] \\
\forall c \in D \; &: [\kappa/c] && \Rightarrow \neg[\kappa/\kappa'] \\
\forall c \in D' &: [\kappa'/c] && \Rightarrow \neg[\kappa/\kappa']
\end{aligned} \tag{19}$$

In practice, we realize Eq. 18 with literal trees (Chapter 5.1.3) and encode Eq. 19 whenever a new equality variable emerges which did not occur before.

5.1.6 Dominated Operations

Last but not least, we turn to the situation where some operation o dominates another operation o' as described in Chapter 4.4. Whenever o becomes active as a child of one of the parents of o' (but no parent of o), we restrict the pseudo-constants of o to be equivalent to those of o'. Again, we make use of variables $[\kappa/\kappa']$ as defined above.

$$\forall o_p \in P_{l,x} \mid o' \in \mathit{children}(o_p, z),\ o \notin \mathit{children}(o_p, z) :\ (o_p)^l_x \wedge o^{l+1}_{s_l(x)+z} \Rightarrow \bigwedge_{\kappa \in o, \kappa' \in o'} [\kappa/\kappa'] \tag{20}$$

This concludes the set of clauses which are required for the correctness of our approach in conjunction with the techniques described in Chapter 4.

5.2 Optimizations

In the following, we describe some improvements to our encoding which are not necessary for correctness but are nevertheless important for the overall performance of our approach.

5.2.1 Accounting for Invariant Facts

So far, we explicitly encoded all facts which appear as a precondition or as an effect at some position $P_{l,x}$. However, during the instantiation of our approach, we perform a reachability analysis which allows us to identify invariant facts, i.e., (ground) facts which are definitely true or definitely false for all reachable world states at a certain position (see Chapter 4.2). We can exploit this knowledge to significantly cut the number of encoded facts and enforced constraints as explained in the following.

First and foremost, we avoid to introduce Boolean variables for invariant facts. Instead of encoding each relevant fact at the zeroth position, we adjust Eq. 2 as follows:

$$\begin{aligned} \forall f \in F_l \mid f \text{ is invariantly true at } P_{l,x} \text{ but not at } P_{l,x+1} :&\quad f^l_x \\ \forall f \in F_l \mid f \text{ is invariantly false at } P_{l,x} \text{ but not at } P_{l,x+1} :&\quad \neg f^l_x \end{aligned} \tag{21}$$

In words, at each layer L_l we delay the encoding and initialization of fact f as a Boolean variable until the last position where f is invariant. Note that all facts are invariant at the zeroth position due to s_I and the closed-world assumption. Replacing Eq. 2 with Eq. 21 allows us to skip many trivial frame axioms which preserve the polarity of an unchanging fact (see Eq. 14 with empty supports) and completely omits the encoding of globally invariant facts. Whenever f is invariant and hence not encoded, we consequently do not encode the equivalence of f to any present pseudo-fact f_p (Eq. 13). However, to preserve correctness, we may need to introduce some other constraints instead.

First, if an operation (action) at $P_{l,x}$ has a ground precondition (effect) that is invariantly true at $P_{l,x}$ ($P_{l,x+1}$), then we can simply omit Eq. 5 (Eq. 6) for this particular constraint. Note that effects are never invariantly false due to construction, and operations with invariantly false preconditions are pruned during instantiation.

Secondly, assume that operation o has k preconditions with pseudo-constants. For each such precondition f, each ground fact resulting from f is either invariantly false or invariantly true or not invariant. All non-invariant facts are handled as before by linking them with a pseudo-fact (Eq. 13). In addition, we must make sure that no substitution is applied which transforms f into an invariantly false precondition. Generally, for each $1 \leq i \leq k$, there are n_i sets $\Pi_{i1}, \ldots, \Pi_{in_i}$ of substitutions rendering precondition i invariantly false, and m_i remaining sets $\Sigma_{i1}, \ldots, \Sigma_{im_i}$ for which precondition i may hold (invariantly or not). For each precondition, we either enforce that some valid substitution set must hold (Eq. 22) or that none of the invalid substitution sets must hold (Eq. 23).

$$\forall i \in \{1, \ldots, k\} \;:\; o^l_x \Rightarrow \bigvee_{j=1}^{m_i} \bigwedge_{[c/\kappa] \in \Sigma_{ij}} [c/\kappa] \tag{22}$$

$$\forall i \in \{1, \ldots, k\} \; \forall j \in \{1, \ldots, n_i\} \;:\; o^l_x \Rightarrow \bigvee_{[c/\kappa] \in \Pi_{ij}} \neg[c/\kappa] \tag{23}$$

As in Eq. 16–17, we encode the smaller of these sets. Eq. 22 is realized using literal trees.

As a special case, if all ground facts which may result from a precondition are invariant, we can omit the pseudo-fact corresponding to the precondition (Eq. 13). This situation occurs frequently in practice because most planning domains contain so-called *rigid predicates* (Ghallab et al., 2004, p. 43) which are not featured in any action effect. In our Factories example (Fig. 7), some rigid predicates are $A{\leftrightarrow}B$ (there is a road between A and B) and $F_1{\Rightarrow}R$ (factory F_1 can produce resource R).

We now turn to effects with pseudo-constants. Each ground fact that can result from such an effect is considered a possible fact change by our reachability analysis. As such, an effect is never invariantly false. It can happen, however, that some substitutions which turn the effect into a ground fact are known to be invalid because a precondition of the same operation becomes invariantly false for these substitutions. We omit Eq. 13 for each such substitution; the according substitutions are already prohibited by the precondition's encoding. If all ground facts resulting from the effect are either omitted this way or invariantly true, we do not encode an according pseudo-fact.

5.2.2 Reduction of Encoded Variables and Clauses

We employ miscellaneous techniques to reduce the number of encoded variables and clauses.

Variables prim^l_x are encoded only where necessary, i.e., at positions where both primitive and non-primitive operations may occur. Otherwise, the variable is considered as constant `true` or constant `false` respectively: If a literal is `false` it is not added to a clause, and if a literal is `true` the whole clause is discarded.

Whenever a fact f must remain unchanged in between positions $P_{l,x}$ and $P_{l,x+1}$, i.e., its direct and indirect supports are empty for both polarities, then we know that the two corresponding Boolean variables must be equivalent: If f^l_{x+1} is not already defined by reusing a variable from the parent position, then we can omit the frame axioms for this fact and instead use variable f^l_x from $P_{l,x}$ to represent f^l_{x+1} as well. We can completely skip a position's frame axioms whenever its parent position features a superset of possible operations and both positions exclusively feature primitive operations, as in this case the full frame axioms have already been specified above and do not need to be re-encoded.

Similarly, if a pseudo-fact occurs at positions $P_{l,x}, P_{l+1,s_l(x)}$ or at positions $P_{l,x}, P_{l,x+1}$, then we reuse the prior variable if the two pseudo-facts encompass the same set of ground facts and, in the latter case, if the pseudo-fact cannot change its polarity in between the two positions.

Our instantiation approach implies that an action occurring at some layer L_l will also be contained and encoded at layers $L_{l+1}, L_{l+2}, \ldots$, inducing clauses which repeatedly enforce the action's preconditions and effects at every layer. Instead, we define the child operation of action a as the *repetition* a^* of a. We do not encode any preconditions or effects for such repetitions of actions as they are already enforced at an earlier layer and will be propagated down. We treat a^* just like a when we compute frame axioms and when we decode a plan.

5.3 Decoding a Plan

We now explain the process of decoding a classical and hierarchical solution from a satisfying assignment to our encoding found by a SAT solver.

Assume that the encoding $\mathcal{L}_{l'}$ of layers L_0 through $L_{l'}$ is satisfiable and that a satisfying assignment $\mathcal{A}$ to all variables is available. We define that an operation o is *active* at $P_{l,x}$ iff $\mathcal{A}(a_x^{l'}) = \texttt{true}$. Similarly, a substitution $[\kappa/c]$ is *active* at $P_{l,x}$ iff $\mathcal{A}([\kappa/c]) = \texttt{true}$.

We first decode a plan π (see Def. 1) from $\mathcal{A}$: We begin with an empty plan $\pi := \langle\rangle$. For each position index $x = 0, \ldots, |L_{l'}| - 1$ we find an active action $a \in P_{l',x}$. If a is an ε-action, we discard it. Otherwise, if a is ground, we append it to π. Otherwise, for each pseudo-constant κ of a we find an active substitution $[\kappa/c]$ and substitute all occurrences of κ in action a with c. The resulting ground action $\tilde{a}$ is appended to π.

To obtain the hierarchical solution leading to π (see Def. 2) we begin with a graph H without any edges and a single node $(r_0, 0, 0)$ which represents the initial reduction r_0 at the zeroth position of layer L_0. We traverse all layers in the order of their instantiation. For each position $P_{l+1,s_l(x)+z}$ where $l + 1 > 0$, we first examine the parent node (o, l, x) in H. If o is an action, then we ignore the child position and continue. Otherwise we find an active child operation o' at $P_{l+1,s_l(x)+z}$, we add a new node $(\tilde{o}', l + 1, s_l(x) + z)$ where $\tilde{o}'$ is the ground representation of o' as explained above for actions $\tilde{a}$, and we add edge $((o, l, x), (\tilde{o}', l + 1, s_l(x) + z))$ to H.

We illustrate this decoding process with our Factories example in Fig. 7: We obtain π by traversing L_4 from left to right and collecting all active actions which are not ε-actions. For each pseudo-constant in each action, we apply the according highlighted equivalence in the top left corner of Fig. 7. Remember that the reduction *goto_noop*(θ, C) at $P_{4,6}$ is treated as an action but is omitted from π, as explained in Chapter 4.1.2. Fig. 7 also contains a representation of the hierarchical solution H: Each position with a highlighted operation corresponds to a node in H except for positions with repeated actions from an earlier layer ($P_{3,1}$, $P_{4,1}$, $P_{2,3}$, etc.) and positions where an ε-action is active.

5.4 Correctness

In the following, we establish our proof of correctness for our encoding on an intuitive level. The proofs themselves are found in Appendix A.

First, we show some important fundamentals for further arguments.

Lemma 1. *For any satisfying assignment $\mathcal{A}$ for $\mathcal{L}_{l'}(\Pi)$ there is exactly one active operation at each position $P_{l,x}$ for $l \in \{0, \ldots, l'\}$ and $x \in \{0, \ldots, |L_l| - 1\}$.*

Lemma 1 follows from the introduced at-most-one and primitiveness constraints.

Lemma 2. *The result π of decoding a plan from a satisfying assignment $\mathcal{A}$ for $\mathcal{L}_{l'}(\Pi)$ as described in Chapter 5.3 is well-defined and unambiguous.*

To show Lemma 2 we use Lemma 1 and the fact that there is exactly one active substitution for each pseudo-constant. Next, we establish a mapping of satisfying assignments of fact variables to world states:

Lemma 3. *Let $\mathcal{A}$ be a satisfying assignment for $\mathcal{L}_{l'}(\Pi)$. Then for each position $P_{l,x}$ we can unambiguously infer a world state $s_{l,x}$ based on $\mathcal{A}$ and s_I.*

The mapping $(\mathcal{A}, s_I) \mapsto s_{l,x} := \{f \mid \mathcal{A}(f^l_x) = \texttt{true}\} \cup \{f \in s_I \mid \mathcal{A}(f^l_x) = \bot\}$ achieves the desired: The world state consists of all facts which are explicitly `true` plus all facts which are (yet) unencoded but hold in the initial state.

In order to reason about the correct application of actions, we show that whenever a particular action with pseudo-constants is active, all preconditions and effects of the implied ground action are enforced correctly.

Lemma 4. *Consider a satisfying assignment $\mathcal{A}$ for $\mathcal{L}_{l'}(\Pi)$ and an active operation o at position $P_{l,x}$ which becomes a ground operation $\tilde{o}$ through zero or more active substitutions. Then the following holds:*
(i) $pre(\tilde{o})^+ \subseteq s_{l,x}$ and $pre(\tilde{o})^- \cap s_{l,x} = \emptyset$.
(ii) If $\tilde{o}$ is an action, then $s_{l,x+1} = (s_{l,x} \setminus eff(\tilde{o})^-) \cup eff(\tilde{o})^+$.

Lemma 4 follows from the constraints for preconditions and effects together with the definition of pseudo-facts and their link to ground facts given particular substitutions.

An induction over the length of the final layer (using Lemma 4) leads us to the following central property which guarantees that any classical plan we decode is executable:

Lemma 5. *When a plan $\pi = \langle a_0, \dots, a_{k-1}\rangle$ is decoded from a valid satisfying assignment $\mathcal{A}$ for $\mathcal{L}_{l'}(\Pi)$, there is a sequence of states $Q = \langle s_0 := s_I, s_1, \dots, s_k\rangle$ such that $pre(a_i)^+ \subseteq s_i$, $pre(a_i)^- \cap s_i = \emptyset$, and $s_{i+1} = (s_i \setminus pre(a_i)^-) \cup pre(a_i)^+$ hold for $0 \leq i < k$.*

In the proof we show that the desired sequence of states is essentially equivalent to the sequence of states $s_{l',x}$ defined as in Lemma 3 except for additional redundant world states induced by ε-actions. Next, we turn to the hierarchical solution for our problem.

Lemma 6. *Let H be the decoded hierarchical solution for $\mathcal{L}_{l'}(\Pi)$. Then the structure of H resembles an actual hierarchical solution, i.e., H satisfies (1) and (2) from Def. 2.*

This "structural integrity" of our hierarchical solution H is shown by an induction over the depth of H where we find that only valid child nodes, matching a subtask of an earlier node, are added to H. Similar to the executability of the classical plan shown in Lemma 5, we argue that the hierarchical plan is executable as well:

Lemma 7. *Let (π, H) be the decoded (classical and hierarchical) solution for the Lilotane encoding $\mathcal{L}_{l'}(\Pi)$ for TOHTN planning problem Π. Traverse H as described in (3) in Def. 2 with the node ordering relation $(g, l, x) \prec (g', l', x') \leftrightarrow x < x'$ and maintain a state s which is initialized as s_I and updated with the effects of each visited action. Then the preconditions of all visited actions and reductions hold in s.*

For this proof we make use of Lemma 5 and 6 as well as the argument that each reduction precondition is being enforced at the correct position. Assembling all the parts, we can finally show that our encoding functions as intended:

Theorem 2. *Let (π, H) be the decoded (classical and hierarchical) solution for the Lilotane encoding $\mathcal{L}_{l'}(\Pi)$ for TOHTN planning problem Π. Then π is a valid solution for Π and H is a valid hierarchical solution for Π.*

We prove this theorem by transforming the recursive formulation of Def. 1 into an iterative procedure where the decisions of how to achieve the next task are dictated by the simultaneous traversal of H. In particular, we make use of a one-to-one correspondence between the tasks T to be achieved and the frontier of nodes in H to be visited.
As a side note, we point out that above proof also implies the correctness of the previous Tree-REX encoding (Schreiber et al., 2019b) for the much simpler case where no pseudo-constants are introduced, only ground operations are added, and all relevant facts are encoded at position zero of each layer. This has not been shown before.

While we do not provide an explicit proof for the completeness of our encoding, note that a similar chain of arguments can be made to show that whenever a problem Π has a solution (π, H) at depth l', our encoding will be satisfiable at layer l' and enable us to extract a valid solution from a satisfying assignment.

5.5 Complexity

In the following, we assess the complexity of the Lilotane encoding, providing the worst case asymptotic number of variables and clauses which emerge during the encoding of $\mathcal{L}_{l'}(\Pi)$.

We first establish a simplified worst-case model for the structure of the problem at hand, the worst case being a hierarchy which indefinitely grows exponentially both regarding the size of its layers and the number of possible operations at each position. Let $X := \max\{|\mathit{subtasks}(m)| \mid m \in M\}$ be the maximum expansion size, i.e., the maximum number of subtasks, of any method. Let B be the maximum lifted branching factor per subtask, i.e., the maximum number of methods with different signature names which achieve the same task. We can see that, given the initial layer size $|L_0| = 1$, the size of layer l' is in $\mathcal{O}(X^{l'})$. Furthermore, given that the number of encoded operations per position can multiply by a factor of B for each further layer, the total number R of encoded operations is in $\mathcal{O}(X^{l'}B^{l'})$.

Let V be the maximum arity (i.e., the number of arguments) of any operation, and let U be the maximum number of free arguments of any method with respect to the task it achieves. Let C be the number of constants, and let P (E) be the maximum number of preconditions (effects) of any operation. Then the number of encoded variables is in

$$\mathcal{O}(X^{l'}(F + B^{l'}(UC + P + E + V^2))) \tag{24}$$

as we derive in Appendix B. Essentially, at each position we need to encode each fact ($X^{l'}F$); and for each operation at some position, we need to encode each new pseudo-constant ($X^{l'}B^{l'}UC$), each pseudo-fact originating from a precondition or an effect ($X^{l'}B^{l'}(P+E)$), and possibly an equality variable for each pair of pseudo-constants in the action ($X^{l'}B^{l'}V^2$).

Let us compare this to the complexity of the previous Tree-REX approach (Schreiber et al., 2019b) for a worst-case result of grounding: When expanding a position in the hierarchy, each operation at some position can lead not to B, but instead to $B \cdot C^U$ new operations for each subtask because we fully ground all operations. This creates $\mathcal{O}(X^{l'}(BC^U)^{l'})$ operations in total and overall leads to $\mathcal{O}(X^{l'}(F + (BC^U)^{l'}))$ variables. However, also note that for Tree-REX the number of operations per position cannot grow indefinitely but is bounded by $\mathcal{O}((|M| + |O|)C^V)$, i.e., the number of syntactically instantiateable operations, while for Lilotane the number of operations at each position is unbounded: At each layer, new pseudo-constants and thus "new" operations may be introduced. This is an issue which we counteract with shared pseudo-constants and dominating operations (Chapter 4.4).

Regarding the number of encoded clauses, we arrive at an asymptotic number of

$$\mathcal{O}\Big(X^{l'} B^{l'} \big(l' \log B + C(U \log C + V^2) + F(P + YE) + YE^2\big)\Big) \tag{25}$$

permanent clauses (i.e., not counting $X^{l'}$ assumptions), as is derived in Appendix B.

If the maximum operation arity V, the maximum number U of unbound arguments in a method, and the maximum predicate arity Y are negligible constants, we obtain

$$\mathcal{O}\Big(X^{l'} B^{l'} \big(l' \log B + C \log C + F(P + E) + E^2\big)\Big) \tag{26}$$

clauses. Thereby terms $l' \log B$ and $C \log C$ are caused by at-most-one constraints over the operations at each position and over the substitutions for a pseudo-constant, term $F(P+E)$ is implied by (among others) the semantics of pseudo-facts, and term E^2 originates from the encoding of contradictory action effects.

For Tree-REX, applying our complexity model under above assumptions yields

$$\mathcal{O}\Big(X^{l'} \big(T \cdot (\log T + P + E) + F\big)\Big) \tag{27}$$

clauses (see Schreiber et al., 2019b, Complexity) where $T := \min\{B^{l'} C^{Ul'}, (|M| + |O|)C^V\}$.

The advantages of Tree-REX are that only a constant number of clauses is added for each fact, each precondition and each effect at each position and that, again, there is an upper bound on the number of operations at each position. By contrast, while Lilotane may encode more clauses per operation due to its more complex handling of facts, its much smaller initial branching factor leads to fewer operations by a factor of $C^{Ul'}$ as long as the upper bound for T is not reached.

The exponential complexity of both encodings, which are compilations of 2-EXPTIME-complete TOHTN planning to NP-complete SAT (see Chapter 2.1.2), is presumably unavoidable. However, it is worth noting that the number of clauses and variables encoded by Lilotane is exponential only in l' while for Tree-REX the encoding size is exponential both in l' and in either Ul' or V. Furthermore, we observed that the theoretical issue of Lilotane producing an unbounded number of operations at each position is rarely a problem in practice because shared pseudo-constants and dominating operations (see Chapter 4.4) can unify operations which have a similar or equal meaning.

We conclude from the perspective of asymptotic complexity that the Lilotane encoding, despite being a generalization of the Tree-REX encoding, is not a pure improvement of the latter but rather a new and to some degree orthogonal approach which focuses on reducing the number of encoded operations, in particular at the first hierarchical layers, at the cost of a more complex logic related to the problem's facts. Evaluations in Chapter 7 will shed more light on these considerations and complement our theoretical study in this chapter with empirical practical insights.

6. Plan Improvement

The length of a given plan $\pi = \langle a_0, \dots, a_{k-1} \rangle$ is given by $|\pi| = k$. We are interested in finding as short plans as possible because in a real-world application each action will require some effort in order to be executed: In most cases, shorter plans are more efficient and executed faster. This simple cost model which considers all actions equally costly is used by related work on quality-aware and/or optimal HTN planning (Behnke et al., 2019b; Schreiber et al., 2019b) and will be used in the following as well.

Similar to its precursor Tree-REX, the base algorithm of Lilotane for finding a plan generally produces sub-optimal plans. At various positions we introduce ε-actions which do not contribute to the plan length. Consequently, in order to minimize $|\pi|$ we must maximize the number of active ε-actions at the layer l where a plan was found. We can see that this optimization (however it is realized) will yield an optimal plan at layer l, which we call a *depth-optimal plan*, but not necessarily a globally optimal plan: A different choice of methods which require a larger depth to be fully expanded may be able to induce an overall smaller number of actual actions, i.e., a higher number of ε-actions. Hence, we may find an even shorter plan by admitting a deeper hierarchy (see Behnke et al., 2019b). In our plan improvement approach we will first construct a depth-optimal plan but no globally optimal plan. Then we argue how our approach can eventually find and output an optimal plan.

6.1 Previous Approaches

As a point of departure, consider the approach of Tree-REX by Schreiber et al. (2019b):

1. After finding an initial plan π_0 at layer $L_{l'}$, the primitiveness of all positions at $L_{l'}$ is enforced permanently by adding Eq. 10 as unit clauses instead of assumptions.

2. We encode further variables and clauses to count the length of a plan in such a way that specific assumptions can restrict the possible plan lengths for a single SAT solver call. We initialize iteration counter $i = 0$.

3. We add assumptions to forbid any plan length equal to or greater than the previous plan length $|\pi_i|$ and call the SAT solver again. In case of satisfiability, we decode the new plan π_{i+1}, count its new length $|\pi_{i+1}| < |\pi_i|$, and repeat 3. for incremented i. In case of unsatisfiability we return π_i which has then proven to be depth-optimal.

This procedure belongs to a broad class of optimization approaches which contain as a subprocedure a decision problem "Is there a solution of cost $\leq k$?" for changing k. This class of approaches has been explored by Rintanen (2004) for the case of scheduling SAT-based planning horizons and was generalized by Streeter and Smith (2007). Both identified query strategies that also incorporate time limits for the decision procedure and are in general more sophisticated than the above linear search strategy.

Faced with a similar search problem, Behnke et al. (2019b) considered three simple strategies for optimal (non-incremental) SAT-based HTN planning, namely a linear increase of k beginning at zero (INC), a linear decrease of k beginning at a value determined by some initial plan (DEC), and a bisection search between these two initial bounds (BIN). They found that the three strategies performed very similarly overall. Independently, Schreiber (2018) evaluated three equivalent search strategies for the plan length optimization of Tree-REX and found the DEC strategy the most appealing. This strategy produces a series of $i \geq 0$ SAT results concluded by a single UNSAT result. As such, an improved plan can be decoded from every intermediate result what leads to an effective anytime approach. In addition, a SAT solver often finds a plan of some length $k' < k$ which allows to skip tests for any of the intermediate values, resulting in good practical performance.

6.2 Our Approach

We build upon the approach of Tree-REX with the DEC strategy as described above and further exploit its monotonic nature for efficient incremental SAT solving.

We use variables which represent that the plan length up to position $P_{l',x-1}$ is *exactly equal* to some k whereas the counter variables of the Tree-REX encoding represent that the plan length up to $P_{l',x-1}$ is *at least* some k. Furthermore, we do not naïvely encode these variables for all possible positions and all plan lengths from 0 to $|L_{l'}|$: Instead, while traversing the final layer from left to right we maintain a "corridor" of possible plan lengths, update its bounds at each step, and only encode new variables where absolutely necessary.

At position $P_{l',0}$, the plan so far is of length zero, so we set our initial bounds to $d_0 = 0$ ("down") and $u_0 = 0$ ("up"). Consequently there are $u_0 - d_0 + 1 = 1$ possible plan lengths up to this position, so we encode a single variable $v_{0,0}$ and enforce it to be `true`.

Consider position $P_{l',x}$ for $x \geq 0$ where we have previous bounds d_x and u_x and encoded $u_x - d_x + 1$ different variables representing the possible lengths $d_x, \dots, u_x$ of partial plans up to position x (exclusively). We analyze position x: If only ε-actions and no other actions are possible at this position, we know that our plan length will stay the same. For instance, this can happen frequently if an operation is pruned retroactively (see Chapter 4.2.3) and leaves behind a number of otherwise empty child positions. In this case we do *not* encode new variables but carry over the variables we already encoded to the next position, and we keep the bounds $d_{x+1} = d_x$ and $u_{x+1} = u_x$. Conversely, if no ε-actions can occur at this position, we know that our plan length will increase exactly by one. Again, we just carry over the existing counter variables and update $d_{x+1} = d_x + 1$ and $u_{x+1} = u_x + 1$.

In the third case where both normal actions and ε-actions may occur at position x, we need to introduce new variables: We know that the plan length will either remain the same or increase by one, so we set $d_{x+1} = d_x$ and $u_{x+1} = u_x + 1$. If $v_0, \dots, v_{u_x - d_x}$ are the previous variables, we encode new variables $v'_0, \dots, v'_{u_{x+1} - d_{x+1}}$ and the following clauses:

$$(a_\varepsilon)_x^{l'} \wedge v_i \Rightarrow v'_i \tag{28}$$

$$\neg(a_\varepsilon)_x^{l'} \wedge v_i \Rightarrow v'_{i+1} \tag{29}$$

In words, we update the current plan length d_{x+i} to $d_{x+i} + 1$ if a normal action is active and to d_{x+i} otherwise. We do not need to enforce the other direction of these implications.

In the end, for upper and lower bounds $\hat{u}$ and $\hat{d}$ we can successively forbid plan lengths by adding restrictions of the form $\neg v_{\hat{u}-\hat{d}}, \neg v_{\hat{u}-\hat{d}-1}, \dots$ to the encoding. We add these clauses not as assumptions but as permanent unit clauses: SAT solvers can perform more simplification once a unit constraint is known to be permanent (e.g., Nadel & Ryvchin, 2012, p. 243). This optimization is only possible due to the monotonic nature of the DEC search strategy.

Let $\mu \leq |L_{l'}|$ be the number of "mixed positions" at layer $L_{l'}$ where both ε-actions and normal actions can occur. Then the number of variables of our plan improvement encoding is $V \approx \frac{\mu^2}{2}$ and the number of clauses is V times a small constant. In case of $\mu \approx |L_{l'}|$ these complexity measures are equivalent to the encoding from Tree-REX which always leads to approximately $\frac{|L_{l'}|^2}{2}$ variables. Empirically we noticed that μ is often noticeably smaller than $|L_{l'}|$ and that even a modest difference leads to considerably fewer variables and clauses due to the squared complexity.

6.3 Finding Globally Optimal Plans

Using our approach to find a depth-optimal plan, we sketch a simple non-terminating procedure which lets Lilotane eventually output a globally optimal plan under the assumption that enough time and memory are available:

Find an initial plan at some layer L_l and perform plan improvement until a depth-optimal plan is found. Beginning with $k = 1$, instantiate and encode k additional layers, find an initial plan at layer L_{l+k} and perform plan improvement until a depth-optimal plan is found; then update $l := l + k$ and repeat for doubled k.

The number k of further layers to instantiate and encode until another plan improvement is performed increases exponentially. As such, the clauses required for plan improvement are encoded only a logarithmic number of times with respect to the number of additional layers. Also note that for this procedure we need to enforce the primitiveness of operations and forbidden plan lengths as assumptions and not as permanent unit clauses because they must be reset when another instance of plan improvement is performed at a later layer.

We consider this mode of operation to be useful in scenarios where the planner operates under a fixed time limit. Hitting this time limit (or a memory limit), the planner is interrupted and outputs the best found plan so far.

7. Evaluation

In the following, we discuss an extensive evaluation of Lilotane which consists both of the IPC 2020 and our own evaluations.

7.1 Implementation

We have implemented our approach in C++17. Our source code is available at `www.github.com/domschrei/lilotane` and all experimental data is available at `www.github.com/domschrei/lilotane-experimental-data`. We make use of pandaPIparser (Behnke et al., 2020) for parsing and performing light preprocessing of HDDL planning problems. We make use of an efficient implementation of unordered hash sets and maps (Ankerl, 2020) to map signatures of facts and operations to variables and other related objects. We used the Re-entrant Incremental SAT solver API (IPASIR, see Balyo, Biere, Iser, & Sinz, 2016) and link our software with a SAT solver. As was the case for Tree-REX, we found Glucose (Audemard & Simon, 2009) to empirically work best among various solvers for the family of SAT problems produced by our encoding approach. As such, we linked Lilotane with Glucose for all evaluations.

7.2 Lilotane as a SAT-Based HTN Planner

As a natural first stage of our evaluations, we compare our planner to its precursor Tree-REX and to PANDA-SAT. We included the up-to-date version of Lilotane, a quality-aware variant which finds a depth-optimal plan at the layer where the initial plan was found (LilotaneQ), Tree-REX without plan improvement, and three configurations of PANDA-SAT: the totally ordered version and hence most direct competitor (PANDA-totSAT, Behnke et al., 2018), the best performing version supporting partial orderings (PANDA-SAT, Behnke et al., 2019a), and the best performing optimal HTN planner to date (PANDA-SAT-OPT, Behnke et al., 2019b). We use PANDA in conjunction with SAT solver Cryptominisat (Soos, Nohl, & Castelluccia, 2009), employ the configuration `sat-exists-forbidden-implication` for the partial order and optimal variant and use the `BIN` search strategy for the optimal version.

Unfortunately, neither PANDA in its current form nor Tree-REX are equipped to handle the benchmarks of the IPC due to technical limitations such as parsing errors and a differing input model. As such, we limited our evaluation to the domains which have been prepared for the comparison of the two planners by Schreiber et al. (2019b) with the help of G. Behnke. While ten of these domains were part of the published evaluation of Tree-REX vs. PANDA-SAT, we include two more domains (Elevator and Zenotravel) which have been used exclusively for tuning purposes in the original publication. Due to similar technical limitations we cannot compare the plans output by Tree-REX and by Lilotane in a fair manner, which is why we did not include a plan improving variant of Tree-REX.

We fixed some notable issues with the grounding backend of Tree-REX to ensure a fair comparison. As acknowledged first by Behnke et al. (2020), two false assumptions are made in the grounding procedure which we rectified: that each constant from the problem is contained in each action's and each method's arguments at most once, and that actions without any effects can be discarded. As a third change, we have removed a condition in the code which caused to discard actions named `nop`. We also removed an erroneous precondition in the Zenotravel domain which led to unsolvable problems in the patched variant of Tree-REX and fixed an inaccuracy in the translated HDDL model of Transport.

We set a timeout of five minutes and a memory limit of 8GB. The experiments have been conducted on a desktop PC running Ubuntu 18.04 with a quad-core Intel i7-6700 processor clocked at 3.40GHz and with 32GB of DDR4 RAM. The runs were performed sequentially.

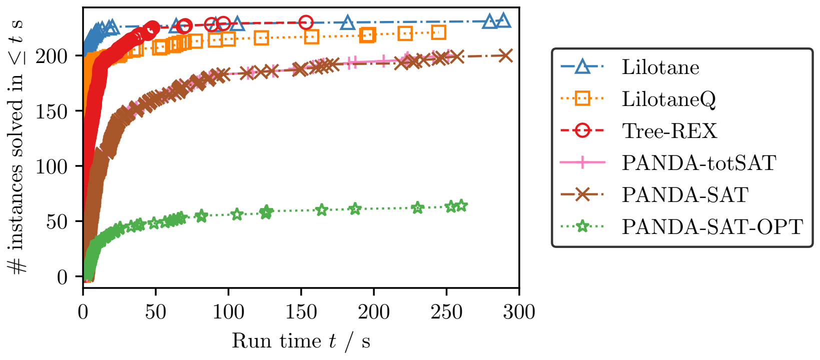


Figure 11: Overview of run times of PANDA-SAT, Tree-REX, and Lilotane

7.2.1 Overview

An overview of the results regarding run times is given in Fig. 11; more detailed plots for each domain are given in Fig. 16, Appendix C. The optimal configuration of PANDA completed 64 out of 242 instances. However, note that PANDA-SAT-OPT employs an anytime algorithm and found *some* plan on 193 instances (not pictured). The two configurations for partial orderings and total orderings performed very similarly and both solved 200 instances: As PANDA-SAT acknowledges each problem to be totally ordered, we believe that it falls back to an encoding variant of PANDA-totSAT. Tree-REX solved 230 instances and Lilotane solved 232 instances. The quality-aware variant LilotaneQ completed plan improvement on 221 instances, including all 189 instances for which PANDA-SAT-OPT found some plan. Among these commonly solved instances, LilotaneQ found a shorter plan in 89 cases and matched the plan length of PANDA-SAT-OPT in the remaining 100 cases. In particular, LilotaneQ found an optimal plan wherever PANDA found an optimal plan.

Detailed comparisons of Lilotane with PANDA-totSAT and with Tree-REX are shown in Fig. 12. Each point (x, y) corresponds to a single instance. For points along the diagonal $y = x$ both approaches performed equally well. For points on the i-th diagonal above (below) the central diagonal, Lilotane performed better (worse) by i orders of magnitude.

In the left graphs, raw solving times are compared. Both PANDA and Tree-REX have a considerable overhead associated to every run which leads to a large relative difference in run times at the bottom left, i.e., for easier instances. This gap is much more pronounced for PANDA which has a quite slow preprocessing (Behnke et al., 2020).

218 out of 242 problems (90.0%) have been resolved by both Lilotane and Tree-REX, and 197 problems (81.4%) have been resolved by both Lilotane and PANDA. Among the instances solved by both, on 98.2% Lilotane outperformed Tree-REX, and on 68.4% (7.3%) Lilotane outsped Tree-REX by more than one (two) order(s) of magnitude. On 99.5% (97.5% / 59.9% / 5.1%) Lilotane outperformed PANDA (by more than one / two / three orders of magnitude). Satellite is the only domain where Lilotane is slower than Tree-REX for multiple instances. This domain heavily features recursive subtask relationships. We conjecture that the grounding procedure of Tree-REX is able to simplify these recursive relationships through grounding which leads to a smaller problem to encode and search.

7.2.2 Encoding Properties

In the right graphs in Fig. 12, the number of encoded clauses is compared, providing more insight into the relative quality of our encoding while implementation-dependent performance differences are excluded. For some domains, the respective set of points resembles a line of slope $m > 1$ in log-log scale, which mathematically implies a polynomial factor in encoding size as the problem size increases. This effect occurs in a pronounced fashion for the domains Childsnack, Depots and Zenotravel in comparison to Tree-REX and for Childsnack, Transport and Gripper in comparison to PANDA. We found this observation to confirm our claim from Chapter 3.1 that grounding can lead to a severe blowup in problem size. For instance, the methods in the Childsnack domain which decompose each initial task have four free arguments. Each of these arguments has $\mathcal{O}(n)$ possible values where n defines the problem difficulty. For each initial task, Tree-REX instantiates $\mathcal{O}(n^4)$ reductions whereas Lilotane instantiates $\mathcal{O}(1)$ reductions. Both instantiate $\mathcal{O}(n)$ facts.

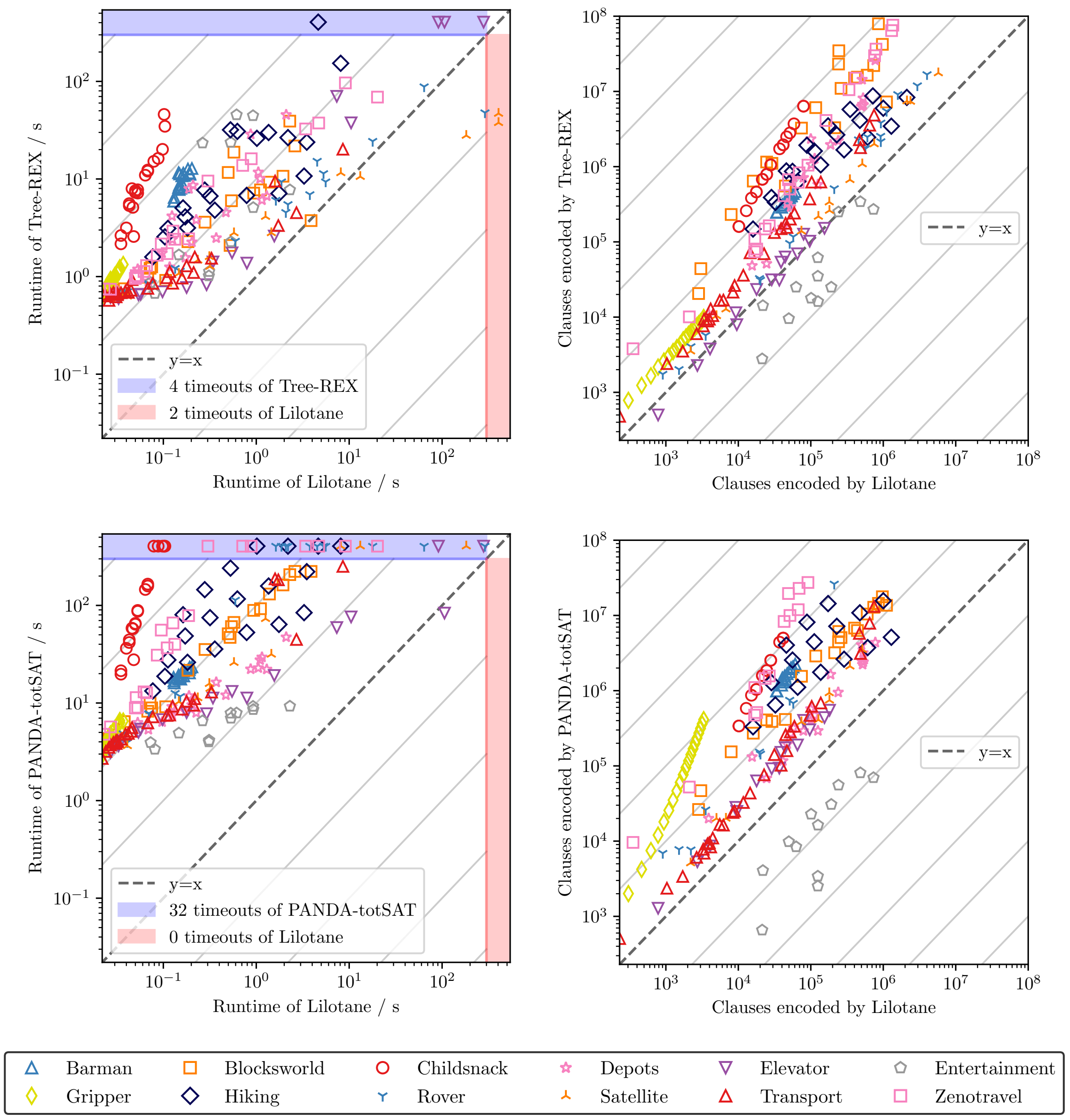


Figure 12: Direct comparison of run times and encoded clauses of Lilotane vs. PANDA-totSAT and Lilotane vs. Tree-REX. Diagonal lines denote orders of magnitude of difference.

By contrast, we do not observe any clearly exponential differences in encoding size. In the Childsnack domain in particular, the problem hierarchy has a constant depth of two, hence the described blowup in the number of child operations occurs only once before the problem is solved. More generally speaking, most domains do not have a hierarchy which expands indefinitely with respect to the density of operations (as assumed in our worst case analysis in Chapter 5.5) but instead become quite simple after few layers.

Entertainment is the only domain for which the Lilotane encoding is consistently larger – by up a factor of 50. For these instances, the grounding procedures of Tree-REX and PANDA prune large parts of search space before encoding them. Our algorithm, however, prunes these parts retroactively when they turn out to be impossible to achieve, after the clauses were already added. A comparison of run times suggests that the pruning done by disabling the respective clauses is still effective whereas PANDA and Tree-REX pay a considerable price for grounding the problem. However, these results do indicate that our lifted approach may be at a disadvantage in some particular cases where the ground problem becomes substantially smaller and simpler than the lifted representation.

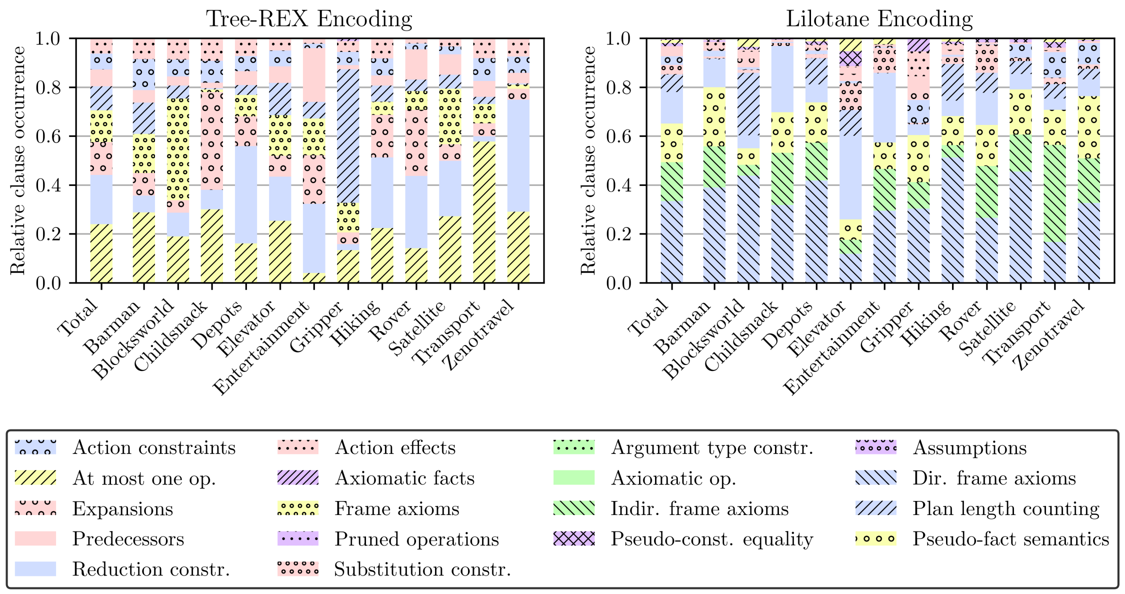


Figure 13: Distribution of occurrences of different clause categories per domain.

In order to investigate which categories of clauses are the most expensive in our encoding and how this compares to the prior Tree-REX encoding, we visualized the relative occurrence of different kinds of clauses in Fig. 13. In the Tree-REX encoding, substantially more operations are encoded due to full grounding, hence at-most-one constraints over operations are the most expensive category of clauses followed by reduction constraints (i.e. non-primitiveness and preconditions) and expansion constraints. In the Lilotane encoding, it is the frame axioms which consistently make up large parts of the encoding, which is why we split them into direct frame axioms (Eq. 14) and indirect frame axioms (Eq. 15). After frame axioms, the definitions of pseudo-facts are the next most costly clauses, followed only then by reduction constraints (which also include constraints of substitutions, Eq. 22–23). Simply put, one may say that for Tree-REX the encoding of operations is the main bottleneck and for Lilotane the encoding of (pseudo-)facts is the main bottleneck.

For the Entertainment domain, there is no clause category which is alone responsible for the much larger encoding Lilotane produces. However we do see an increased ratio of reduction constraints. Meanwhile Tree-REX encodes Entertainment problems with the lowest ratio of at-most-one constraints throughout all domains, implying a simple task network with few alternatives. These findings are consistent with our explanation regarding the Entertainment domain that Lilotane encodes an overall much larger problem as it has no access to valuable information gained from grounding.

	minimum	median	maximum	average
PANDA-totSAT	2.00	2.31	7.36	2.50
Tree-REX	2.46	2.66	12.89	3.13
Lilotane	2.25	2.95	5.56	3.15

Table 1: Distribution over average clause lengths reported per solved instance

To set the number of clauses in relation to the average size of each clause, Tab. 1 provides basic measures for the average clause length reported per instance for each approach. We can see that the median of average clause lengths of Lilotane encodings is larger than that of PANDA (Tree-REX) by 0.64 (0.29) literals on average: While the prior approaches encode large numbers of two-literal constraints between individual operations such as at-most-one constraints, the majority of clauses produced by Lilotane are related to frame axioms and pseudo-facts. Such clauses contain three or more literals as we simplify away clauses which merely enforce the equivalence between two variables (Chapter 5.2.2). As such, while Lilotane may produce clauses which are longer by up to 30% on average, it often produces fewer clauses by a factor of ten or more as displayed in Fig. 12.

	Lilotane	LilotaneQ	P-OPT	P-SAT	P-totSAT	Tree-REX
Avg. (GB)	0.128	0.099	0.931	1.083	1.069	0.790
Median (GB)	0.026	0.024	0.494	0.551	0.515	0.369

Table 2: Mean and median of memory peaks per solved instance

7.2.3 Resource Usage

We measured the amount of memory each of the planners used: As shown in Table 2, Lilotane is much more memory efficient than its competitors on average. LilotaneQ has the lowest memory footprint because we only considered completed runs – the instances solved by Lilotane but left unfinished by LilotaneQ naturally tend to be large problems.

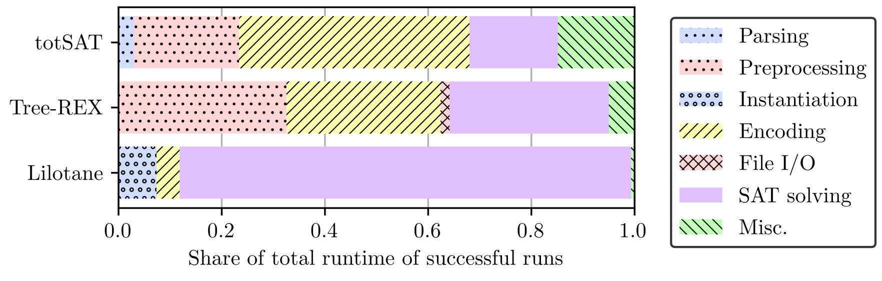


Figure 14: Partition of run times by stage

Fig. 14 shows the share on the total run time which certain stages in the planning algorithms contribute. For each competitor, we sum up the time spent in each stage over all successful runs. The stage of preprocessing encompasses any tasks related to grounding and transforming the problem into a fit form. For Lilotane there is an additional "Instantiation" stage that captures the time needed for the instantiation of hierarchical layers. The encoding stage encompasses the creation of clauses as well as, in the case of Tree-REX, the one-time creation of a schematic encoding which is later handed to a separate interpreter application. For Tree-REX there is an additional stage "File I/O" capturing the time needed to write the schematic encoding file. Finally, "Miscellaneous" refers to any portion of the run time which we could not attribute to any major stage.

While previous planners spend around two thirds of their run time on preprocessing and encoding, Lilotane spends most of its time (more than 85%) on SAT solving. Together with the improved run times compared to previous approaches, this result indicates that Lilotane successfully reduces the overhead associated with SAT-based planning as it shifts effort from expensive preprocessing to the actual search for a plan during SAT solving.

7.3 International Planning Competition 2020

In the following, we shed light on the International Planning Competition (IPC) 2020 (Behnke et al., 2020b) and the performance of Lilotane in this competitive event.

7.3.1 Benchmarks

The IPC was based on an exceptionally large and diverse set of benchmarks for hierarchical planning: In previous work the co-existence of various input formats of HTN planners hindered direct performance comparisons (e.g., Nau et al., 1999; Schreiber et al., 2019b; Höller et al., 2020), but for the IPC a de-facto standard format for hierarchical planning problems was established and many domain authors contributed a large variety of benchmarks.

Table 3 lists averaged properties of old and new benchmarks in accordance with our complexity model. On average, larger benchmarks were included than most of the previous common benchmarks from research in HTN planning. Each of the prior domains Blocksworld(-GTOHP), Childsnack, Depots, Hiking, Rover, and Satellite has been included as a benchmark in the IPC with an identical hierarchical model (as hinted by identical entries in all but the last three columns for the respective rows). For each of these domains except for Satellite, considerably larger instances have been added, as can be seen in the differences in the last three columns which represent the mean size of input problems.

The IPC benchmarks are also consistent with our assumption that the maximum arity Y_f of predicates is always a small constant (four in the IPC's Entertainment domain and at most three everywhere else) and is mostly smaller but never larger than the maximum arity of actions (Y_a) and reductions (Y_r). The latter two, by contrast, can scale linearly with problem size (see Blocksworld-HPDDL, Multiarm-Blocksworld, and Snake) due to universal quantifications in preconditions. PandaPIparser compiles these out by adding one additional argument to the operation for each constant in the quantified domain. This also leads to accordingly high numbers for the maximum number of preconditions P_a,P_r. However, no pseudo-constants are added for these artificial arguments because each encompasses a

Domain	#	O	M	X	B	U	P_a	P_r	E	Y_f	Y_a	Y_r	C	$\lvert s_I \rvert$	$\lvert T \rvert$
Barman	20	23	21	7	3	5	7	4	8	2	6	8	41	80	14
Blocksworld	20	13	10	4	2	1	4	4	5	2	2	2	24	29	24
Childsnack	20	9	2	5	2	5	5	5	5	2	5	9	77	101	16
Depots	20	17	14	4	4	3	5	4	6	2	5	5	28	45	8
Elevator	20	16	15	3	3	2	5	5	2	2	3	3	32	299	1
Entertainment	12	15	23	3	9	2	10	7	4	2	4	4	45	403	3
Gripper	20	8	4	6	2	3	3	2	3	2	3	6	27	28	12
Hiking	20	20	20	9	4	5	8	7	8	3	8	8	27	52	1
Rover	20	30	16	4	3	3	6	4	4	3	6	6	58	877	16
Satellite	20	14	12	3	3	2	5	3	3	2	4	4	121	176	79
Transport	30	6	10	4	3	2	4	1	4	2	5	5	25	43	9
Zenotravel	5	11	9	4	3	3	8	8	4	2	9	12	15	20	4
AssemblyHierarchical	30	17	20	2	8	2	9	6	3	3	7	7	116	368	1
Barman-BDI	20	33	23	6	3	5	7	7	8	2	6	6	59	89	11
Blocksworld-GTOHP	30	13	10	4	2	1	4	4	5	2	2	2	138	148	144
Blocksworld-HPDDL	30	14	14	4	5	2	200	200	5	2	200	200	200	413	1
Childsnack	30	9	2	5	2	5	5	5	5	2	5	9	262	390	62
Depots	30	17	14	4	4	3	5	4	6	2	5	5	83	150	48
Elevator-Learned	147	41	25	5	3	2	4	3	2	2	3	3	47	660	16
Entertainment	12	19	26	7	4	2	7	2	2	4	4	7	45	403	1
Factories-simple	20	17	11	4	3	3	4	3	4	3	4	4	73	98	1
Freecell-Learned	60	283	304	7	15	6	14	8	7	2	8	8	52	177	4
Hiking	30	20	20	9	4	5	8	7	8	3	8	8	44	87	1
Logistics-Learned	80	56	54	3	8	3	4	4	2	2	5	5	55	48	22
Minecraft-Player	20	21	24	6	4	3	4	4	2	3	5	16	752	217363	1
Minecraft-Regular	59	15	14	6	3	2	3	3	2	3	5	16	22150	129799	1
Monroe-Fully-Obs.	20	68	84	6	10	3	6	1	6	3	5	8	91	419	1
Monroe-Partially-Obs.	20	67	83	6	10	3	5	1	6	3	5	8	91	420	1
Multiarm-Blocksworld	74	15	15	4	5	2	54	54	5	2	54	55	58	118	4
Robot	20	6	13	2	4	3	3	3	3	3	3	3	60	97	1
Rover-GTOHP	30	30	16	4	3	3	6	4	4	3	6	6	95	2351	28
Satellite-GTOHP	20	14	12	3	3	2	5	3	3	2	4	4	121	176	79
Snake	20	7	5	3	3	3	28	28	8	3	28	28	29	149	1
Towers	20	7	10	2	3	3	4	3	6	2	5	5	14	111	1
Transport	40	4	6	4	3	2	4	0	4	2	5	5	55	125	26
Woodworking	30	15	51	3	4	4	9	0	7	2	12	12	110	241	22

Table 3: Averaged per-domain properties of HDDL benchmarks (after preprocessing), divided into prior benchmarks (see Chapter 7.2) and IPC benchmarks. Left to right: Number of operators, methods; max. expansion size, max. methods per task, max. free non-trivial arguments per method; number of preconditions of actions / reductions, effects; arity of facts / actions / reductions; number of constants / initial true facts / initial tasks.

domain of size one. For the computation of the maximum number of unbound arguments U, we only considered non-trivial free arguments with a domain of size greater than one.

7.3.2 Rules and Participants

Planners have been rated according to the following metric: If a planner solves an instance within one second, a score of 1 is attributed. If a planner solves an instance within $1 < t \leq T$ seconds (where $T = 30$ min is the time limit), a score of $1 - \log(t)/\log(T)$ is attributed.

This so-called agile metric has several consequences: Planners which find plans very quickly are favored over slower but more robust planners. For instance, if a planner solves five instances in 2 seconds while not solving five other instances, it is attributed a score of around $5 \cdot 0.91 = 4.55$. If a planner solves each of the ten instances in two minutes, it is attributed a score of around $10 \cdot 0.36 = 3.6$. Furthermore, plan quality is ignored.

The competition consisted of two tracks: Total Order and Partial Order. Six planners were submitted to the Total Order track while only three planners were submitted to the Partial Order track (one of which was disqualified). Among these six planners are the preliminary version of Lilotane, the lifted progression search planner HyperTensioN, the ground progression search planner PDDL4J in a Total Order and a Partial Order version, the lifted progression search planner SIADEX, and the plan-space planner pyHiPOP. All planners are described in the IPC proceedings (Behnke et al., 2020b).

The following improvements were not part of the IPC version of Lilotane: Our optional anytime plan improvement procedure (Chapter 6); various techniques for reducing clauses and distinct variables (Chapter 5.2.2); retroactive pruning which not only logically disables the pruned operation but also prunes the subtree relying on it (Chapter 4.2.3); and sharing pseudo-constants among operations and eliminating dominated operations (Chapter 4.4).

	HyperT.		Lilotane		P4JTO		P4JPO		SIADEX		pyHiPOP	
Domain	NC	IPC	NC	IPC	NC	IPC	NC	IPC	NC	IPC	NC	IPC
AssemblyHierarchical	0.10	0.08	**0.17**	**0.12**	0.07	0.06	0.07	0.06	0.00	0.00	0.03	0.02
Barman-BDI	**1.00**	**1.00**	0.80	0.74	0.55	0.48	0.55	0.49	**1.00**	0.92	0.00	0.00
Blocksworld-GTOHP	0.53	0.43	**0.77**	**0.64**	0.53	0.41	0.57	0.43	0.47	0.35	0.03	0.01
Blocksworld-HPDDL	**1.00**	**0.89**	0.03	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Childsnack	**1.00**	**1.00**	0.97	0.87	0.70	0.46	0.70	0.47	0.73	0.50	0.00	0.00
Depots	**0.80**	**0.76**	**0.80**	0.73	0.77	0.57	0.77	0.60	0.73	0.70	0.00	0.00
Elevator-Learned	**1.00**	**1.00**	**1.00**	0.78	0.01	0.01	0.01	0.01	0.07	0.07	0.01	0.01
Entertainment	0.00	0.00	**0.42**	0.14	**0.42**	0.19	0.25	**0.27**	0.00	0.00	0.08	0.07
Factories-simple	0.15	0.14	**0.20**	**0.19**	0.00	0.00	0.00	0.00	0.00	0.00	0.05	0.01
Freecell-Learned	0.00	0.00	**0.15**	**0.05**	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Hiking	**0.83**	**0.83**	0.73	0.60	0.57	0.32	0.50	0.39	0.00	0.00	0.00	0.00
Logistics-Learned	0.28	0.26	**0.55**	**0.32**	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Minecraft-Regular	**0.98**	**0.88**	0.54	0.35	0.39	0.32	0.39	0.32	0.59	0.33	0.00	0.00
Minecraft-Player	**0.25**	**0.25**	0.05	0.03	0.05	0.03	0.05	0.03	0.15	0.13	0.00	0.00
Monroe-Fully-Obs.	0.00	0.00	**1.00**	**0.78**	**1.00**	0.49	**1.00**	0.58	0.50	0.27	0.00	0.00
Monroe-Partially-Obs.	0.00	0.00	**1.00**	**0.73**	0.05	0.03	0.05	0.03	0.00	0.00	0.00	0.00
Multiarm-Blocksworld	**0.11**	**0.11**	0.05	0.03	0.00	0.00	0.00	0.00	0.01	0.01	0.00	0.00
Robot	**1.00**	**0.96**	0.55	0.52	0.30	0.27	0.30	0.27	0.00	0.00	0.05	0.05
Rover-GTOHP	**1.00**	**0.92**	0.77	0.55	**1.00**	0.60	0.87	0.65	**1.00**	0.77	0.20	0.14
Satellite-GTOHP	**1.00**	**1.00**	0.75	0.59	**1.00**	0.44	0.50	0.73	0.00	0.00	0.35	0.19
Snake	**1.00**	**1.00**	0.90	0.74	**1.00**	0.71	**1.00**	0.71	0.35	0.29	0.10	0.03
Towers	**0.85**	**0.77**	0.50	0.39	0.80	0.58	0.75	0.61	0.55	0.47	0.10	0.09
Transport	**1.00**	**1.00**	0.88	0.76	0.85	0.65	0.82	0.71	0.03	0.03	0.45	0.23
Woodworking	0.23	0.23	**1.00**	**0.98**	0.20	0.17	0.20	0.17	0.10	0.10	0.13	0.09
Total rel. score	14.12	**13.51**	**14.57**	11.60	10.25	7.47	9.35	6.36	6.29	4.93	1.6	0.94

Table 4: Central results of the IPC 2020. NC = normalized coverage, IPC = agile runtime score. Higher is better, best scores per line are printed in bold for each metric.

7.3.3 Results

The results of the IPC evaluation are given in Tab. 4. In addition to the IPC score explained above, we included a second important metric named coverage. This simple metric leads to a score of 1 for a solved instance and a score of 0 for an unsolved instance. As each run was repeated ten times, we count an instance as solved if it was solved in any of the runs. We normalized each coverage score per domain by the number of instances in a domain (e.g., a normalized coverage score of 0.6 means that 60% of all instances within the domain were solved). The IPC scores were normalized in the same way.

Regarding IPC scores, which decided the official ranking, Lilotane scored second, behind HyperTensioN by a decent margin. All further competitors scored significantly lower: Notably, Lilotane outperformed ground approach PDDL4J on all but four domains (Entertainment, Minecraft-Player, Rover, and Towers). HyperTensioN scored best on 15/24 domains and Lilotane scored best on 8/24 domains; only a single domain (Entertainment) was neither won by HyperTensioN nor by Lilotane. Lilotane's worst performances are on the domains Blocksworld-HPDDL, Minecraft(-Player), and Multiarm-Blocksworld. We noticed that each of these domains leads to deep and large hierarchical task networks which favor greedy progression search planners over planners such as Lilotane which are required to instantiate the entire hierarchy with all alternatives up to the layer where a plan can be found. Furthermore, as discussed above, due to compiled universal quantifications, Blocksworld-HPDDL and Multiarm-Blocksworld feature many preconditions per operator which are comparably costly for our encoding. By contrast, our planner excels on domains such as Monroe (complex goal and task recognition problems; see Blaylock & Allen, 2005; Höller, Behnke, Bercher, & Biundo, 2018) and Woodworking (large manufacturing and processing tasks with a high number of arguments per operator and method).

Regarding coverage, Lilotane solved three more instances (548) than HyperTensioN (545) and scored slightly better. Yet, Lilotane solved 14 of these instances only in some of the runs, while HyperTensioN solved each of its instances consistently with one exception. Still, overall we observe that while the agile score benefits the very fast execution times of HyperTensioN, Lilotane performs similarly to HyperTensioN in terms of robustness and, unlike HyperTensioN, is able to solve some instance(s) on every single domain.

7.4 Follow-Up Evaluation

We now head on to our own evaluation based on the benchmarks of the IPC in order to account for some aspects of our planner which the IPC did not cover yet. First, we improved Lilotane in various aspects after the submission deadline of the IPC so we expect better results with our final planner than what the competition version already achieved[3]. Secondly, as solution quality did not matter in the IPC, we also want to evaluate the quality of our approach with and without plan improvement on a large set of benchmarks.

As PDDL4J, SIADEX, and pyHiPOP were mostly dominated regarding both IPC scores and coverage, we do not include them in the following evaluations. We do include the winner HyperTensioN and different versions of Lilotane: (i) Prelilotane (the preliminary version submitted to the IPC), (ii) Lilotane (the up-to-date version without quality awareness),

3. Note that we integrated each of these improvements before gaining access to the benchmarks of the IPC and only applied bugfixes thereafter, hence no fine-tuning with respect to the benchmarks was done.

(iii) Lilotane-Q (a quality-aware configuration which finds the depth-optimal plan at the first layer where a plan is found), and (iv) Lilotane-Q+ (a configuration which instantiates one extra layer after finding an initial plan and then finds the depth-optimal plan). For approaches (iii) and (iv), unfinished runs where a valid but not necessarily fully optimized plan was output on termination are considered unsolved.

The evaluations were conducted on an server with an AMD EPYC 7702P 64-Core processor (plus hyperthreading) clocked between 2.0 and 3.35 GHz with 1024 GB of DDR4 RAM, running Ubuntu 20.04. We executed up to 63 runs in parallel and set a time limit of 30 minutes and a memory limit of 8GB as in the IPC.

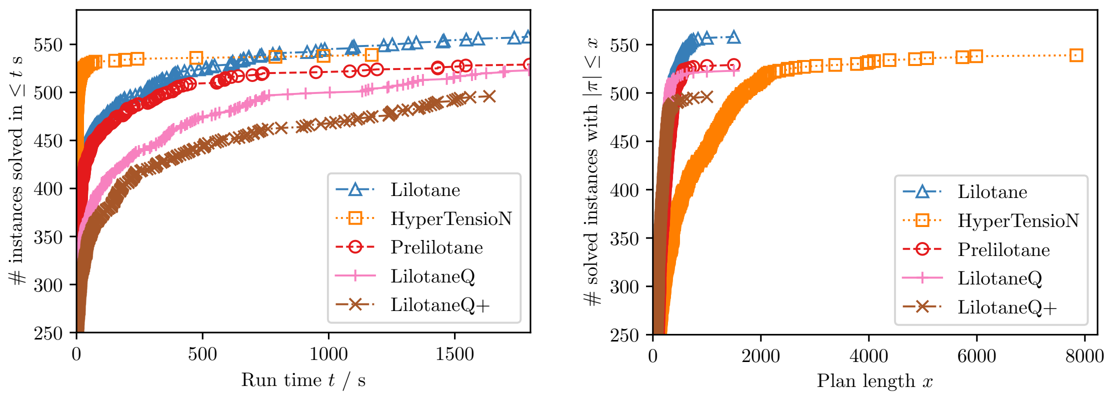


Figure 15: Run times and found plan lengths of HyperTensioN and Lilotane

7.4.1 Overview

A first overview on the results is provided in Fig. 15. In Appendix C, detailed per-domain plots are provided in Fig. 17–18, and more details on Lilotane's behavior split by domain and by algorithm stage are provided in Fig. 19–20 and in Tab. 6.

HyperTensioN solved 539 out of 892 (60.4%) instances and Prelilotane solved 529 instances (59.3%). The lower coverages compared to the IPC data provided above can be explained by (a) different hardware and (b) the fact that we performed only one run for each competitor-instance combination. HyperTensioN retains its status being fastest on the majority of benchmarks. However, up-to-date Lilotane solved 558 instances (62.6%) making it more robust in the long run. LilotaneQ finished its plan improvement on 523 instances and LilotaneQ+ finished on 496 instances. In other words, the quality-aware configurations of Lilotane found a depth-optimal plan at the first solvable layer on 93.7% of the solved instances and a depth-optimal plan at the subsequent layer on 88.9%.

Concerning the planners' coverage in conjunction with found plan quality, Lilotane outperforms HyperTensioN. The plans output by HyperTensioN, although found rapidly, are longer than the plans found by Lilotane even without employing plan improvement. This is because Lilotane always finds a plan at a point where the hierarchy is as shallow as possible, which strongly correlates with the potential length of plans. By contrast, HyperTensioN performs a kind of depth-first search and hence traverses search space more greedily, resulting in larger plans.

Domain	# slv.	HyperTensioN	Prelilotane	Lilotane	LilotaneQ	LilotaneQ+
AssemblyHierarchical	3	0.80	1.00	1.00	1.00	1.00
Barman-BDI	16	0.55	0.64	0.63	0.92	1.00
Blocksworld-GTOHP	16	0.87	0.89	0.92	1.00	1.00
Blocksworld-HPDDL	1	0.91	1.00	1.00	1.00	1.00
Childsnack	29	1.00	1.00	1.00	1.00	1.00
Depots	23	1.00	0.96	0.97	1.00	1.00
Elevator-Learned	144	0.35	0.83	0.83	1.00	0.98
Factories-simple	3	0.58	0.62	0.67	1.00	1.00
Hiking	21	0.98	0.96	0.96	1.00	1.00
Logistics-Learned	22	0.78	0.72	0.71	1.00	1.00
Minecraft-Player	1	1.00	0.85	0.74	1.00	1.00
Minecraft-Regular	28	1.00	0.85	0.87	1.00	1.00
Multiarm-Blocksworld	4	0.91	0.82	0.85	1.00	1.00
Robot	11	0.51	1.00	1.00	1.00	1.00
Rover-GTOHP	20	0.51	0.69	0.69	0.99	0.96
Satellite-GTOHP	13	0.50	0.58	0.60	1.00	0.99
Snake	17	0.28	0.92	0.89	1.00	1.00
Towers	9	1.00	1.00	1.00	1.00	1.00
Transport	33	0.73	0.75	0.74	1.00	0.99
Woodworking	7	0.94	0.93	0.95	0.98	0.98
Total	421	15.20	17.01	17.02	19.89	19.90

Table 5: Normalized plan length scores over all IPC instances for which HyperTensioN and all configurations of Lilotane found some plan.

7.4.2 Plan Quality

Table 5 provides more insights into the plan quality of the involved approaches. We make use of satisficing IPC scores: If π is the shortest plan found by some algorithm for some instance, a score of $|\pi|^*/|\pi|$ is attributed where π^* is some reference plan. In our case, π^* is the best plan found by any of the competitors for this instance. To exclude coverage results, we only considered the instances for which each competitor found some (not necessarily final) plan. For each competitor we summed up the scores within a domain and then normalized the result by the number of solved instances in that domain.

Computed over all 421 such instances, all configurations of Lilotane outperform HyperTensioN with respect to this metric. Unsurprisingly, Prelilotane and Lilotane achieve very similar results: None of the employed post-IPC improvements impact plan lengths.

Our plan improvement procedure in LilotaneQ improves upon Lilotane's score by almost three points. The degree of improvement heavily depends on the domain model. For instance, in the Factories domain, which truck to use and where to construct which factory is up to the planner. This makes the domain well-suited for evaluating quality-aware planning (Sönnichsen & Schreiber, 2020). By contrast, other domains such as Childsnack or Towers feature a rigid hierarchy and leave no room for plan improvement.

Another interesting result is how few improvement LilotaneQ+ brings over LilotaneQ. Only on 17 instances from six domains did we observe that instantiating an additional layer before performing plan improvement leads to a shorter final plan within time and memory constraints, and the only domain where this difference is reflected in the above scores is Barman-BDI. Also note that LilotaneQ+ occasionally scores lower than LilotaneQ: This is because plan improvement on the first solvable layer can be much easier to perform than on the subsequent layer, especially if the layers' complexity grows considerably. Still, among all IPC instances, LilotaneQ+ finished on a total of 477 instances without any improvement over LilotaneQ. We deduce that depth-optimal plans found at the first solvable layer are rarely improvable any further when the hierarchy is extended by another layer. We expect the improvement induced by deepening the hierarchy to further diminish if we instantiate even more layers, leading us to the conclusion that, in practice, employing LilotaneQ provides a good tradeoff between high plan quality and acceptable run times and often even finds plans of optimal or near-optimal quality.

7.5 Discussion

From the presented evaluations involving other SAT-based planners, we conclude that our grounding-free approach of encoding the lifted problem representation directly into propositional logic is highly efficient and generally leads to the smallest propositional logic formulae of any SAT-based HTN approach to date. While there are planning domains where *a priori* grounding significantly reduces the effective size of the problem to encode, these merits can be outweighed by Lilotane's faster planning procedure and lower overhead. Furthermore, Lilotane is significantly more memory-efficient than previous SAT-based planners. All in all, we are confident that Lilotane is the most efficient SAT-based TOHTN planner to date.

From the results of the IPC we observed that Lilotane compares very favorably to the submitted grounding-based approaches. In comparison with the arguably best TOHTN planner to date, HyperTensioN, the results are less one-sided. Lilotane is more robust than HyperTensioN if sufficient time is available, but often takes more time than the ultimately more lightweight greedy progression search planner. We believe that this may be an intrinsic property of SAT-based approaches compared to progression search approaches: Some planning problems are very easy to resolve greedily such that the detour over propositional logic imposes unnecessary overhead. However, investing additional time to solve an instance with Lilotane is worthwhile because the found plans are of high quality, even with a configuration that does not perform any plan improvement. As such, we consider HyperTensioN and Lilotane to correspond to two different points on the pareto-frontier between speed, robustness, and quality in the current state-of-the-art in TOHTN planning.

8. Conclusion

We have presented an approach of grounding-free SAT-based TOHTN planning motivated by the combinatorial blowup which grounding can induce. To process the lifted problem representation, we proposed a lazy instantiation approach coupled with a reachability analysis and the introduction of non-committal *pseudo-constants* when we encounter free method arguments. We presented an according SAT encoding and showed its correctness. We performed a worst-case analysis where we found that our encoding is exponential along fewer dimensions than the prior Tree-REX encoding but introduces more complex logic related to the facts in the problem. We enhanced an existing anytime plan improvement procedure to make Lilotane quality-aware. We presented evaluations which suggest that Lilotane outperforms existing SAT-based TOHTN approaches, produces the smallest SAT encodings for TOHTN problems to date, and has a lower memory footprint. While often outperformed by state-of-the-art TOHTN planner HyperTensioN in terms of run times, Lilotane excels regarding its robustness and the high-quality plans it finds.

8.1 Outlook

In the near future, we expect our approach and its implementation to find adoption as an efficient and reliable TOHTN planner. Our lifted encoding approach may influence neighboring areas of research, most naturally general HTN planning as well as further extensions such as HTN planning with task insertion (Geier & Bercher, 2011). Furthermore, some of the insights we gained may be transferable to lifted SAT-based classical planning.

From a theoretical perspective, we are interested in investigating more abstract encodings which encode every single argument in the task network on a symbolic level. While we are unsure about the practical efficiency of such an approach, we believe that we may find an encoding which encodes a maximum of $|O| + |M|$ operations at each position and whose complexity in relation to the lifted problem description does not feature any superquadratic terms except for the unavoidable exponential growth of layer sizes.

It remains to be seen whether both lifted and ground SAT-based HTN planning approaches remain significant and are refined further, excelling on different kinds of planning problems. As we have seen in our evaluations, there are some planning domains where grounding a problem brings great advantage while on some other domains it is next to infeasible to do so. We believe that our reachability analysis can be improved further by computing a better approximation of the possible fact changes an operation may effect. This could lead to much earlier and more effective pruning of irrelevant operations and may render Lilotane more competitive regarding the domains where grounding is highly effective. It is an interesting question for future research whether the “best of both worlds” can be achieved by creating an efficient instantiation approach which keeps the problem lifted but effectively performs the same *a priori* pruning that is achieved by high quality grounding.

Last but not least, having circumvented a major scaling problem of prior approaches, we intend for future work to improve the scalability of TOHTN planning even further by exploring possible approaches to parallelize Lilotane. We believe that an efficient parallel and distributed hierarchical planner will not only do justice to the true (multi-core and cloud-like) nature of today’s computing but, with the help of distributed SAT solving, will also be able to resolve much larger problems than any prior HTN planning system.

Acknowledgments

The author would like to thank the reviewers for providing valuable feedback and criticism which significantly improved the publication, and Markus Iser, Simon Reiß, Ekkehard Schreiber, Nico Wildermuth, and Marvin Williams for proofreading. Furthermore, many thanks to Peter Sanders for his encouraging support as thesis advisor; to the organizers of the IPC 2020 for making this important competitive event possible; and to Gregor Behnke for fruitful discussions regarding the results of the IPC 2020. Last but not least, many thanks to Damien Pellier, Humbert Fiorino, and Tomáš Balyo who introduced the author to the exciting topics of SAT-based planning and HTN planning.

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agr. No. 882500).

Appendix A. Proof of Correctness

Theorem 1. *For some l and $x \geq 0$, let $O_l := \langle o_0, \dots, o_x \rangle$ be a sequence of operations where o_i is chosen among all possible operations at position $P_{l,i}$ of some problem Π. Expand each $o_i \in O_l$ into some sequence $\mathcal{O}_i$ of actions such that $\mathcal{O} := \mathcal{O}_0 \circ \ldots \circ \mathcal{O}_x$ is executable from s_I. (1) If fact f holds after the execution of $\mathcal{O}$, then f is reachable at $P_{l,x+1}$ according to S_l^{x+1}. (2) Similarly, if f does not hold after executing $\mathcal{O}$, then $\neg f$ is reachable at $P_{l,x+1}$.*

Proof. First, we note that the sets $\pm S_l^{x+1}$ grow monotonically in x. This observation directly follows from their definition.

(1) If f holds after executing $\mathcal{O}$, then either (i) $f \in s_I$ and f never changed, or (ii) the execution of $\mathcal{O}$ causes f as an effect in some action. In case (i), f is reachable at $P_{l,x+1}$ by definition. In case (ii), there is some action o_f which causes f as a direct effect, implying $f \in \mathit{pfc}(o_f)$ and either (a) $o_f = o_j$ for $0 \leq j \leq x$ or (b) o_f is a (transitive) child of one such o_j. In case (a), $\mathit{PFC}_{l,j+1}^{+} \supseteq \mathit{pfc}(o_f) \ni f$ by definition, and in case (b), $\mathit{PFC}_{l,j+1}^{+} \supseteq \mathit{pfc}(o_j) \supseteq \mathit{pfc}(o_f) \ni f$ because o_f is a child of o_j. In both cases (a) and (b) $\mathit{PFC}_{l,j+1}^{+}$ is added to $+S_l^{j+1}$ and, since $j \leq x$ and $+S_l^x$ grows monotonically in x, we obtain $f \in +S_l^{x+1}$, hence f is reachable according to S_l^{x+1}.

(2) If f does not hold after executing $\mathcal{O}$, then either (i) $f \notin s_I$ and f never changed, or (ii) as (1)(ii) but with a negative effect. In case (i), $f \notin +S_l^{x+1}$ follows because f was never added to $+S_l^{x+1}$. Consequently, $f \notin s_I \cup +S_l^{x+1}$ and hence $\neg f$ is reachable. In case (ii), similar to (1)(ii) we obtain $f \in \mathit{PFC}_{l,x+1}^{-}$ and consequently $f \in -S_l^{j+1}$. Due to the monotonicity of $-S_l^x$, we obtain $f \in -S_l^{x+1}$, hence $\neg f$ is reachable. □

Lemma 1. *For any satisfying assignment $\mathcal{A}$ for $\mathcal{L}_{l'}(\Pi)$ there is exactly one active operation at each position $P_{l,x}$ for $l \in \{0, \dots, l'\}$ and $x \in \{0, \dots, |L_l| - 1\}$.*

Proof. At position $P_{0,0}$ exactly the initial reduction is active due to construction and Eq. 1. At each further position $P_{l,x}$ at most one action and at most one reduction is active due to Eq. 4 which, together with Eq. 3, ensures that at most one operation is active. Also, there is at least one active operation at each position: If there were not a single active operation at some position, then, due to Eq. 8 and our instantiation technique where all offset positions are filled with children (possibly with ε-actions), the parent position would be empty as well. Repeatedly applying this argument leads to a contradiction to Eq. 1. □

Lemma 2. *The result π of decoding a plan from a satisfying assignment $\mathcal{A}$ for $\mathcal{L}_{l'}(\Pi)$ as described in Chapter 5.3 is well-defined and unambiguous.*

Proof. First we observe that at the final layer l' there must be an action (and no reduction) at every position due to Eq. 10 together with Lemma 1.

Next we argue that the substitution of pseudo-constants with actual constants in any active operation o is unambiguous. For each pseudo-constant κ, due to Eq. 11 at most one substitution can be active. Furthermore, if the operation o_p from which κ originated is active, then due to Eq. 12 each pseudo-constant κ must have at least one and thus exactly one constant substituted for it. The only possibility for o to be active yet o_p to be inactive is that o came into effect by dominating some other operation o' which has another parent and does not contain κ. In that case, Eq. 20 will enforce at least one substitution of κ to be active. Otherwise, by construction κ can only occur in the hierarchy induced by the origin operation of κ. Hence, for each active operation in a solution there is exactly one set of active substitutions which replace each pseudo-constant with actual constants.

From these observations and Lemma 1 we conclude that there is exactly one operation at each spot in the hierarchy with exactly one ground representation each, and the final layer only consists of actions. This renders our plan decoding well-defined and unambiguous. $\square$

Lemma 3. *Let $\mathcal{A}$ be a satisfying assignment for $\mathcal{L}_{l'}(\Pi)$. Then for each position $P_{l,x}$ we can unambiguously infer a world state $s_{l,x}$ based on $\mathcal{A}$ and s_I.*

Proof. At each position $P_{l,x}$ there is a set of variables f_x^l which represent ground facts. Each such fact f has a polarity assigned to it by $\mathcal{A}$. Additionally, some positive facts may not have been encoded (yet) at $P_{l,x}$ but must hold nevertheless: These are exactly the unencoded facts which are contained in the initial state (see Chapter 5.2.1). Consequently we define $s_{l,x} := \{f \mid \mathcal{A}(f_x^l) = \texttt{true}\} \cup \{f \in s_I \mid \mathcal{A}(f_x^l) = \perp\}$. $\square$

Lemma 4. *Consider a satisfying assignment $\mathcal{A}$ for $\mathcal{L}_{l'}(\Pi)$ and an active operation o at position $P_{l,x}$ which becomes a ground operation $\tilde{o}$ through zero or more active substitutions. Then the following holds:*
(i) $pre(\tilde{o})^+ \subseteq s_{l,x}$ and $pre(\tilde{o})^- \cap s_{l,x} = \emptyset$.
(ii) If $\tilde{o}$ is an action, then $s_{l,x+1} = (s_{l,x} \setminus eff(\tilde{o})^-) \cup eff(\tilde{o})^+$.

Proof. Eq. 5 and Eq. 6 enforce that each precondition of o holds at position $P_{l,x}$ and (if o is an action) each effect of o holds at position $P_{l,x+1}$; however, each precondition or effect may contain pseudo-constants. Eq. 18 and Eq. 19 ensure that the effects are enforced in a consistent way: If some (possibly empty) set of active substitutions unifies a pair of effects to be contradictory, i.e., $\{f, \neg f\} \subseteq pre(\tilde{o})$ for some f, then the positive effect is enforced as usual while Eq. 18 and Eq. 19 allow the negative effect to be ignored because an appropriate set of substitutions is active.

As $\tilde{o}$ was obtained from o by substituting each of its pseudo-constants κ with the unique substitution $[\kappa/c]$ that is active in $\mathcal{A}$, Eq. 13 enforces that any non-ground preconditions and effects of o are logically equivalent to the respective ground preconditions and effects of $\tilde{o}$ – as long as these ground facts are indeed encoded. Some ground precondition (effect) f arising from a precondition (effect) of o through active substitutions may not have been encoded because f is invariant there. In case of an effect, f must be invariantly true because due to construction an effect cannot be invariantly false. In case of a precondition, Eq. 22 or Eq. 23 prevent any combination of substitutions which unify a precondition of o with an invariantly false fact, hence f must be invariantly true as well. It follows from the correctness of our reachability analysis (Theorem 1) that f holds in $s_{l,x}$. Hence, $s_{l,x}$ and $s_{l,x+1}$ induced by assignment $\mathcal{A}$ are consistent with the constraints of $\tilde{o}$ at $P_{l,x}$ and $P_{l,x+1}$.

If o is an action, then any fact *not* featured as an effect does not change, following from our frame axioms: As the position is primitive due to Eq. 3 if $\tilde{o}$ is an action, Eq. 14 constrains each fact to change only if an action from its direct or indirect support is active. Hence, for each such fact f that changes its polarity, either $o \in \mathit{supp}(f)$ or $o \in \mathit{isupp}(f)$. In the former case, f is a direct effect of o and it follows directly that f is also a direct effect of $\tilde{o}$. In the latter case, Eq. 15 implies that a set of substitutions must be active which unify a pseudo-fact effect of o with f. As we know that the active substitutions for the pseudo-constants of o unify o with $\tilde{o}$, we also know that the respective pseudo-fact effect of o must be an effect of $\tilde{o}$ as well. □

Lemma 5. *When a plan $\pi = \langle a_0, \dots, a_{k-1} \rangle$ is decoded from a valid satisfying assignment $\mathcal{A}$ for $\mathcal{L}_{l'}(\Pi)$, there is a sequence of states $Q = \langle s_0 := s_I, s_1, \dots, s_k \rangle$ such that $\mathit{pre}(a_i)^+ \subseteq s_i$, $\mathit{pre}(a_i)^- \cap s_i = \emptyset$, and $s_{i+1} = (s_i \setminus \mathit{pre}(a_i)^-) \cup \mathit{pre}(a_i)^+$ hold for $0 \leq i < k$.*

Proof. We construct the sequence of states Q from π and the states $s_{l',x}$ induced by $\mathcal{A}$.

When plan π contains k actions, the size of the final layer, $k' := |L_{l'}|$, may be larger than k: Layer $L_{l'}$ contains $k' - k \geq 0$ ε-actions which do not contribute to π and which have no preconditions or effects. We inductively construct Q for $k' \geq 0$.

For the base case $k' = 0$ and $\pi = \langle\rangle$, we note that at position 0 all fact assignments must be consistent with the initial state s_I due to construction: When a fact is introduced at some position, then its assignment is fixed according to the initial state (Eq. 2 or Eq. 21). Consequently $Q := \langle s_I \rangle$ fulfils above requirements.

Let $k' > 0$. By induction, for some $k < k'$ we have a valid sequence of states $Q_{k'-1} := \langle s_0, \dots, s_k \rangle$ for $\pi_{k'-1} := \langle a_0, \dots, a_{k-1} \rangle$ decoded from positions $0, \dots, k'-1$, and we want to construct $Q_{k'}$ for $\pi_{k'}$ decoded from positions $0, \dots, k'$. There is exactly one action active at position k' (see Lemma 2) which is either an ε-action or a normal action. In the former case, we note by trivially applying Lemma 4 that $s_{l',k'} = s_{l',k'-1}$, i.e., no fact changes are possible over the course of this position, so $Q_{k'} := Q_{k'-1}$ fulfils above requirements for the unchanged plan $\pi_{k'} = \pi_{k'-1}$. In the latter case, we have $\pi_{k'} = \pi_{k'-1} \circ \langle \tilde{a} \rangle$, where action $\tilde{a}$ was constructed from the active action a at position k' through a set of substitutions. We apply Lemma 4 and obtain $\mathit{pre}(\tilde{a})^+ \subseteq s_{l',k'}$, $\mathit{pre}(\tilde{a})^- \cap s_{l',k'} = \emptyset$, and $s_{l',k'+1} = (s_{l',k'} \setminus \mathit{pre}(\tilde{a})^-) \cup \mathit{pre}(\tilde{a})^+$. As a result, setting $Q_{k'} := Q_{k'-1} \circ \langle s_{l',k'+1} \rangle$ fulfils above requirements and concludes the inductive construction of Q. □

Lemma 6. *Let H be the decoded hierarchical solution for $\mathcal{L}_{l'}(\Pi)$. Then the structure of H resembles an actual hierarchical solution, i.e., H satisfies (1) and (2) from Def. 2.*

Proof. First we note that the structure of H resembles the structure of hierarchical layers, i.e., H is a tree where each node (o, l, x) has depth l within H and each of its outgoing edges lead to nodes of depth $l + 1$. We perform an induction over the maximum depth $l' \geq 0$ of H counted from its root $(r_0, 0, 0)$ and show that each node (o, l, x) corresponds to a valid operation o and, if o is a reduction, either resides at the maximum depth l' or has outgoing edges to valid child nodes in accordance to *subtasks*(o). All leaves being actions then directly follows from the well-definedness of the plan decoding procedure (Lemma 2), notably from only actions being active at the final layer l'.

Let $l' = 0$. Then the only node in the graph is $(r_0, 0, 0)$ where r_0 is fully ground and a valid reduction of the problem by definition.

Let $l' > 0$. Assume that H up to layer $L_{l'-1}$ fulfills each of the Lemma's requirements. We first show that each node at layer $L_{l'-1}$ has the correct number of children at layer $L_{l'}$. Let $(o, l'-1, x)$ be a node at layer $L_{l'-1}$. If o is an action, then the node has no children by construction. In the following, let o be a reduction. By construction, node $(o, l'-1, x)$ has $n \geq 0$ outgoing edges to nodes $(o'_k, l', s_{l'-1}(x) + z_k)$ where each z_k is a valid offset for reduction o: $0 \leq z_k < e_{l'-1,x}$. Each o'_k is a ground instantiation of some child action or reduction of o at offset z_k. For each $0 \leq z_k < |\mathit{subtasks}(o)|$ there must be exactly one such child node due to Eq. 8 and Lemma 1, and for $z_k \geq |\mathit{subtasks}(o)|$ there are no such children because by construction o induces ε-actions as children at those offsets which are never added to H. As a consequence the given parent node has the correct number of children.

Next we show for $0 \leq k < |\mathit{subtasks}(o)|$ that o'_k from child node $(o'_k, l', s_{l'-1}(x) + k)$ matches the k-th subtask of o. Operation o'_k was constructed from the original active operation $\hat{o}'_k$ at position $s_{l'-1}(x)+k$ of layer l' by a series of zero or more active substitutions. We know from Eq. 8 and Eq. 9 that $\hat{o}'_k$ matches the k-th subtask of its parent $\hat{o}$ from which o was constructed, the only exception being differing pseudo-constant names if the original child of $\hat{o}$ was dominated by another operation (see Chapter 4.4). For each substitution $[\kappa/c]$ involved in the transformation of $\hat{o}'_k$ into o'_k, pseudo-constant κ originated either (a) from $\hat{o}'_k$ itself or (b) from parent $\hat{o}$ or some common ancestor or (c) from (a parent of) an operation which dominated the original child of $\hat{o}$. In case (a) we know that the argument of $\hat{o}'_k$ which took κ as its value is a new, free argument not bound by $\hat{o}$. In case (b) we know that κ is globally substituted with the same single constant c. In case (c), Eq. 20 sets κ equivalent to the original pseudo-constant, hence (a) or (b) applies. In all cases the arguments of $\hat{o}'_k$ must correspond to the arguments of the k-th subtask of $\hat{o}$, hence o'_k matches the respective subtask of o.

Concerning argument types, the constant c which κ is substituted with is in the valid domain of the respective argument of its origin operation due to Eq. 12. Furthermore, Eq. 16 and/or Eq. 17 enforce any further restrictions to the type of κ in child operations.

All in all, o'_k matches the k-th subtask of o: edge $((o, l'-1, x), (o'_k, l', s_{l'-1}(x) + k))$ represents a valid subtask instantiation o'_k of reduction o. $\square$

Lemma 7. *Let (π, H) be the decoded (classical and hierarchical) solution for the Lilotane encoding $\mathcal{L}_{l'}(\Pi)$ for TOHTN planning problem Π. Traverse H as described in (3) in Def. 2 with the node ordering relation $(g, l, x) \prec (g', l', x') \leftrightarrow x < x'$ and maintain a state s which is initialized as s_I and updated with the effects of each visited action. Then the preconditions of all visited actions and reductions hold in s.*

Proof. When performing a depth-first traversal of H as specified, we traverse all action nodes in exactly the order in which the respective actions occur in π. As changes to s are made only when encountering an action node, it follows from Lemma 5 that whenever we visit an action node (a, l, x), s is equivalent to the state $s_{l,x}$ that can be inferred from $\mathcal{A}$ at position $P_{l,x}$. In particular, the preconditions of each visited action are met in s.

In the following, we show that when we visit a reduction node (r, l, x) during the traversal of H, the preconditions of r must hold in s. We know that r was constructed from some active reduction $\hat{r}$ and a set of substitutions, hence Lemma 4 implies that the preconditions of r hold in the state $s_{l,x}$ inferred from $\mathcal{A}$ at position $P_{l,x}$. Therefore, we know that each precondition f of r is either invariantly true at $P_{l,x}$ or encoded as a direct fact constraint as in Eq. 5, possibly via a pseudo-fact (Eq. 13).

In the first case, we know due to the correctness of our reachability analysis (Theorem 1) that f must hold in all reachable states so far. In particular, f holds in s_I and no action visited so far may have changed it. For this reason, f also holds in s.

In the second case, due to Eq. 7, these constraints are propagated down to the final layer $L_{l'}$ from where π was extracted. At this point, assume that we already visited $k \geq 0$ actions. Consequently $s = s_k$ (where s_k is defined in Lemma 5 as the intermediate state after applying k actions) and we are currently traversing a subtree to the right of action a_k. It follows that the precondition constraints of r from layer L_l propagate down to a position at the final layer where a_k has already been applied but a_{k+1} has not yet been applied, which is exactly where s_k holds. As a consequence, fact constraints from layer L_l also directly enforce the preconditions to hold in state s_k and consequently in s. □

Theorem 2. *Let (π, H) be the decoded (classical and hierarchical) plan for the Lilotane encoding $\mathcal{L}_{l'}(\Pi)$ for TOHTN planning problem Π. Then π is a valid solution for Π and H is a valid hierarchical solution for Π.*

Proof. We know due to Lemma 6 that H satisfies criteria (1) and (2) from Def. 2 for a hierarchical solution. It remains to be shown that H satisfies criterium (3) for the decoded plan π. Namely, we show the following: If $\Omega := \langle o_1, o_2, \ldots, o_k \rangle$ is the sequence of operations visited by a depth-first traversal of H with the node ordering relation $(o, l, i) \prec (o', l', j) \Leftrightarrow i < j$, then π is a (classical) solution for Π in such a way that Ω is a "witness" for which operation should be applied at each recursive step of Def. 1.

Let $T := \langle t_0 \rangle$ be the initial tasks of Π where, in accordance with the transformation of T described in Chapter 2.1.3, t_0 is a virtual task with r_0 as its only matching reduction. Let Γ be the frontier of unvisited nodes during the depth-first traversal of H: Initially $\Gamma := \langle (r_0, 0, 0) \rangle$, and each step of the traversal removes node v from the front of Γ and pushes the children of v with respect to $\prec$ to the front of Γ. Clearly, the i-th operation o_i in Ω corresponds to the node at the front of Γ after $i - 1$ nodes have already been visited.

In the following, we transform the recursive Definition 1 into an iterative procedure in a straight forward manner to obtain a more natural proof. We maintain a state s initialized with s_I, a task sequence $\mathcal{T}$ initialized with T, and an action sequence $\mathcal{P}$ initialized with π. At each iteration, if we can apply case 2 or case 3 of Def. 1 to problem $\Pi := (D, s, \mathcal{T})$ and to solution $\pi := \mathcal{P}$ and obtain some altered Π' and π', then we update s, $\mathcal{T}$, and $\mathcal{P}$ accordingly. Specifically, we apply the i-th operation visited in H at the i-th iteration. If there is no such operation left and we can apply case 1 instead, the procedure terminates.

We show invariant (I): At all times of this procedure, $|\Gamma| = |\mathcal{T}|$ holds and for each $0 \leq i < |\mathcal{T}|$ the operation corresponding to the i-th node in Γ matches the i-th task in $\mathcal{T}$. Initially, (I) holds because $\Gamma = \langle (r_0, 0, 0) \rangle$ and $\mathcal{T} = \langle t_0 \rangle$. Next, assume that (I) holds at some point during the procedure. If task t at the front of $\mathcal{T}$ is compound, then according to Def. 1, case 2 we replace it with its subtasks. Due to (I), the frontmost node in Γ then corresponds to a reduction matching t. Following the definition of the depth-first traversal, this node in Γ is replaced with operations matching its subtasks in the correct order (Lemma 6). If task t is primitive, then according to Def. 1, case 3 we remove it from $\mathcal{T}$. Due to (I), the frontmost node in Γ then corresponds to an action matching t and, as such, is a leaf in H, hence it is removed from Γ. Both cases preserve (I).

Next, we observe that state s from above procedure is equivalent to state s from Lemma 7 because we begin with $s := s_I$ and alter s with the effects of each visited action.

Now we show that above procedure is well-defined and terminates. At iteration zero, we have tasks $\mathcal{T} := \langle t_0 \rangle$ and frontier $\Gamma := \langle (r_0, 0, 0) \rangle$. By definition, r_0 matches t_0 and r_0 has no preconditions; we can apply case 2 of Def. 1.

At the i-th iteration, we have state s, tasks $\mathcal{T}$, and action sequence $\mathcal{P}$. If $\mathcal{T}$ is not empty yet, then we know due to (I) that operation o corresponding to the first node in Γ matches the first task in $\mathcal{T}$. Lemma 7 implies that $\mathit{pre}(o)$ hold in s. Hence, if o is an action, we can apply case 3 – we remove a from the front of $\mathcal{P}$ and apply $\mathit{eff}(a)$ to s – and if o is a reduction, we can apply case 2.

If $\mathcal{T}$ is empty, then (I) implies that Γ is empty and the traversal of H is finished. In particular, all actions in π (each of which corresponds to a leaf node in H) have been visited, so each of them has been removed from $\mathcal{P}$. As a result, $\mathcal{P} = \langle \rangle$ and case 1 applies.

To conclude, we have shown that we can recursively apply the definition for a classical solution for Π to π using the sequence of visited operations in H. This proves that π is a classical solution for Π and that criterium (3) of Def. 2 is satisfied for H, concluding the proof that H is a valid hierarchical solution for Π. □

Appendix B. Derivation of Complexity Results

We now derive the complexity results discussed in Chapter 5.5.

B.1 Number of Variables

With our instantiation and encoding approach, each operation at some position of some layer may induce up to $X \cdot B$ new operations at the subsequent layer: At each of the X child positions, up to B new children are possible. Initial layer L_0 has size $|L_0| = 1$ and contains exactly $R_l = 1$ operation per position. Given layer L_{l-1} of size $|L_{l-1}|$ and with R_{l-1} operations at each position, the next layer L_l has a maximum size of $|L_l| \leq X \cdot |L_{l-1}|$ and up to $R_l \leq R_{l-1} \cdot B$ operations at each position. From these recurrent inequalities we can deduce $|L_l| \leq X^l$ and $R_l \leq B^l$. For encoding $\mathcal{L}_{l'}$ we have $R := \sum_{l=0}^{l'} |L_l| \cdot R_l \leq \sum_{l=0}^{l'} X^l \cdot B^l = 1 + XB + X^2B^2 + \ldots + X^{l'}B^{l'} \in \mathcal{O}(X^{l'}B^{l'})$ operations in total.

Each operation with U free arguments induces up to U pseudo-constants and thus $U \cdot C$ new variables to represent their possible substitutions. We arrive at $RUC \in \mathcal{O}(X^{l'}B^{l'}UC)$ variables for pseudo-constant substitutions in total.

Let F be the number of relevant facts during the planning process. In the worst case where we need to encode each fact at each position, we obtain a total of $\sum_{l=0}^{l'} |L_l| \cdot F \in \mathcal{O}(X^{l'}F)$ variables representing ground facts.

At each position, each of the up to P preconditions (and E effects) of each operation (action) may introduce a variable representing a pseudo-fact. We arrive at $R \cdot (P + E) \in \mathcal{O}(X^{l'} B^{l'} (P+E))$ variables for pseudo-facts. For some action effects and for each dominated operation, a variable denoting the equivalence of a pair of pseudo-constants may be encoded. For an action of arity V the equality of $\mathcal{O}(V^2)$ pairs of pseudo-constants may be encoded which leads to $\mathcal{O}(R \cdot V^2)$ variables. When each encoded operation dominates some other (unencoded) operation, an additional $\mathcal{O}(R \cdot U)$ equality variables could be added.

For each position we add one variable representing the primitiveness of the position, leading to $\mathcal{O}(X^{l'})$ additional variables. Finally, as explained in the next section, we introduce $\log n$ helper variables whereever we encode at-most-constraints over n variables. This results in $\mathcal{O}(X^{l'} \log B^{l'})$ variables from Eq. 4 and in $\mathcal{O}(RU \log C)$ variables from Eq. 11.

In total, the asymptotic number of encoded variables evaluates to

$$\begin{aligned} & R + RUC + X^{l'} F + R(P+E) + RV^2 + X^{l'} + X^{l'} \log B^{l'} + RU \log C \\ = \ & X^{l'} (B^{l'} (1 + UC + P + E + V^2 + U \log C) + F + 1 + \log B^{l'}) \\ \in \ & \mathcal{O}(X^{l'} (F + B^{l'} (UC + P + E + V^2))). \end{aligned} \tag{30}$$

B.2 Number of Clauses

We traverse our encoding in the same order as presented in Chapter 5.1 and provide asymptotic complexity measures for the number of added clauses according to each rule.

Eq. 1 is a single unit clause. The introduction of new facts in Eq. 2 introduces each fact once per layer, leading to $\mathcal{O}(l'F)$ unit clauses, while its optimized replacement Eq. 21 subsumes this measure. Eq. 3, the primitiveness of a position w.r.t. its active operation, is added once for each operation, leading to $\mathcal{O}(R)$ clauses.

For at-most-one constraints over operations, Eq. 4 presents the naive method of adding $\mathcal{O}(n^2)$ binary clauses to restrict n variables. However, if n becomes sufficiently large we employ a more sophisticated encoding where the number of possible variables is represented as an explicit binary number, each digit being represented by a new Boolean variable (see Schreiber, 2018, Appendix A). This enables at-most-one constraints with $\mathcal{O}(\log n)$ helper variables and $\mathcal{O}(n \log n)$ clauses. As a consequence, Eq. 4 asymptotically leads to $\mathcal{O}(X^{l'} (B^{l'} \log B^{l'})) = \mathcal{O}(R \log B^{l'})$ clauses, as $B^{l'}$ operations may occur at each position.

The preconditions and effects of operations in Eq. 5 and Eq. 6 make up for $\mathcal{O}(R(P+E))$ clauses. The fact propagations introduced by Eq. 7 are purely virtual as explained earlier.

To define child and parent relationships, we add $\mathcal{O}(R)$ clauses from Eq. 8 and Eq. 9.

Eq. 10 leads to $X^{l'}$ assumptions (which are not permanently added to the encoding). The enforcement of at least one active substitution for each pseudo-constant as defined in Eq. 12 leads to $\mathcal{O}(RU)$ clauses; one clause for each introduced pseudo-constant. Again employing a binary at-most-one constraint scheme for substitutions instead of Eq. 11 we obtain $\mathcal{O}(RU(C \log C))$ further clauses. Eq. 13 links each pseudo-fact to its respective ground fact for each possible substitution combination. Overall $\mathcal{O}(R(P+E))$ pseudo-facts are encoded, one for each precondition and effect. In the worst case, up to F ground facts may correspond to a single pseudo-fact and overall we need to add $\mathcal{O}(R(P+E)F)$ clauses.

Direct frame axioms, Eq. 14, induce up to two clauses for each fact at each position, leading to $\mathcal{O}(X^{l'} F)$ clauses. Indirect frame axioms in Eq. 15 need to be added for each fact f for each action that may indirectly support the change of f, so we may need to instantiate the formula $\mathcal{O}(FR)$ times. The number of CNF clauses constructed for each axiom is asymptotically equivalent to the number of nodes in the corresponding literal tree (see Chapter 5.1.3). For every action effect there is at most one combination of substitutions which unifies it with f. We also know that each substitution combination creates a new path in the literal tree of depth Y (the maximum arity of a predicate) counted from the end of the header literals. Hence, the tree can have up to E paths of depth Y which in the worst case lead to EY nodes and a total of $\mathcal{O}(FREY)$ indirect frame axioms.

Clauses from Eq. 22 and Eq. 23 may be added for each precondition of each operation. There can be $\mathcal{O}(F)$ possible substitution combinations for a given pseudo-fact. We can either encode the invalid options (Eq. 23) leading to one clause for each invalid substitution or encode the valid options (Eq. 22) which may induce $\mathcal{O}(Y)$ clauses for each valid substitution if realized with a literal tree. In the worst case we have around $F/2$ valid substitutions and $F/2$ invalid substitutions, in which case we add $\mathcal{O}(RPF)$ clauses in total.

Type restrictions for pseudo-constants – Eq. 16 or Eq. 17 – make up a constant number of clauses for each pseudo-constant if we choose correctly, leading to $\mathcal{O}(RU)$ clauses.

To handle contradictory effects, in Eq. 18 we enumerate the possible substitution combinations which unify two given effects of an action. In the worst case, each operator has $E/2$ negative effects which can each be unified with each of the remaining $E/2$ positive effects. This leads to $\mathcal{O}(E)$ literal trees over combinations of substitution and equality variables, similar to the ones introduced for indirect frame axioms (where pseudo-facts are unified with a *ground* fact). Each tree can induce $\mathcal{O}(EY)$ clauses, so each operation induces $\mathcal{O}(E^2 \cdot Y)$ clauses, leading to $\mathcal{O}(RE^2Y)$ clauses overall.

The equality of pairs of pseudo-constants is encoded as in Eq. 19 in some cases. For up to RV^2 variables, $\mathcal{O}(C)$ clauses are encoded, leading to $\mathcal{O}(RV^2C)$ clauses.

When each encoded operation dominates some other (unencoded) operation, Eq. 20 leads to $\mathcal{O}(R)$ clauses and up to RU equality variables inducing $\mathcal{O}(RUC)$ further clauses.

In total we arrive at an asymptotic number of

$$\begin{aligned}
&\mathcal{O}\Big(l'F + R + R\log B^{l'} + R(P+E) + R + RU + RU(C\log C) + R(P+E)F + X^{l'}F \\
&\qquad + FREY + RPF + RU + RE^2Y + RV^2C + RUC\Big) \\
&= \mathcal{O}\Big(R\big(l'\log B + UC\log C + (P+E)F + FEY + PF + E^2Y + V^2C\big) + FX^{l'}\Big) \\
&= \mathcal{O}\Big(R\big(l'\log B + C(U\log C + V^2) + F(P+YE) + YE^2\big)\Big) \qquad (31)
\end{aligned}$$

permanent clauses added to the encoding.

Appendix C. Supplementary Figures

We conclude with a number of supplementary illustrations and tables which provide some more insight into our evaluations.

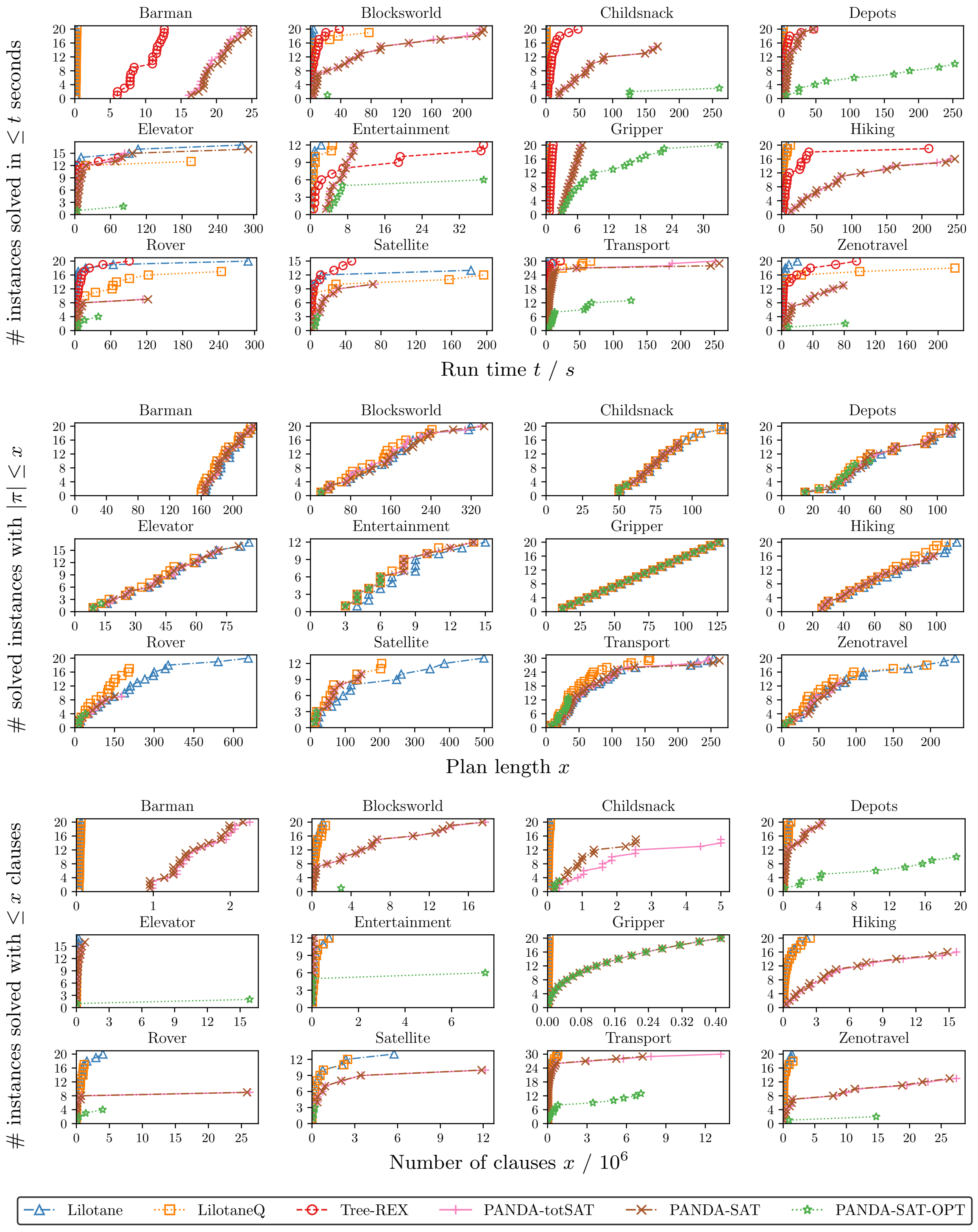


Figure 16: Per-domain properties of SAT-based planners (plan lengths without Tree-REX).

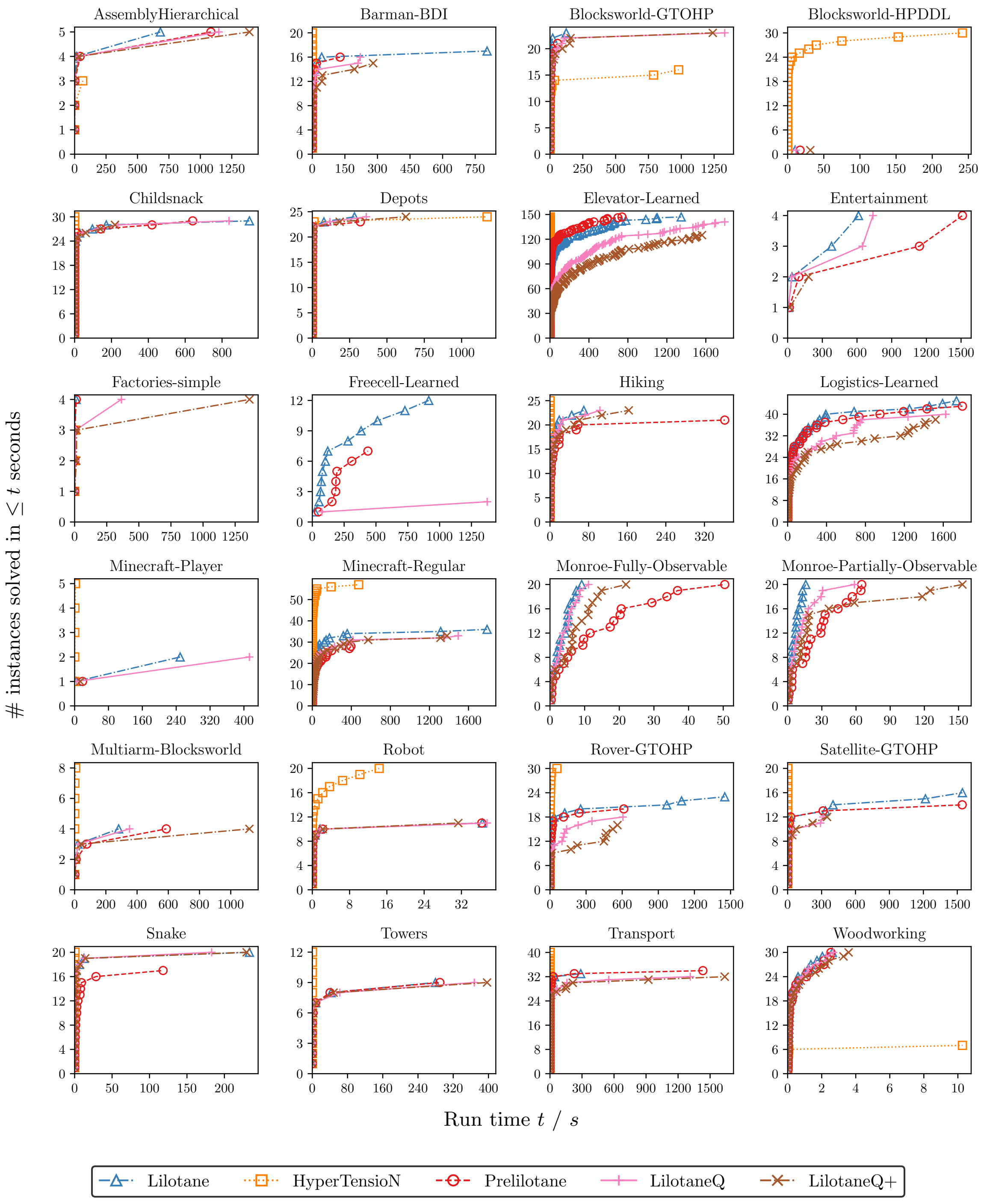


Figure 17: Per-domain run times in the IPC followup evaluations.

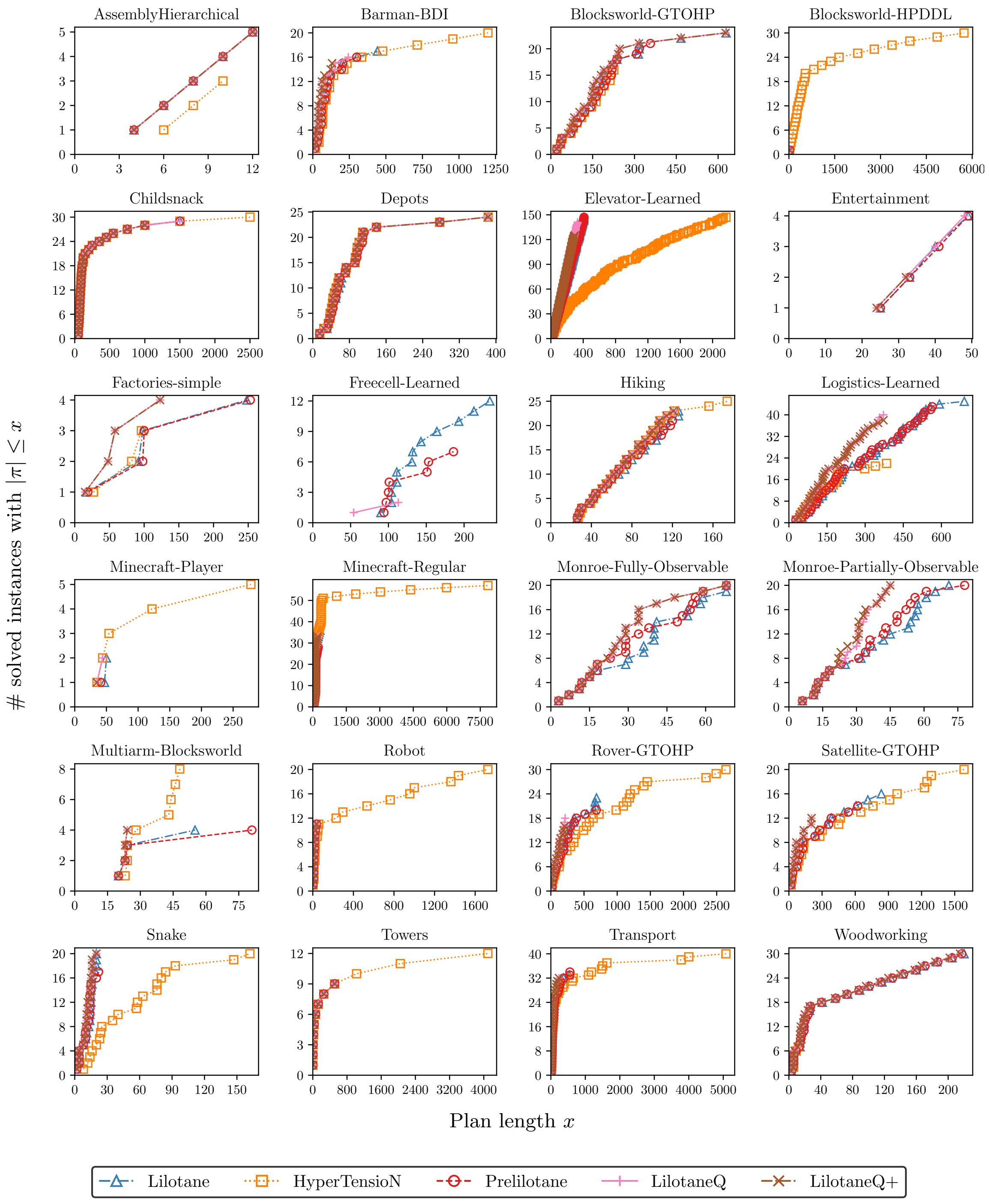


Figure 18: Per-domain plan lengths in the IPC followup evaluations.

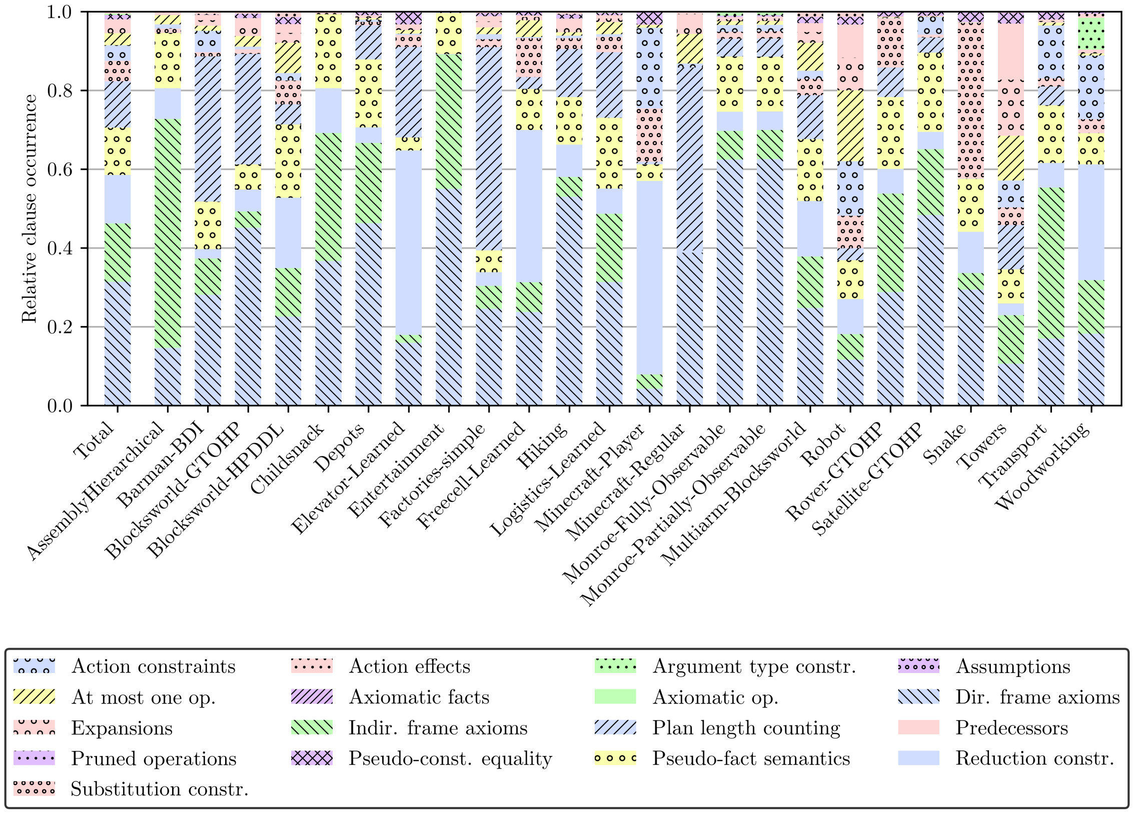


Figure 19: Distribution of occurrences of different clause categories encoded by LilotaneQ on the IPC benchmarks, overall (leftmost column) and per domain. All instances for which LilotaneQ found some initial plan were considered.

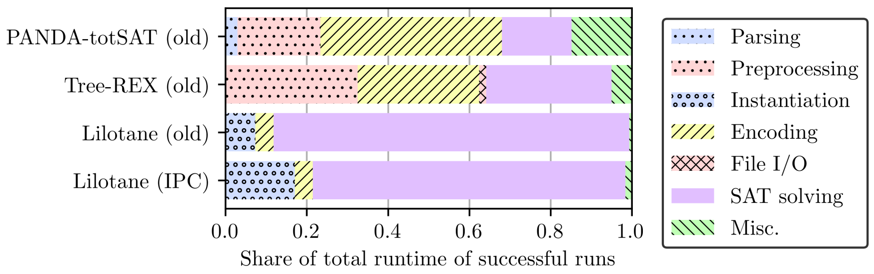


Figure 20: Partition of run times by stage for three SAT-based planners on an old set of benchmarks (see Fig. 14) and, additionally, for Lilotane on the IPC benchmarks.

Domain	slv.	pos.	lay.	cls./10^6	$\frac{\text{act.}}{\text{pos.}}$	$\frac{\text{red.}}{\text{pos.}}$	$\frac{\text{psc.}}{\text{pos.}}$	ret.pr.	pruned	dom.	+prec.
AssemblyH.	5	136.00	14.00	0.43	9.75	1.54	1.94	50.60	50.60	71.80	76.00
Barman	17	702.53	6.00	0.14	2.15	0.54	0.59	0.00	0.00	0.00	2.00
Blocksworld-G.	23	8215.70	10.39	0.66	2.22	0.07	0.05	7.48	80.61	26.00	6.00
Blocksworld-H.	1	10929.00	13.00	3.34	3.99	3.25	1.25	0.00	0.00	3.00	11.00
Childsnack	29	284.38	2.00	1.67	0.83	0.17	0.66	0.00	0.00	0.00	8.00
Depots	24	1696.58	6.00	1.48	2.59	0.24	0.42	0.96	7.12	25.67	11.00
Elevator	147	3547.09	8.99	1.05	2.23	0.61	0.25	0.01	0.01	0.00	7.00
Entertainment	4	510.75	7.25	20.31	3.26	2.00	0.95	0.00	0.00	282.00	64.00
Factories	4	4747.75	12.75	0.51	2.01	0.94	0.79	0.00	0.00	21.25	0.00
Freecell	12	3147.08	9.33	13.85	8.85	11.00	3.12	3.75	3.75	3653.75	642.00
Hiking	23	10578.04	12.48	0.78	1.53	0.15	0.32	18.57	3873.09	0.00	7.00
Logistics	45	3483.71	11.91	1.99	4.65	1.29	0.73	0.00	0.00	0.00	41.07
Minecraft-P.	2	777.50	8.50	5.05	4.98	3.07	1.50	19.50	47.50	0.50	5.00
Minecraft-R.	35	10699.46	12.60	1.22	4.25	0.69	0.00	94.46	630.17	0.00	2.00
Monroe-FO.	20	1153.95	7.85	0.84	3.14	0.67	0.60	6.55	28.15	86.45	112.75
Monroe-PO.	20	1811.55	8.00	1.32	3.22	0.64	0.58	4.35	15.95	134.45	89.80
Multiarm-Bl.	4	22465.75	14.50	5.26	4.16	1.90	0.88	0.00	0.00	7.25	11.00
Robot	11	186.73	15.00	0.01	4.25	0.55	0.45	2.91	3.09	0.00	13.00
Rover	23	826.22	4.57	1.44	2.07	0.23	0.30	0.04	0.04	0.00	43.43
Satellite	16	1562.56	7.00	3.12	1.95	0.43	0.31	0.00	0.00	0.00	7.00
Snake	20	244.20	8.55	1.00	2.44	0.56	0.59	0.00	0.00	0.00	7.00
Towers	9	19601.33	119.56	0.30	2.47	0.01	0.02	0.00	0.00	0.00	4.00
Transport	33	393.64	4.64	0.27	2.46	0.29	0.60	0.24	0.24	14.94	13.00
Woodworking	30	249.47	4.63	0.09	1.53	0.56	1.60	4.97	5.10	0.57	126.77

Table 6: Statistics overview of all IPC instances solved by Lilotane. From left to right: Solved instances; created positions / layers / clauses; actions / reductions / pseudo-constants per position; retroactive prunings, operations pruned by these prunings, dominated operations, inferred preconditions. All but the three "per position" measures are arithmetic averages over all solved instances in the domain. Measures for operations and pseudo-constants per position also include the objects which are later removed again due to retroactive prunings or dominated operations.

References

Alford, R., Bercher, P., & Aha, D. (2015). Tight bounds for HTN planning. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling*, Vol. 1, pp. 7–15.

Ankerl, M. (2020). robin_hood unordered map & set. `https://github.com/martinus/robin-hood-hashing`. Accessed: 2020-09-25.

Audemard, G., & Simon, L. (2009). Predicting learnt clauses quality in modern SAT solvers. In *Twenty-first International Joint Conference on Artificial Intelligence*, pp. 399–404.

Balyo, T., Biere, A., Iser, M., & Sinz, C. (2016). SAT race 2015. *Artificial Intelligence*, *241*, 45–65.

Behnke, G., Bercher, P., & Höller, D. (2020a). Plan verification. `http://gki.informatik.uni-freiburg.de/ipc2020/format.pdf`. Accessed: 2020-09-24.

Behnke, G., Bercher, P., & Höller, D. (2020b). *Proceedings of the 2020 International Planning Competition (IPC)*. To be published.

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

Behnke, G., Höller, D., & Biundo, S. (2019a). Bringing order to chaos – a compact representation of partial order in SAT-based HTN planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33, pp. 7520–7529.

Behnke, G., Höller, D., & Biundo, S. (2019b). Finding optimal solutions in HTN planning – a SAT-based approach. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pp. 5500–5508.

Behnke, G., Höller, D., Schmid, A., Bercher, P., & Biundo, S. (2020). On succinct groundings of HTN planning problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34, pp. 9775–9784.

Bercher, P., Alford, R., & Höller, D. (2019). A survey on hierarchical planning – one abstract idea, many concrete realizations. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pp. 6267–6275.

Blaylock, N., & Allen, J. (2005). Generating artificial corpora for plan recognition. In *International Conference on User Modeling*, pp. 179–188. Springer.

Bonet, B., & Geffner, H. (2020). Learning first-order symbolic representations for planning from the structure of the state space. In *European Conference on Artificial Intelligence*. IOS Press.

Bylander, T. (1994). The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, *69*(1-2), 165–204.

Cashmore, M., Fox, M., & Giunchiglia, E. (2013). Partially grounded planning as quantified boolean formula. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling*, pp. 29–36.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pp. 151–158.

Corrêa, A. B., Pommerening, F., Helmert, M., & Frances, G. (2020). Lifted successor generation using query optimization techniques. In *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling*, pp. 80–89.

Ernst, M. D., Millstein, T. D., & Weld, D. S. (1997). Automatic sat-compilation of planning problems. In *IJCAI*, Vol. 97, pp. 1169–1176.

Erol, K., Hendler, J., & Nau, D. S. (1994). HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 1123–1128.

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

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

Georgievski, I., & Aiello, M. (2015). HTN planning: Overview, comparison, and beyond. *Artificial Intelligence*, *222*, 124–156.

Ghallab, M., Nau, D., & Traverso, P. (2004). *Automated Planning: theory and practice*. Elsevier.

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

Helmert, M. (2009). Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, *173*(5-6), 503–535.

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

Höller, D., Behnke, G., Bercher, P., Biundo, S., Fiorino, H., Pellier, D., & Alford, R. (2020). HDDL: An extension to PDDL for expressing hierarchical planning problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34, pp. 9883–9891.

Kautz, H., & Selman, B. (1998). BLACKBOX: A new approach to the application of theorem proving to problem solving. In *AIPS98 workshop on planning as combinatorial search*, pp. 58–60.

Kautz, H., Selman, B., & Hoffmann, J. (2006). SatPlan: Planning as satisfiability. In *5th international planning competition*, p. 156.

Kautz, H. A., Selman, B., et al. (1992). Planning as satisfiability. In *European Conference on Artificial Intelligence*, Vol. 92, pp. 359–363. Citeseer.

Magnaguagno, M., Meneguzzi, F., & de Silva, L. (2020). HyperTensioN – a three-stage compiler for planning. In *Proceedings of the 2020 International Planning Competition (IPC)*. To appear.

Mali, A. D., & Kambhampati, S. (1998). Encoding HTN planning in propositional logic. In *Artificial Intelligence Planning Systems*, pp. 190–198.

Nadel, A., & Ryvchin, V. (2012). Efficient SAT solving under assumptions. In *International Conference on Theory and Applications of Satisfiability Testing*, pp. 242–255. Springer.

Nau, D., Cao, Y., Lotem, A., & Munoz-Avila, H. (1999). SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, Vol. 2, pp. 968–973.

Ramoul, A., Pellier, D., Fiorino, H., & Pesty, S. (2017). Grounding of HTN planning domain. *International Journal on Artificial Intelligence Tools*, *26*(05), 1760021.

Reiter, R. (1981). On closed world data bases. In *Readings in artificial intelligence*, pp. 119–140. Elsevier.

Rintanen, J. (2004). Evaluation strategies for planning as satisfiability. In *European Conference on Artificial Intelligence*, Vol. 16, p. 682.

Rintanen, J. (2014). Madagascar: Scalable planning with SAT. In *Proceedings of the 8th International Planning Competition (IPC-2014)*.

Robinson, N., Gretton, C., Pham, D. N., & Sattar, A. (2009). SAT-based parallel planning using a split representation of actions. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, pp. 281–288.

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

Schreiber, D., Pellier, D., Fiorino, H., & Balyo, T. (2019a). Efficient SAT encodings for hierarchical planning. In *Proceedings of the 11th International Conference on Agents and Artificial Intelligence*, Vol. 2, pp. 531–538.

Schreiber, D., Pellier, D., Fiorino, H., et al. (2019b). Tree-REX: SAT-based tree exploration for efficient and high-quality HTN planning. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling*, pp. 382–390.

Sönnichsen, M., & Schreiber, D. (2020). The "Factories" HTN domain. In *Proceedings of the 2020 International Planning Competition (IPC)*. To appear.

Soos, M., Nohl, K., & Castelluccia, C. (2009). Extending SAT solvers to cryptographic problems. In *Theory and Applications of Satisfiability Testing - SAT 2009*, pp. 244–257. Springer.

Streeter, M. J., & Smith, S. F. (2007). Using decision procedures efficiently for optimization. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling*, pp. 312–319.

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

Wichlacz, J., Torralba, A., & Hoffmann, J. (2019). Construction-planning models in minecraft. In *Proceedings of the 2nd ICAPS Workshop on Hierarchical Planning*.

Williams, M. (2020). Partially instantiated representations for automated planning. Master's thesis, Karlsruhe Institute of Technology.