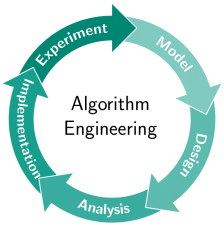

Pruning Techniques for Lifted SAT-Based Hierarchical Planning

Master's Thesis of

Nikolai Schnell

at the Department of Informatics
Institute for Theoretical Computer Science, Algorithm Engineering

Reviewer: Prof. Dr. rer. nat. Peter Sanders
Advisor: M.Sc. Dominik Schreiber

01. June 2021 – 01. December 2021

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, December 1, 2021

. . . . . .

(Nikolai Schnell)

Abstract

The very universal and generic problem of *automated planning* deals with finding sequences of actions to be executed by autonomous agents to achieve certain goals. Automated planning is applied in many fields that benefit from autonomous decision making and plan finding. In recent years, *hierarchical planning* formalized by *HTN (Hierarchical Task Network) planning* has become a more popular approach for automated planning, which can be used to create planning problems with a hierarchical structure using expert knowledge. A recently popularized approach to solving HTN planning problems is *SAT-based HTN planning*, where an increasing portion of an HTN planning problem is encoded into a SAT formula, the SAT formula is solved by a SAT solver, and the resulting solution to the SAT formula is decoded back into a solution for the planning problem. Most SAT-based HTN planners use a *grounding* step as part of their algorithm which can introduce an exponential overhead. However, a recently introduced *lifted* SAT-based HTN planner called Lilotane skips this grounding step and operates on lifted (parametrized) tasks and methods instead of a flat representation. While this approach is often beneficial regarding runtimes, the techniques for reducing the size of the encoding as used by ground SAT-based HTN planners are not trivially transferable to Lilotane. Hence, in this thesis we introduce pruning techniques for lifted SAT-based HTN planning in Lilotane that replace the existing techniques with more sophisticated algorithms and invest additional work before the encoding step to reduce the size of the resulting SAT formula. Our main approach is a more sophisticated reachability analysis consisting of a look-ahead traversal of the HTN hierarchy during runtime as well as procedures to infer postconditions. An evaluation of our approach using the benchmarks from the IPC 2020 shows that it improves the overall performance of Lilotane. On some domains of the benchmark set, our approach achieves a pruning in the number of clauses in the SAT formula of up to two orders of magnitude and runtime improvements of up to one order of magnitude. When comparing our approach to other state-of-the-art TOHTN planners, it improves on Lilotane's advantage in terms of performance to previously beaten planners and makes Lilotane the best performing planner on seven to previously only five out of 24 domains.

Zusammenfassung

Das sehr universelle und allgemeine Problem der automatisierten Planung befasst sich mit dem Finden von Handlungssequenzen, die von autonomen Agenten ausgeführt werden sollen, um bestimmte Ziele zu erreichen. Automatisierte Planung wird in vielen Bereichen angewandt, die von autonomer Entscheidungsfindung und Planfindung profitieren. In den letzten Jahren hat sich *hierarchische Planung*, formalisiert durch *HTN (Hierarchical Task Network)-Planung*, zu einem beliebteren Ansatz für die automatisierte Planung entwickelt, mit dem Planungsprobleme mit einer hierarchischen Struktur unter Verwendung von Expertenwissen erstellt werden können. Ein neuerdings populär gewordener Ansatz zur Lösung von HTN-Planungsproblemen ist die *SAT-basierte HTN-Planung*, bei der ein zunehmender Teil eines HTN-Planungsproblems in eine SAT-Formel kodiert, die SAT-Formel von einem SAT-Solver gelöst, und die sich daraus ergebende Lösung zurück in eine Lösung für das Planungsproblem dekodiert wird. Die meisten SAT-basierten HTN-Planer verwenden einen *grounding*-Schritt als Teil ihres Algorithmus, der einen exponentiellen Overhead verursachen kann. Ein kürzlich vorgestellter parametrisierter (lifted) SAT-basierter HTN-Planer mit dem Namen Lilotane überspringt diesen grounding-Schritt und arbeitet mit parametrisierten Aktionen und Methoden statt einer instanziierten Darstellung. Während dieser Ansatz oft vorteilhaft für die Laufzeit ist, sind die Techniken zur Reduzierung der Größe der Kodierung, wie sie von instanziierten (grounded) SAT-basierten HTN-Planern verwendet werden, nicht trivial auf Lilotane übertragbar. Daher führen wir in dieser Arbeit Pruning-Techniken für parametrisierte SAT-basierte HTN-Planung in Lilotane ein, die die bestehenden Techniken durch anspruchsvollere Algorithmen ersetzen und zusätzliche Arbeit vor dem Kodierungsschritt verrichten, um die Größe der resultierenden SAT-Formel zu reduzieren. Unser Hauptansatz ist eine verfeinerte Erreichbarkeitsanalyse, die aus einer vorausschauenden Traversierung der HTN-Hierarchie während der Laufzeit sowie aus Prozeduren zur Ableitung von Postkonditionen besteht. Eine Evaluation unseres Ansatzes anhand der Benchmarks der IPC 2020 zeigt, dass er die Gesamtleistung von Lilotane verbessert. In einigen Domänen der Benchmarks erreicht unser Ansatz eine Reduzierung der Anzahl der Klauseln in der SAT-Formel um bis zu zwei Größenordnungen und Laufzeitverbesserungen von bis zu einer Größenordnung. Beim Vergleich unseres Ansatzes mit anderen TOHTN-Planern, die auf dem neuesten Stand der Technik sind, verbessert er den Vorteil von Lilotane in Bezug auf die Leistung gegenüber anderen Planungsalgorithmen und macht Lilotane zum besten Planungsalgorithmus auf sieben gegenüber zuvor nur fünf von 24 Domänen.

Contents

List of Figures

List of Tables

1. Introduction

Automated planning is a subfield of Artificial Intelligence which deals with the finding of sequences of actions to be executed by autonomous agents to achieve certain goals [11]. This very universal and generic problem has previously found application in spacecraft control [10], cyber security [8], greenhouse logistics [12] and many more fields that benefit from autonomous decision making and plan finding.

So called *classical planning* represents the original and most popular formalization of the automated planning problem. To give a general overview, an instance of a classical planning problem defines the features (state) of a world, actions to be executed in this world which alter its state, an initial state and a goal state. A solution to such a classical planning problem is a sequence of actions, so that after the execution of this sequence, starting from the initial state, the goal state is reached. In addition, each action has preconditions which must hold at its execution.

In recent years however, the so-called concept of hierarchical planning has become more popular, with its most noteworthy formalization being *HTN* (Hierarchical Task Network) planning. It extends classical planning by introducing the option to model planning domains as a hierarchical structure and thus allows human experts to model planning problems more effectively and intuitively. For this extension, so called *tasks* are introduced. While some tasks can be achieved by actions similar to the actions in classical planning that directly change the world state, other tasks can only be achieved by so-called *methods* which define multiple further tasks that need to be achieved to solve the original task. This latter mechanism gives HTN planning its hierarchical structure. Additionally, all operations (methods and actions) are parametrized with variables that need to be substituted with concrete values for a solution to the problem.

An approach for solving hierarchical planning problems is *SAT-based HTN planning*. This approach relies on the boolean satisfiability problem (SAT), in which a variable assignment for a formula in propositional logic has to be found. Due to decades of intense research, there are various highly efficient algorithms for solving SAT – SAT solvers – freely available, and SAT-based HTN planning makes use of these algorithms. For a given HTN problem, SAT-based HTN planning algorithms create partial, iteratively growing expansions of the given HTN hierarchy and repeatedly encode these partial expansions into a SAT formula until the SAT solver can find a solution, which is then decoded back into a solution to the HTN problem.

Most SAT-based HTN planning algorithms contain a *grounding* procedure. Roughly speaking, this grounding procedure takes the operations from the HTN problem and computes all combinations of substitutions for the variables of each operation, because working with these fully substituted operations is easier than with the original non-ground problem. Depending on the planning problem, the grounding procedure can introduce an exponential blowup in terms of runtime and space, since the number of possible

combinations might be exponentially big. To mitigate this, Schreiber recently introduced a SAT-based HTN planner called *Lilotane* in [21]. This planner skips the grounding step and works on the non-ground, also called *lifted*, version of the problem and leaves the substitution of variables to be done to the SAT solver. Lilotane competed in the 2020 International Planning Competition (IPC) [4] and scored the second place, performing best on various domains.

To decide which operations to encode into the SAT formula and how to restrict their variables, Lilotane uses a generalized state and a reachability analysis. This generalized state is computed by extending the state by the possible effects of each operation, which are currently computed in a highly over-approximative way. Depending on the planning problem, this can cause operations to be encoded which are actually not possible and causes unnecessary overhead in Lilotane. Our motivation is to introduce a more sophisticated way of computing the possible effects of an operation that is more accurate and can mitigate this overhead by pruning these operations before they are encoded.

1.1. Contributions

We developed pruning techniques for lifted SAT-based hierarchical planning with Lilotane that work by improving the reachability analysis in Lilotane to be more accurate, and invest additional work before the encoding step to reduce the amount of computation spent on encoding and SAT solving by pruning impossible operations before they are encoded for an overall reduction of runtime. Our main technique for finding more accurate possible effects for an operation is a dynamic, look-ahead traversal of the hierarchy originating from that operation, during which we check preconditions to identify impossible operations and restrict the variables of operations occuring in that hierarchy with our concept called *variable restrictions*. We further successfully introduced the concept of *postconditions* into Lilotane by presenting procedures to infer such postconditions, which we then integrated into our look-ahead traversal to achieve an even finer reachability analysis.

We evaluated our approach using the benchmarks of the IPC 2020 [4]. The evaluation of our approach revealed an overall improvement of Lilotane's Performance. On some domains, we observed a reduction of the number of clauses in the final encoding of up to two orders of magnitude and an improvement in runtime of up to one order of magnitude. When comparing our approach to other state-of-the-art HTN planners, it improves Lilotane's advantage in terms of performance to previously beaten planners and makes Lilotane the best performing planner on seven to previously only five out of 24 domains.

Lastly, we have identified a bug[1] in Lilotane's previous source code and an inconsistency[2] in the previous Lilotane publication [21] and have proposed corrections which have been integrated since then.

[1] https://github.com/domschrei/lilotane/pull/7
[2] https://github.com/domschrei/lilotane/pull/6

1.2. Structure of Thesis

This thesis is structured as follows. In chapter 2 we will introduce the necessary concepts, definitions and notations for the thesis, touch on related work that has been done in SAT-based HTN planning and elaborate on the general approach of Lilotane. After this we present our pruning techniques in chapter 3. We then evaluate our approach extensively in chapter 4 and finally summarize our findings and touch on possible future work in chapter 5.

2. Preliminaries

In this chapter we first define the problem of totally ordered HTN planning and introduce the approach of SAT-based HTN planning. We then touch on related work and finally give an overview of lifted SAT-based HTN Planning with Lilotane.

2.1. Totally Ordered Hierarchical Planning

Our formalization of TOHTN (Totally Ordered Hierarchical Task Network) is based on the formalization from [21]. To make understanding more intuitive we will use the planning domain 'Factories' as an example. In this domain, factories are constructed at certain locations by using resources produced from already existing factories and by transporting these resources between locations via trucks. This domain has three types of objects: factories, resources and locations, where locations represent distinct locations and resources and factories represent types of resources and factories. We call specific objects of these types *constants*.

The next concept we need to introduce is that of a *signature*. Signatures are of the form $\sigma(a_1, \ldots, a_k)$ and consist of a name σ and a fixed number k of parameters a_i. The number k is called the signature's *arity*. The parameters can either be constants or *variables* (placeholders for constants), where variables have a type τ_i that represents a subset of a domain $\tau_i \subset D$. Signatures with all parameters being constants are called *ground* signatures, and non-ground signatures are called *lifted* signatures. We refer to the variables left in a signature as *free arguments*.

A *predicate* is a signature that represents a logical atom. In the Factories domain, an example for a predicate is *produces(f,r)* which represents the logical atom of a factory of type f being able to produce a resource of type r. To represent whether this logical atom holds or not, the predicate is equipped with a polarity (positive or negative) and forms a *literal*. A *fact* is a ground literal and a *state* is a set of positive facts. A world state is always well-defined: every ground predicate whose positive fact is not contained in the world state is implicitly assumed to be false. For a literal l we represent the equivalent literal with switched polarity as $\neg l$ and for a set set of literals we represent its negative literals as set^- and its positive literals as set^+.

The central part of HTN planning are *tasks*. These are non-predicate signatures that represent something to be achieved. In the factories domain, $construct\_factory(f, l)$ is such a task which represents the construction of a factory f at a location l. Tasks can be achieved by operations and there are two types of operations: *methods* or *operators*. If an operation can achieve a task we also say it *matches* the task. Speaking generally, a method achieves a task by decomposing it into a sequence of further tasks called *subtasks*, which can be matched by further operations that we will often call the *children* of the method.

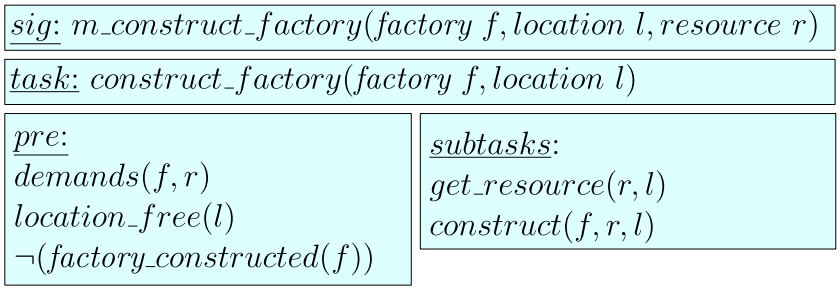

Figure 2.1.: The *m_construct_factory* method

The defining factor of *totally ordered* HTN planning is the total order of this sequence of subtasks. When not requiring an order for the subtasks of methods as it is the case in general HTN planning, the problem becomes much more difficult as we will touch on in Section 2.1.1.

More specifically, a method is a tuple $(task, sig, pre, subtasks)$. Here, $task$ is the task to be achieved and sig is a signature that provides a unique name to the method and contains all parameters used in pre, $task$ and $subtasks$. pre is a set of literals that represent preconditions that need to hold in the world state s ($pre \subset s$) in order to apply the method and $subtasks$ is the sequence of further tasks that need to be achieved. A method where sig, and thus every other signature in the method, is ground, is called a *reduction*. In the Factories domain, a method that matches the *construct_factory* task is shown in Fig. 2.1. In addition to the parameters f and l of the task, the signature also contains the parameter r. This is the resource that is required to be able to construct the factory of type f, which is enforced via the precondition $demands(f, r)$. The other preconditions ensure that there is not already a factory at location l and that a factory of type f has not been constructed yet.

Conversely, an operator achieves a task without introducing further tasks. Operators are tuples *(task, sig, pre, eff)* similarly to methods, but instead of a sequence of subtasks, the operator contains a set of literals *eff* that change the world state when this operator is executed: First, the negative effects are removed from the world state and then the positive effects are added. To give an example, consider the task $construct(f, r, l)$, which represents the task of constructing factory f at location l, by using a resource r which is already at the location l. This task is one of the subtasks of the $m\_construct\_factory$ method. The operator for achieving this task can be seen in Fig. 2.2.

We can now formally define the actual TOHTN *problem*. For this, we first define a TOHTN domain as follows.

Definition 1 *A TOHTN domain $D = (P, C, M, O)$ consists of Predicates P, Constants C, Methods M and Operators O as defined above. The possible Reductions R and Actions A arise from grounding the Methods and Operators with all possible combinations of constants for their parameters.*

Using this, we can define a problem instance.

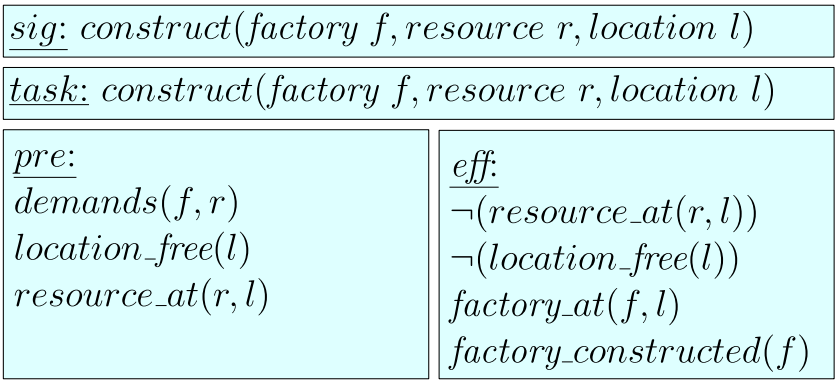

Figure 2.2.: The *construct* operator

Definition 2 *An instance of the TOHTN Problem* $\Pi := (D, s_I, r_0, T)$ *consists of a TOHTN Domain* D*, an initial state* s_i *and an initial reduction* r_0 *without preconditions and subtasks* T*, whereas* T *is an ordered list of ground tasks to be executed.*

We define a solution to a TOHTN problem using the following notions:

Definition 3 *A solution candidate for a TOHTN problem* $P = (D, s_i, r_0, T)$ *is a directed tree* $T = (G, V)$ *with a total node ordering relation and the following properties:*

1. *Every non-leaf node corresponds to a reduction and its children correspond to one operation each that match the subtasks of this reduction.*
2. *The root node in* T *corresponds to* r_0*.*
3. *Every leaf corresponds to an action from* A*.*

Further we define:

Definition 4 *A solution candidate traversal for a specific solution candidate is*

1. *a depth-first traversal of the tree according to the total node ordering relation*
2. *during which a world state* s *is maintained that is initialized as the intial state* s_i*, and is updated when reaching an action by applying the action's effects.*

Definition 5 *A solution candidate traversal for a solution candidate is valid if the preconditions* pre *of every operation, when its corresponding node is reached, hold, i.e.,* $pre^+ \subset s$ *and* $pre^- \bigcap s = \emptyset$*.*

We can now finally define a solution for a TOHTN problem:

Definition 6 *A solution candidate for a given TOHTN instance is a solution for this instance if its well defined traversal is valid.*

Further we define multiple symbols that we will use later on.

Definition 7 *For a set of literals s, $\overline{s}$ represents the elementwise negation of s, meaning the set of all literals in s with switched polarity.*

Definition 8 *Given two sets s_1 and s_2 of literals, $s_1 \setminus s_2$ represents the relative complement of s_2 in s_1, meaning all literals that are part of s_1 that are not part of s_2.*

Definition 9 *Given two sets s_1 and s_2 of literals, $s_1 \setminus_{pred} s_2$ represents the relative predicate complement of s_2 in s_1, meaning all elements that are part of s_1 for which no literal of the same predicate and same polarity exists in s_2.*

We also group predicates into two groups.

Definition 10 *For a TOHTN domain $D = (P, C, M, O)$, a predicate $p \in P$ is called rigid if*

$$\forall (task, sig, pre, eff) \in O : \forall e \in eff : e.predicate \neq p$$

Definition 11 *For a TOHTN domain $D = (P, C, M, O)$, a predicate $p \in P$ is called fluent if it is not rigid.*

Further, we also call literals and preconditions rigid (fluent) if their predicates are rigid (fluent).

2.1.1. Complexity

Compared to classical planning which is PSPACE-complete, general hierarchical planning is semi-decidable and, as such, more complex. The total order of subtasks in TOHTN planning softens this a bit; TOHTN planning is only 2-EXPTIME-complete, roughly meaning that no general algorithm for solving TOHTN planning problems can be faster than $O(2^{2^{p(n)}})$ where p is a polynomial function and n is the problem size in bits. Despite this high worst-case complexity, there are still solvable TOHTN and HTN instances in practice.

2.1.2. TOHTN as Satisfiability

The SAT or propositional satisfiability problem is the problem of finding an assignment to all Boolean variables in a propositional formula in conjunctive normal form [21] so that the formula evaluates to *True*. In general, many problems are reduced to SAT by transforming them into an equivalent SAT instance, solving this formula with a SAT solver, and then decoding the found variable assignment into a solution to the original problem. In HTN planning this is not quite that simple. As SAT is famously NP-complete and TOHTN planning is 2-EXPTIME-complete, the resulting (conjectured) asymptotic gap makes a direct encoding of a complete TOHTN problem instance into a single SAT formula generally infeasible. Instead, often the encoding is expanded iteratively until a solution is found.

2.2. Related Work

In this section we will summarize the literature on general HTN planning and SAT-based HTN planning algorithms.

2.2.1. HTN Planning

What could be considered as the canonical approach to solving HTN problems is *progression search*, since this type of algorithm closely follows the definition of the solution to an HTN problem. Essentially, progression search algorithms build the solution in a forward manner while maintaining a state by progressively finding operations that match the tasks to be executed, adding applicable primitive operations to the final plan, using found compound operations to substitute tasks to be executed by new tasks and outputting the final plan when no more tasks are left. Some of the earliest successful algorithms of this type were the SHOP algorithm and its successor SHOP2 [18, 19], but newer progression search planners are still being developed like HyperTensioN [16], which won the 2020 International Planning Competition. Another recently introduced progression search planner is PandaGBFS [13], which combines progression search and using heuristics that incorporate the maintained state to guide the search. It should be noted that there exist both progression search planners which work on the lifted problem domain (SHOP, SHOP2 or HyperTensioN) and progression search planners working with a grounded version of the problem domain (PandaGBFS).

2.2.2. SAT-Based HTN Planning

Encodings for SAT-based HTN planning were first introduced in 1998 by Mali and Kambhampati in [17], after SAT-based approaches for classical planning had been found to be feasible. However, because their encoding was inefficient [21] and could only be applied to acyclic HTN domains, meaning the graph defined by the task relationships of a domain could not contain a cycle, it did not find much application and essentially no literature was published on SAT-based HTN planning until two decades later.

Then, in 2018 Behnke et al. presented a SAT-based TOHTN planner called totSAT using a novel encoding that outperformed other non-SAT-based HTN planners[5]. They extended this planner further to be able to find optimal plans and find plans for general HTN domains, and we will refer to the current version of their planner as PandaSAT, since we will also be using it in our evaluation in chapter 4.

Independently of this, in 2019 Schreiber et al. also introduced a novel and efficient encoding for SAT-based HTN planning that was used to create the TOHTN planning algorithm Tree-REX, which also compared favourably against other state-of-the-art HTN planners [23, 22].

While developed independently, both Tree-REX and PandaSAT converged in a certain part of their general procedure: Both planners traverse the HTN hierarchy along its depth, incrementally expanding an encoding after every layer and handing this encoding to a SAT solver. PandaSAT hands these encodings to a SAT solver operating non-incrementally,

meaning it knows nothing of the previous encodings and starts from scratch every time, while Tree-REX makes use of incremental SAT solving to avoid redundant work.

Both PandaSAT and Tree-REX also rely on the previously mentioned grounding step. For this procedure, Tree-REX uses the result from Ramoul et al., while the authors of PandaSAT created their own grounding procedure [20, 6].

Finally, Schreiber introduced Lilotane (Lifted Logic for Task Networks) as the successor of Tree-REX [21]. Like Tree-REX, Lilotane iteratively traverses the given HTN hierarchy layer by layer and makes use of incremental SAT solving. On the other hand, Lilotane skips the grounding step and hence presents a lifted SAT-based HTN planning approach, which is reflected in a different encoding procedure than Tree-REX, because in Lilotane the SAT Solver is responsible for substituting constants for variables. Lilotane took part in the International Planning Competition 2020 [4], where it scored the second place. In the upcoming Section 2.3 we will go into detail on how Lilotane works.

The authors of PandaSAT have also published pruning techniques for SAT-based TOHTN planning [3]. These techniques work by doing a reachability analysis at the bottom of their current expansion of the hierarchy to identify impossible actions and then propagating this information to the rest of the hierarchy to prune further operations. Lilotane actually uses similar techniques for pruning but due to the lifted nature of Lilotane the reachability analysis is less accurate and these techniques are not fully transferable to Lilotane.

2.3. Lifted SAT-Based Hierarchical Planning with Lilotane

In this section we will introduce lifted SAT-based HTN Planning as it is approached by Lilotane. We will start by giving a general overview of Lilotane.

2.3.1. Lilotane Overview

A simplified pseudocode for the general Lilotane procedure can be seen in Alg. 1.

Generally, Lilotane features an iterative procedure in which the given TOHTN problem is subdivided into a sequence of layers which are generated and encoded after one another until a solution is found. In each iteration, Lilotane instantiates all possible children of the operations of the previous layer, the details of which will be presented in the next section. After the instantiation, the entire network of reductions is encoded into a SAT formula and given to a SAT solver. If the SAT solver finds a solution, this solution is decoded into a valid plan and the algorithm is done. If the SAT solver does not find a solution, the next layer is instantiated. This procedure is repeated until a solution is found. Additionally, one should note that the SAT formula is not generated from scratch in each iteration, rather the formula from the previous layer is extended using the newly found operations.

2.3.1.1. Instantiation

In this section we will give an overview of the *instantiate* and *getPFC* functions used in Alg. 1.

Input: $\Pi = (D, s_I, T)$
Result: solution π

$r_0 := createInitialReduction(T)$;
$last\_layer := \langle\langle r_0 \rangle\rangle$;
$task\_network := last\_layer$;
while $True$ **do**
  $possible\_facts := (s_I, \langle\rangle)$;
  $new\_layer := \langle\rangle$;
  for $position \in last\_layer$ **do**
    $max\_num\_subtasks := max\{1, max\{|subtasks(r)|, r \in position\}\}$;
    for $offset \in \{0, \ldots, max\_num\_subtasks - 1\}$ **do**
      $new\_position := instantiate(position, \text{offset}, s)$;
      $possible\_facts := possible\_facts \bigcup getPFC(new\_position)$;
      $new\_layer := new\_layer \circ new\_position$
  $task\_network := task\_network \circ new\_layer$;
  $sat\_formula := encode(task\_network)$;
  $(result, solution) := sat\_solver.solve(sat\_formula)$;
  if $result = SAT$ **then**
    return $decode(solution)$;
  $last\_layer := new\_layer$;

Algorithm 1: Simplified Lilotane algorithm

In the *instantiate* function, each possible operation for a new position x in layer l is instantiated. This instantiation is done *lifted* by Lilotane using a concept called *pseudo-constants*. A pseudo constant can be defined as follows:

Definition 12 *A pseudo-constant ϕ is a symbol replacing a free argument α_i of an operation o, which contains an effective domain $dom(\alpha_i)$ forming a subset $dom(\alpha_i) \subset \tau_i$ of the arguments type.*

To understand what 'lifted' means in this context and further illustrate pseudo constants and how they are computed, consider the example shown in Fig. 2.3, which shows a partial instantiation of Lilotane's layers for a problem of the Factory domain. In this example, the root forms a fully ground *m_construct_factory* method that represents the task of constructing factory F_2 at location C using resource R. The *m_construct_factory* method has two subtasks, one for transporting the resource R to location C and one for the actual construction of the factory afterwards. The first subtask is matched by a method *m_get_resource*, which besides the inherited parameters of C and R for destination and resource, introduces two new parameters that do not exist in *m_construct_factory*. The parameter l_1 for the source location of the resource and the parameter f_2 for the factory that produces the resource r_1. These parameters are constrained by the preconditions of the method *m_get_resource* that can be seen on the left. The precondition $produces(f_1, r_1)$ ensures that the parameter f_1 can produce the resource r_1, the precondition $at(f_1, l_1)$ enforces that the factory f_1 exists at location l_1.

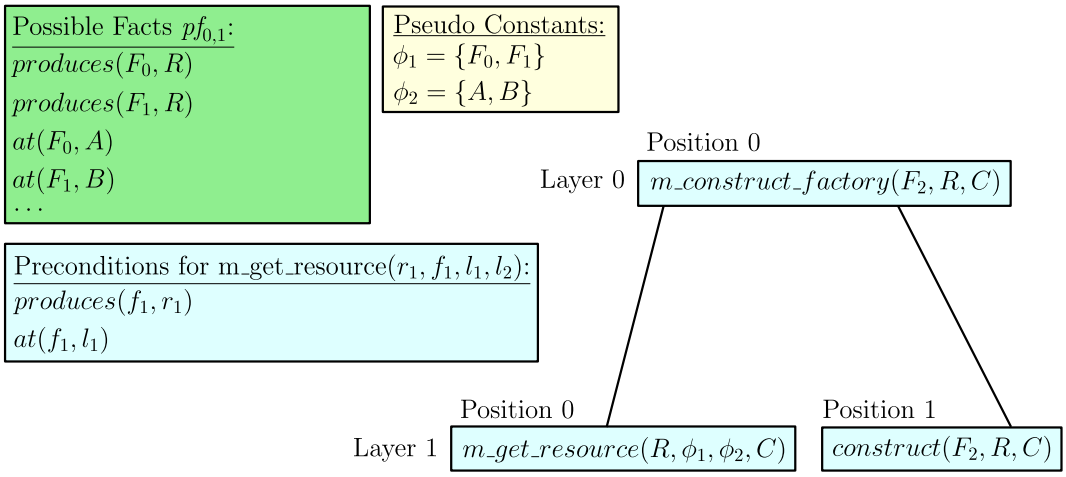

Figure 2.3.: An instantiation example

During the instantiation of an operation, Lilotane uses these preconditions together with the possible facts $\text{pf}_{x,l}$ at the position x and layer l to restrict the operation's new parameters. In the example one can see that the possible facts on the left imply, that the factories which are able to produce resource R, are F_0 and F_1 and that these exist at locations A and B respectively. Thus, Lilotane introduces two new pseudo-constants for the parameters f_1 and l_1, namely ϕ_1 and ϕ_2, whose domains consist of the possible constants for these parameters. These pseudo constants with their possible domain are also encoded into the SAT formula and the SAT solver then tries out the possible combinations of constants when looking for a solution. This lifted procedure using these pseudo-constants avoids a possible combinatorial blowup that could happen during instantiation if an operation was instantiated with all its possible fully ground parameter combinations.

2.3.1.2. Computation of Possible Facts

We will now explain the calculation of the possible facts $\text{pf}_{x,l}$. These are an over-approximation of possible facts that could hold at the position and layer that is currently being instantiated and are computed using the possible fact changes pfc_o for each operation o at the position x. The set $\text{pf}_{x,l}$ contains both positive and negative facts and is instantiated at the beginning of each iteration (layer) with the initial state s_i as the positive facts $\text{pf}^+_{0,l}$ and the empty set for $\text{pf}^-_{0,l}$. Whether a given fact f is reachable at position x is assessed in the following way: In the case that f is positive it is reachable if $f \in \text{pf}^+_{x,l}$, in the case that f is negative it is reachable if $f \in \text{pf}^-_{x,l}$ *or* $\neg f \notin \text{pf}^+_{0,l}$, where the latter case can be intuitively explained as a negative fact being reachable if it could have been caused by an operation at a previous position *or* if it was implicitly true in the initial state, i.e., its positive equivalent was *not* contained in $\text{pf}^+_{0,l}$.

We have already established that $\text{pf}_0 := s_I$. For $x > 0$, $\text{pf}_{x,l}$ is computed in the following way, where $O_{x,l}$ refers to the set of operations possible at position x, l:

$$\text{pf}_{x,l} := \text{pf}_{x-1,l} \cup \bigcup_{o \in O_{x,l}} \text{pfc}_o$$

For an operation o, pfc_o is recursively computed as follows:

1. If o is an operator, pfc_o is simply the *ground hull* of $\text{eff}(o)$, meaning all possible facts that can be achieved by grounding the literals in $\text{eff}(o)$ with all possible combinations of constants.

2. Otherwise
$$\text{pfc}_o := \bigcup_{c \in children(o)} \text{pfc}_c$$
where $children(o)$ are the possible operations that can match o's subtasks.

The possible fact changes of an operation o, when calculated as shown above, can be computed once for every lifted operation in the TOHTN instance domain in a preprocessing stage and to compute the possible fact changes pfc_o for an operation during instantiation, this preprocessed set of possible fact changes is substituted with the constants or pseudo constants contained in the signature of o and then added to $\text{pf}_{x,l}$. Because the computation of pfc_o is not directly dependent on $\text{pf}_{x,l}$ we will refer to this set of possible fact changes as being *state independent*.

2.3.1.3. Retroactive Pruning

During the instantiation of an operation o when variables are restricted using preconditions and the set of possible facts, it can happen that there is no valid combination of constants for a precondition of o that is reachable according to the possible facts at this position $\text{pf}_{x,l}$. The operation is as such not instantiated and pruned. Further, this can cause a subtask of an operation in the previous layer $l - 1$ to have no possible child for at least one of its subtasks, causing the parent operation to be pruned as well. Pruning this operation can then lead to further prunings in layer l and also $l - 2$ and so on and can cause many operations to be pruned retroactively that where instantiated before. While many future operations in further layers are avoided like this, the previously instantiated operations cannot be removed from the SAT encoding, rather it is extended by further clauses that invalidate these invalid operations. Also, the time lost from instantiation of these invalid operations can obviously not be won back. Because of these reasons, it could be beneficial to extend the calculation of possible fact changes to look further ahead while taking into account the current state to prune more operations earlier. The further refinement of this conservative implementation of the calculation of possible fact changes to achieve higher amounts of pruning is the central focus of this thesis.

3. Pruning Techniques

In this chapter we present our contributions to pruning techniques in Lilotane. In the first part, we present the core ideas of our new approach for a finer reachability analysis. In the second part, we introduce the concept of postconditions, and then integrate it into our approach from Section 3.1 to arrive at the final version of our algorithm to compute possible fact changes.

3.1. State Dependent Computation of Possible Facts

As mentioned before, the previous function for computing possible fact changes relied on a precomputed set of lifted possible fact changes per operation, that is substituted at runtime and ground with the appropriate constants and pseudo constants. In contrast, our new algorithm looks ahead by traversing possible subtasks and children of the queried operation, and uses the reachable facts at the current position and possible fact changes found during this traversal to check preconditions. This approach achieves more and/or earlier pruning of operations during the instantiation phase of Lilotane. Pruning can happen in two different ways:

- By descending the tree and checking preconditions, the found set of possible fact changes that is returned can be smaller and more precise than what the previous function would have returned. As such, the new algorithm can cause *indirect* pruning since operations at later positions in the same layer might be pruned earlier or pruned at all, since no possible instantiation of their parameters exists.
- The modified algorithm may detect that one of the possible subtasks of the queried operation has no possible children and as such trigger the *direct* pruning of the queried operation. This latter case obviously has implications on the found set of possible facts, and thus can also cause further indirect pruning.

Our new function for computing possible fact changes is shown in Alg. 2. In addition to receiving an operation o as input like before, the function is now also given $\text{pf}_{x,l}$, the possible (reachable) facts at the current position. Because of this dependence on the current state $\text{pf}_{x,l}$, we refer to the resulting set of possible fact changes as *state dependent* and represent it with $\text{pfc}_{o,x,l}$. Our algorithm always computes $\text{pfc}_{o,x,l}$ by first collecting all (potentially) lifted possible fact changes $\text{lpfc}_{o,x,l}$ and then computing the ground hull of these at the end of the procedure. This is done both for performance reasons, and because it is necessary for our postcondition approach in Section 3.2 to work. The new function also outputs a Boolean value. In the case that it finds a subtask of the queried operation to be impossible, this value is set to *False* (line 11) so that a pruning of this operation at

Input: operation o, $\text{pf}_{x,l}$
Result: Bool, $\text{pfc}_{o,x,l}$

```
1  root := getPreprocessedTree(o);                          // Section 3.1.1
2  nodes_left := initialNodeBudget;                         // Section 3.1.2.3
3  if isPrimitive(root) or root.subtasks.size() > nodes_left then
4  |  lpfc_{o,x,l} := root.pfc;
5  else
6  |  nodes_left := nodes_left − root.subtasks.size();
7  |  lpfc_{o,x,l} := ⟨⟩;
8  |  for s ∈ root.subtasks do
9  |  |  (valid, lpfc_s) := traverse(s, lpfc_{o,x,l}, pf_{x,l}, nodes_left)   // Section 3.1.2
10 |  |  if ¬valid then
11 |  |  |  return (False, {});
12 |  |  lpfc_{o,x,l} := lpfc_{o,x,l} ⋃ lpfc_s;
13 return (True, groundHull(lpfc_{o,x,l}));
```

Algorithm 2: The new algorithm for computing state dependent possible fact changes during instantiation.

the position can be triggered. In this case, $\text{pfc}_{o,x,l}$ is undefined and the function returns an empty set instead.

The new algorithm starts with a retrieval of a preprocessed tree containing an excerpt of the hierarchy of subtasks and children originating from the queried operation, which can be seen in line 1 of Alg. 2. How this tree is computed in preprocessing will be elaborated in the next section. While the interesting case of our new function happens when there are child operations to traverse, it can happen that this tree only consists of a single node and hence cannot be traversed. We call such an operation *pfc-primitive* and the check for this happens in line 3. An operation is pfc-primitive if it is primitive *or* if no children to traverse were computed in the preprocessing step due to a limit given to the preprocessed tree. Further, even for an operation that is not pfc-primitive, the algorithm might not descend if the operation has too many child nodes that would need to be visited. This is also checked in line 3 and was introduced for our dynamic work adjustment that we will elaborate on in Section 3.1.2.3. In both of the previously mentioned cases, the new function behaves like the old function and $\text{lpfc}_{o,x,l}$ is set to the set of fact changes from the preprocessed *root* object (line 4), which are the state independent fact changes for the queried operation.

In the case that the algorithm does descend, all the subtask objects that are pointed to by *root* are traversed in the given order. For this, the *traverse* function is called in line 9, which will be explained in detail in Section 3.1.2. The *traverse* function is given a subtask s, i.e. an unordered set of child operations, the so far found lifted possible fact changes $\text{lpfc}_{o,x,l}$, the set of a-priori reachable facts $\text{pf}_{x,l}$, and nodes_left, the number of nodes that can be further traversed. It returns a Boolean value to indicate an impossible subtask and the set of lifted possible fact changes lpfc_s for the subtask s. The found sets lpfc_s are unified to compute $\text{lpfc}_{o,x,l}$ iteratively (line 12), and if all subtasks were valid the ground hull of $\text{lpfc}_{o,x,l}$ is returned at the end of the algorithm (line 13).

3.1.1. Preprocessing

The preprocessing for our techniques extends the previous preprocessing by the creation of a tree of a limited constant size for every possible (lifted) operation o. The nodes in this tree represent operations and contain the signature of the operation, preconditions, possible fact changes, and, if they are not pfc-primitive, pointers to further child nodes that are organised in an ordered list of sets for each subtask.

The constant size limit of the tree is an integer called $treeSizeLimit$ that represents the maximum amount of nodes in the tree. The tree is computed via a breadth-first search up to the point where the further expansion of an operation, i.e., adding all direct child nodes for all subtasks of this operation, would cause the tree size to exceed the size limit. We check the limit in this manner because it is necessary to expand a node by all of its possible direct children for the correctness of our tree traversal algorithm. A partial expansion of a node may lead to critical effects being omitted. For this reason the computed tree size might not reach the node limit exactly.

All parameters that appear in the signatures, effects or preconditions of the tree nodes are either inherited from the original operation o or are newly introduced by the child operation. These newly introduced parameters are assigned new, unique variable names, and are used for the technique we will present later in Section 3.1.2.2.

3.1.2. Tree Traversal

The algorithm for the traversal is shown in Alg. 3. It is essentially a recursive, depth-first search of the preprocessed tree. As mentioned before, it computes the possible fact changes of the given subtask s and checks whether the subtask is valid. It is given as input the subtask s, the set of a-priori reachable possible facts lpf, the set $\text{pf}_{x,l}$ and a budget of nodes left to traverse nodes_left.

For computing the possible fact changes of the subtask, the *traverse* function iterates over all possible child operations of the subtask. For a single child, it first restricts the child's variables (line 4) and checks its preconditions (line 6), the details of which are explained in the annotated sections. After this, the child might be identified as not valid (line 7). In this case, the child cannot contribute to lpfc_s and the algorithm moves on to the next child. Additionally, for every invalid child that is found and for every child where variables are restricted, nodes_left is increased by a constant amount (lines 8 and 11) which will be explained in the annotated section.

If the child is found to not be invalid, there are two possible scenarios. Firstly, if the child is pfc-primitive or if there are not enough nodes left to traverse and secondly if the converse holds. In the former case, the lifted possible fact changes for the child lpfc_{child} are simply set to the state independent possible fact changes found in the child node (line 13). In the latter case, the child's subtasks are further traversed (lines 13-22).

In the case of further traversal, nodes_left is decreased by the amount of direct children of the child (line 17), since at least all direct children of the child need to be traversed. The traversal is then done iteratively in the order of the subtasks of the child, and lpfc_{child} is iteratively expanded by the found $\text{lpfc}_{subtask}$ sets. During this, another set lpf_$copy$ is also expanded by the found $\text{lpfc}_{subtask}$ sets and given to the traverse function. This set is

instantiated with lpf (line 17) and represents the lifted set of possible fact changes found up to the current subtask. If a subtask is found to be invalid the traversal is aborted (lines 20-21).

Lastly, if the child has not been found to be invalid, *subtask_valid* is set to *True* since a single child being valid is enough for the subtask to be considered valid, and lpfc_{child} is added to lpfc_s (lines 25-27). After all children have been processed, *subtask_valid* and lpfc_s are returned.

Input: subtask s, lpf, $\text{pf}_{x,l}$, *nodes_left*
Result: Bool, lpfc_s

```
1  subtask_valid := False;
2  lpfc_s := ⟨⟩;
3  for child ∈ s do
4      child_valid := restrictVariables(child, lpf, pf_{x,l});          // Section 3.1.2.2
5      if child_valid then
6          child_valid := checkPreconditions(child, lpf, pf_{x,l});     // Section 3.1.2.1
7      if ¬child_valid then
8          nodes_left := nodes_left + nodeIncrease;                     // Section 3.1.2.3
9          continue;
10     if child.hasRestrictedVariables() then
11         nodes_left := nodes_left + nodeIncrease;                     // Section 3.1.2.3
12     if isPrimitive(child) or child.subtasks.size() > nodes_left then
13         lpfc_child := child.pfc;
14     else
15         lpfc_child := ⟨⟩;
16         nodes_left := nodes_left − child.subtasks.size();
17         lpf_copy := lpf;
18         for subtask ∈ child.subtasks do
19             valid, lpfc_subtask := traverse(subtask, lpf_copy, pf_{x,l}, nodes_left);
20             if ¬valid then
21                 child_valid := False;
22                 break;
23             lpf_copy := lpf_copy ⋃ lpfc_subtask;
24             lpfc_child := lpfc_child ⋃ lpfc_subtask;
25     if child_valid then
26         subtask_valid := True;
27         lpfc_s := lpfc_s ⋃ lpfc_child;
28 return (subtask_valid, lpfc_s);
```

Algorithm 3: *traverse* Algorithm

3.1.2.1. Checking Preconditions

The procedure for checking preconditions is shown in Alg. 4. Here, every precondition is checked for *reachability*, and if one precondition is found that is not reachable the set of preconditions is not reachable since they must all hold for an operation to be applicable.

Input: *node*, lpf, $\text{pf}_{x,l}$
Result: Bool
1 **for** $precondition \in node.preconditions$ **do**
2 **if** $\neg isReachable(precondition, lpf, pf_{x,l})$ **then**
3 **return** $False$;
4 **return** $True$;

Algorithm 4: *checkPreconditions* function

Input: *precondition*, lpf, $\text{pf}_{x,l}$
Result: Bool
1 **if** $isRigid(precondition)$ **then**
2 **if** $\neg precondition.negative$ **then**
3 **for** $fact \in groundHull(precondition)$ **do**
4 **if** $fact \in s_I$ **then**
5 **return** $True$;
6 **else**
7 **for** $fact \in groundHull(precondition)$ **do**
8 **if** $\neg fact \notin s_I$ **then**
9 **return** $True$;
10 **else**
11 **if** $\neg precondition.negative$ **then**
12 **for** $fact \in groundHull(precondition)$ **do**
13 **if** $fact \in (pf_{x,l} \bigcup groundHull(lpf))$ **then**
14 **return** $True$;
15 **else**
16 **for** $fact \in groundHull(precondition)$ **do**
17 **if** $fact \in (pf_{x,l} \bigcup groundHull(lpf))$ ***or*** $\neg fact \notin s_I$ **then**
18 **return** $True$;
19 **return** $False$;

Algorithm 5: *isReachable* function

How a preconditions reachability is computed is shown in Alg. 5. There are several differences to the way reachability was defined in 2.3.1.1. Firstly, preconditions might not be fully ground literals, since the parameters might be constants, pseudo constants or free variables. For this reason, we go through the possible groundings of the literal until a reachable grounding is found (lines 3, 7, 12, 16). Secondly, the algorithm differentiates between rigid and fluent preconditions.

For rigid preconditions, the initial state s_I can be used to check their validity, because by definition no operation's effects match a rigid precondition's predicate and so the additional possible fact changes collected in lpf and $\text{pf}_{x,l}$ are irrelevant for the validity of these preconditions. Positive, rigid preconditions are validated by finding a grounding of the precondition that *is* contained in the initial state (line 4), and negative, rigid preconditions are validated by finding a grounding of the precondition whose negation is *not* contained in the initial state.

For fluent preconditions, both sets lpf and $\text{pf}_{x,l}$ need to be considered, because the possible facts of both sets combined might hold before the currently examined operation. This is because both sets are constructed by the possible effects of operations that could be part of a plan in which they are executed before the current operation as to the total node ordering relation defined in the TOHTN domain. However, we cannot just check whether the precondition is contained in the union of lpf and $\text{pf}_{x,l}$, because we keep lpf lifted. Instead, we check whether there exists a grounding of the precondition in the union $(\text{pf}_{x,l} \bigcup \mathit{groundHull}(\text{lpf}))$ (lines 13 and 17). If we do not find such a grounding there is no possibility of the preconditions holding, regardless of which constants would later be substituted for lifted variables, making the operation impossible at the current location. For groundings of negative, fluent preconditions we further check if their positive equivalent is not contained in the initial state if they are not contained in the found possible facts (line 17). While the negative preconditions not being contained in the found possible facts means no operation could have directly caused them at the current position, they might still hold in the initial state. Since negative facts are only implicitly contained in the initial state, we need to check whether the positive equivalent is *not* contained in the initial state to check whether the negative precondition held in the initial state.

Our actual implementation has further efficiency improvements not shown in Alg. 5, which we will touch on in Section 4.1.

3.1.2.2. Variable Restrictions

In the early stages of our experiments we discovered that while descending down a single layer of subtasks and no further in our traversal function lead to significant pruning and speed improvements on some domains, descending down further than this caused little further pruning while causing a significant overhead. Investigating this lead to the concept of variable restrictions that we will introduce in this chapter.

In general, operations that match tasks often introduce parameters beyond the parameters contained in the task's signature. Broadly speaking this means that if we descend the subtasks of an operation recursively like in our DFS algorithm, more and more free variables arise that did not exist in the parent operation. Each precondition p featuring such unbound variables essentially blurs the logic of our procedure: Without further knowledge we must consider p valid as soon as any single ground fact matching p is valid. This in turn causes preconditions to be more and more generic and the collected effects to be highly over-approximative. As a consequence, our procedure finds new invalid preconditions whereas the set of reachable facts increases considerably. Our hypothesis is that these are the reasons we found so little further pruning with a more comprehensive descent of each operation's tree.

Input: *node*, lpf, $\text{pf}_{x,l}$
Result: Bool
1 $valid := True$;
2 **for** $precondition \in (node.preconditions)^+$ **do**
3 **for** $v \in precondition.parameters$ **do**
4 **if** $v \in node.new\_parameters$ **then**
5 **if** $groundHull(precondition).size() < restrictLimit$ **then**
6 $rd_v := \langle\rangle$;
7 **for** $groundFact \in groundHull(precondition)$ **do**
8 **if** $isReachable(groundFact, lpf, pf_{x,l})$ **then**
9 $rd_v.insert(groundFact.parameters[v])$;
10 $rd_v := rd_v \bigcap node.getDomain(v)$;
11 **if** $rd_v.size() = 0$ **then**
12 **return** $False$;
13 **else**
14 $node.setDomain(v, rd_v)$;
15 **return** $True$;

Algorithm 6: *restrictVariables* algorithm

To solve this we introduced *variable restrictions*.

Definition 13 *For a given variable v that is newly introduced by a node in the preprocessed tree, the restricted domain rd_v of this variable is a set of constants that v could be substituted with, constants not in this set cannot be substituted for v.*

In general, variable restrictions are very similar to pseudo constants, with the difference that they only exist temporarily in a single call of the function for computing possible fact changes. They are not used outside of this context like pseudo constants which are used in the encoding. Variable restrictions use the unique variables that are introduced in the preprocessing as explained in Section 3.1.2.1. These free variables are restricted using both rigid and fluent preconditions, similar to how they are restricted in the instantiation mentioned in Section 2.3.1.1. But there are two major differences to that approach. Firstly, we do not restrict all variables we find, we only restrict variables where we can be sure that the time needed to find the set of valid constants is limited by a constant amount. As a consequence, we also ensure that the set of valid constants for a given variable is of a limited size. Secondly, we only consider positive preconditions because we have found through preliminary experiments that using negative preconditions to restrict variables is not effective.

How the variable restriction is done is shown in Alg. 6. Its input are a node *node* whose newly introduced variables shall be restricted, the set lpf and the set $\text{pf}_{x,l}$. It returns *False* if a precondition turns out to not be reachable and *True* otherwise. For every positive precondition we go through every parameter that is a new parameter introduced by the given node (lines 2-4). If the size of the ground hull of this precondition is smaller than the constant *restrictLimit* we go through all facts in the ground hull and check them for reachability with respect to lpf and $\text{pf}_{x,l}$. If they are reachable we add the corresponding

constants to the restricted domain for this parameter (lines 8-9). After this, if the parameter already has a restricted domain, the intersection of *restrictedDomain* and the existing domain contained in the node object is computed (line 10) as the new set rd_v. If the resulting *restrictedDomain* is empty, the precondition is not reachable and in this case the function returns *False*, otherwise the new domain is added to the node.

For the actual implementation of this function we have also made efficiency improvements not shown here which we will touch on in Section 4.1.

3.1.2.3. Dynamic Work Adjustment

While we have found significant prunings on multiple domains, there are many domains where no or little pruning happens and the traversal of the tree adds an overhead without any benefit. Depending on the limit that is used for the size of the preprocessed tree, this overhead can be quite large. At the same time, when comparing domains that do have significant pruning, setting a constant tree size limit might optimally exploit the possible pruning on one domain without unnecessary overhead, while not exploiting the possible pruning on another. In an attempt to solve these challenges, we introduced a dynamic budget for traversing the tree to find a more balanced optimum regarding overhead and pruning.

Our new function for computing possible fact changes differentiates itself from the old function by checking the reachability of preconditions, either to identify invalid preconditions or to restrict variables. If it finds no invalid preconditions or no variable restrictions, the returned set of possible facts is the same as in the old function that was not dependent on the state. Conversely, we hypothesize that the more variables are restricted and the more invalid preconditions are found, the more exact our output is and the more pruning is done. We use this hypothesis to come up with our dynamic traversal approach.

The basis of this traversal are two parameters, namely two integers *initialNodeBudget* and *nodeIncrease*. Our dynamic traversal uses an integer *nodes_left* that limits the amount of nodes which the *traverse* function is allowed to traverse. *nodes_left* is always initialized with the parameter *initialNodeBudget* (Alg. 2 line 2). If a node's subtasks are traversed, the number of direct children in these subtasks is subtracted from *nodes_left* (Alg. 2 line 6, Alg. 3 line 16). Only if there are still enough nodes left to check all of the direct children of a node, it is traversed (Alg. 2 line 3, Alg. 3 line 12). If an invalid node is found or if the variables of a node are successfully restricted, *nodes_left* is increased by the value of *nodeIncrease* (Alg. 3 lines 7-11). Like this, our traversal algorithm is 'encouraged' to traverse further when finding invalid nodes and restricting variables. Appropriate values for each of these parameters will be determined in Section 4.3.

3.2. Postconditions

To motivate the concepts of postconditions we will use an example method from the 'Minecraft' domain [24]. Generally this domain is concerned with placing blocks of different materials in a three-dimensional world. Naturally, a task that exists in this domain is the $place\_block\_abstract(m, l)$ task, which represents the task of placing a

block of a certain material m to a location l represented by a three-dimensional coordinate. A method that matches this task is the $place\_block\_3(m, l, m_2)$ method shown in Fig. 3.1. Intuitively speaking, this method achieves placing a block at the location by removing a block that already exists at the location l of a possibly different material m_2 than m as its first subtask, and then placing a block of the correct material m at the location l.

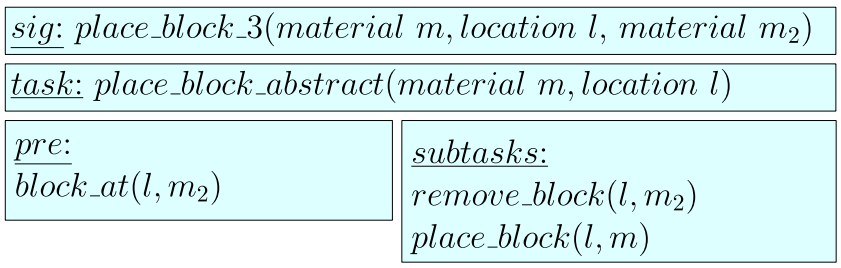

Figure 3.1.: The $place\_block\_3$ method of the Minecraft Domain.

When the possible fact changes are computed by Lilotane for $place\_block\_3(m, l, m_2)$ as described in 2.3.1.1, two literals $empty(l)$ and $\neg empty(l)$ are added: Literal $empty(l)$ is added because the first subtask removes a block at l while $\neg empty(l)$ is added because the second subtask places a block at l. Intuitively speaking, a method responsible for placing a block at a specific location l should not have the literal $empty(l)$ as a possible fact change, since a block *will* exist at location l after the methods execution. In fact, it is not possible that $empty(l)$ holds after the execution of $place\_block\_3(m, l)$. Our concept of *postconditions* solves this issue, and Fig. 3.2 shows the state independent possible facts for $place\_block\_3(m, l)$ computed with and without postconditions.

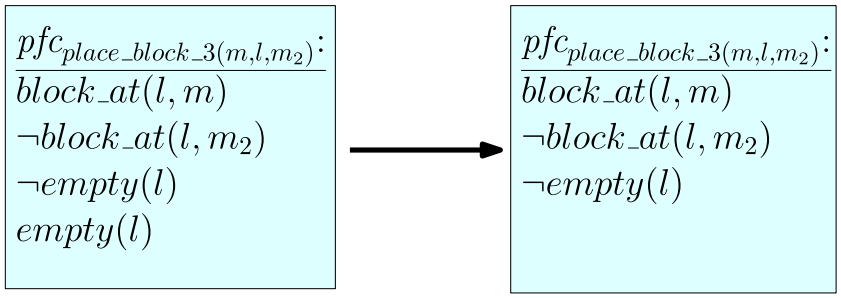

Figure 3.2.: The possible fact changes for the $place\_block\_3$ method of the Minecraft Domain, computed without (left) and with postconditions (right).

3.2.1. Concept

Intuitively speaking, an operations postconditions are a set of literals that always hold after the execution of said operation, similarly to how preconditions must hold before the execution of an operation. In the fundamental publication [9], one of the first formalizations of HTN planning was presented and this formalization allowed HTN problems to define constraints that could among other things be used to enforce certain facts before, between

or after certain tasks. While the input format by which planning problems can be handed to Lilotane does not support the notion of constraining predicates between or after tasks, even if it did, the planning domain might still be possible to model without them and as such, a domain modeler might not have the motivation to encode this extra information. Even the set of preconditions for a given operation is often not as expressive as it could be, and for this reason Lilotane actually already contains a procedure for inferring additional preconditions. Similarly to this, we now present procedures to infer postconditions in Lilotane and use them to make the reachability analysis more accurate as shown in the example in the previous section.

3.2.2. Computation of State Independent Postconditions

We will now describe how the state independent postconditions pc_o for a lifted primitive operation o are computed. For a lifted, primitive operation o with effects eff_o and preconditions pre_o, its state independent postconditions pc_o are computed as:

$$\text{pc}_o := (pre_o \setminus_{pred} \overline{\text{eff}_o}) \cup (\text{eff}_o^- \setminus_{pred} \overline{\text{eff}_o^+}) \cup \text{eff}_o^+$$

This formula is correct because it consists only of literals that always hold after the execution of o. To give an intuitive understand of this, we can summarize the computation as the combination of three sets:

1. The preconditions of o that cannot be contradicted by any effect of o. These hold because they held before the execution of o and since no effect with different polarity and the same predicate exists, they could not be negated by any effect of o.

2. The negative effects of o that cannot be contradicted by any positive effect of o. Similarly, since no positive effect exists of the same predicate for each of these, they hold after the execution of o.

3. The positive effects of o, which always hold after the execution of o by definition.

It might seem too strict to use the relative predicate complement instead of the regular relative complement and deleting only literals that are matched exactly as to their variables in the calculation above, but we have found that this is strictly necessary and the computation is otherwise not correct: While literals of the same predicate with differing polarity and different parameters are not necessarily in direct conflict after they are grounded, we cannot know whether this will be the case with the lifted literals at hand, if the intersections of their respective argument's domains are not empty. Since it is crucial that postconditions of children hold under *all* circumstances for the computation and later usage of postconditions for compound operations, we have to calculate the relative predicate complement of eff_o^+ in eff_o^- and of $\overline{\text{eff}_o}$ in pre_o as shown above.

Since we not only want to compute state independent postconditions for a compound operation, but also use them during the calculation of state independent possible facts of that operation, we will present the computation of state independent postconditions and possible facts for compound operations in an intertwined way. Given a compound operation o with subtasks $subtasks_o$ containing possible child operations, the state independent

1 $\text{pfc}_o := \langle\rangle$;
2 $\text{pc}_o := pre_o$;
3 **for** $s \in subtasks_o$ **do**
4 $\quad \text{pc}_s := \bigcap_{c \in s} \text{pc}_c$;
5 $\quad \text{pfc}_s := \bigcup_{c \in s} \text{pfc}_c$;
6 $\quad$;
7 $\quad \text{pc}_o := \text{pc}_o \setminus_{pred} \overline{\text{pfc}_s}$;
8 $\quad \text{pfc}_o := \text{pfc}_o \setminus \overline{\text{pc}_s}$;
9 $\quad$;
10 $\quad \text{pc}_o := \text{pc}_o \bigcup \text{pc}_s$;
11 $\quad \text{pfc}_o := \text{pfc}_o \bigcup \text{pfc}_s$;

Algorithm 7: Computing state independent postconditions and possible fact changes for compound operations.

postconditions pc_o and possible fact changes pfc_o are computed with the procedure shown in Alg. 7.

This algorithm, starts with the empty set for pfc_o and the preconditions of o as the initial pc_o. It then iterates over all subtasks, repeating the same procedure: First it computes the intersection pc_s of postconditions (line 4) and the union pfc_s of possible fact changes of the children (line 5) for each subtask. Using these subtask postconditions and possible facts, the previously found global possible facts pf_o and postconditions pc_o are reduced. The postconditions pc_o are reduced via relative predicate complement, since any of the possible fact changes with different polarity and the same predicate might annul this postcondition once grounded. The set pfc_o is reduced via regular relative complement and pc_s, because only the exactly matching literal with different polarity can annul an effect with absolute certainty regardless of which constants are later substituted for the free lifted variables. Finally, the procedure unifies the reduced sets pc_o and pfc_o with the newly found sets from the subtask. Even though this algorithm shows the computation for preprocessing state independent postconditions, the computation of state dependent postconditions at runtime follows a very similar pattern as will be seen in the next section.

Alg. 7 together with the procedure to compute state independent postconditions for primitive operations is integrated into the previous computation of state independent possible fact changes in the preprocessing step of Lilotane. Through this altered preprocessing alone, we already found further pruning in many domains.

3.2.3. Computation of State Dependent Postconditions

To achieve even further pruning, we also compute state dependent postconditions during runtime when computing the state dependent possible fact changes $\text{pfc}_{o,x,l}$ for an operation o at location x, l. The updated and final procedure of how this is done can be seen in Alg. 8. As an additional input, it receives the postconditions that hold at the current position $\text{pc}_{x,l}$. How these are calculated will be explained in Section 3.2.4.

The first difference of the updated algorithm can be seen in lines 3 and 4. Line 3 checks whether a the negation of a precondition of the queried operation is contained in $\text{pc}_{x,l}$.

Input: operation o, $\mathrm{pf}_{x,l}$, $\mathrm{pc}_{x,l}$
Result: Bool, $\mathrm{pfc}_{o,x,l}$, $\mathrm{pc}_{o,x,l}$

1 $root := getPreprocessedTree(o)$;
2 $nodes\_left := initialNodeBudget$;
3 **if** $(\overline{root.pre} \cap pc_{x,l}) \neq \emptyset$ **then**
4 | **return** $(False, \{\}, \{\})$;
5 **if** $isPrimitive(root)$ ***or*** $root.subtasks.size() > nodes\_left$ **then**
6 | $\mathrm{lpfc}_{o,x,l} := root.\mathrm{pfc}$;
7 | $\mathrm{pc}_{o,x,l} := (\mathrm{pc}_{x,l} \setminus_{pred} \overline{root.\mathrm{pfc}}) \bigcup root.\mathrm{pc}$;
8 **else**
9 | $nodes\_left := nodes\_left - root.subtasks.size()$;
10 | $\mathrm{lpfc}_{o,x,l} := \langle\rangle$;
11 | $\mathrm{pc}_{o,x,l} := \mathrm{pc}_{x,l} \bigcup root.pre$;
12 | **for** $s \in root.subtasks$ **do**
13 | | $(valid, \mathrm{lpfc}_s, \mathrm{pc}_{o,x,l}) := traverse(s, \mathrm{lpfc}_{o,x,l}, \mathrm{pf}_{x,l}, \mathrm{pc}_{o,x,l}, nodes\_left)$;
14 | | **if** $\neg valid$ **then**
15 | | | **return** $(False, \{\}, \{\})$;
16 | | $\mathrm{lpfc}_{o,x,l} := \mathrm{lpfc}_{o,x,l} \bigcup \mathrm{lpfc}_s$;
17 | | $\mathrm{lpfc}_{o,x,l} := \mathrm{lpfc}_{o,x,l} \setminus \overline{\mathrm{pc}_{o,x,l}}$;
18 **return** $(True, groundHull(lpfc_{o,x,l}), sanitize(pc_{o,x,l}))$;

Algorithm 8: Final algorithm for computing state dependent possible fact changes and postconditions, differences to Alg. 2 highlighted in green.

If that is the case, the operation cannot be possible at this position and *False* is returned to trigger the operation's pruning. The queried operation's preconditions were not been checked in Alg. 2, since the instantiation step for the queried operation already ensured their validity.

Further, the updated algorithm calculates the postconditions $\mathrm{pc}_{o,x,l}$ that hold after the queried operation. In the case that the hierarchy is not traversed, $\mathrm{pc}_{o,x,l}$ is computed by reducing the set $\mathrm{pc}_{x,l}$ with the new possible fact changes and combining what is left with the state independent postconditions found in the root node (line 7). In the case of further traversal, $\mathrm{pc}_{o,x,l}$ is computed iteratively by calling the *traverse* function (line 13). After every subtask, the found postconditions are used to reduce the set of found possible facts in line 17. As an additional output, the updated *getPFC* function then returns the set of postconditions $\mathrm{pc}_{o,x,l}$ that hold after the execution of the current operation. This set is sanitized, meaning all postconditions with parameters that are not constants, or pseudo constants contained in the signature of o, are removed.

Input: subtask s, lpf, $\text{pf}_{x,l}$, pc, *nodes_left*
Result: Bool, lpfc_s, pc_s

```
1  subtask_valid := False;
2  lpfc_s := ⟨⟩;
3  pc_s := ⟨⟩;
4  for child ∈ s do
5      child_valid := restrictVariables(child, lpf, pf_{x,l});
6      if child_valid then
7          child_valid := checkPreconditions(child, lpf, pf_{x,l}, pc);
8      if ¬child_valid then
9          nodes_left := nodes_left + nodeIncrease;
10         continue;
11     if node.hasRestrictedVariables() then
12         nodes_left := nodes_left + nodeIncrease;
13     if isPrimitive(child) or child.subtasks.size() > nodes_left then
14         pc_child := pc \_pred child.pfc;
15         pc_child := pc_child ⋃ child.pc;
16         lpfc_child := child.pfc;
17     else
18         lpfc_child := ⟨⟩;
19         nodes_left := nodes_left − child.subtasks.size();
20         lpf_copy := lpf;
21         pc_child := pc ⋃ child.pre;
22         for subtask ∈ child.subtasks do
23             valid, lpfc_subtask, pc_child := traverse(subtask, lpf_copy, pf_{x,l}, pc_child, nodes_left);
24             if ¬valid then
25                 child_valid := False;
26                 break;
27             lpf_copy := lpf_copy ⋃ lpfc_subtask;
28             lpf_copy := lpf_copy \ pc_child;
29             lpfc_child := lpfc_child ⋃ lpfc_subtask;
30             lpfc_child := lpfc_child \ pc_child;
31     if child_valid then
32         subtask_valid := True;
33         lpfc_s := lpfc_s ⋃ lpfc_child;
34         pc_s := pc_s ⋂ pc_child;
35 return (subtask_valid, lpfc_s, pc_s);
```

Algorithm 9: Final *traverse* algorithm with postconditions, differences to Alg. 3 highlighted in green.

3.2.3.1. Updated Algorithm for Recursive Traversal

The updated and final algorithm for recursive traversal of the hierarchy can be seen in Alg. 9. It too receives a set of postconditions pc that hold before this subtask s. This set is used when checking preconditions in line 7. The updated *checkPreconditions* algorithm can be seen in Alg. 10. In it, preconditions are additionally checked for validity concerning the postconditions pc by checking if the negation of a precondition is contained in pc. Further, the updated *traverse* method computes postconditons for every child pc_{child}. In the case of no further traversal for a child (line 13), the postconditions of the child are computed as the relative predicate complement of $\overline{child}$.pf in pc (line 14), combined with the state independent postconditions of the child node *child*.pc. In the case of further traversal, pc_{child} is instantiated with pc and the preconditions of the child *child.pre* (line 21), and is then iteratively computed via recursive calls to *traverse* in line 23. Additionally, the 'temporary' child postconditions pc_{child} are used to reduce the sets lpf_*copy* and lpfc_{child} in lines 28 and 30. Finally, *traverse* calculates the subtask's postconditions pc_s as the intersection of the child postconditions in line 34, which is returned as an additional output in line 35.

Input: *node*, lpf, $\text{pf}_{x,l}$, pc
Result: Bool
1 **for** *precondition* $\in$ *node.preconditions* **do**
2 **if** $\neg$*precondition* $\in$ *pc* **then**
3 **return** *False* ;
4 **if** $\neg$*isReachable*(*precondition*) **then**
5 **return** *False*;
6 **return** *True*;

Algorithm 10: *checkPreconditions* function using postconditions.

3.2.4. Computing Positional Postconditions

1 ...
2 $\text{pf}_{x+1,l} := \text{pf}_{x,l}$;
3 **for** *operation* $\in (x, l)$ **do**
4 $valid, \text{pfc}_{o,x,l}, \text{pc}_{o,x,l} := getPFC(operation, \text{pf}_{x,l}, \text{pc}_{x,l})$;
5 **if** *valid* **then**
6 $\text{pf}_{x+1,l} := \text{pf}_{x+1,l} \bigcup \text{pfc}_{o,x,l}$;
7 $\text{pc}_{x+1,l} := \text{pc}_{x+1,l} \bigcap \text{pc}_{o,x,l}$;
8 ...
9 ...
10 $\text{pf}_{x+1,l} := \text{pf}_{x+1,l} \setminus \{\text{fact} \in \overline{\text{pc}_{x+1,l}} \mid \text{fact is fully ground}\}$;
11 ...

Algorithm 11: Schematic excerpt of instantiation function using postconditions.

For further pruning, we also use the calculated postconditions per operation o at a given position x to compute postconditions which hold after this position x. These can then be used when instantiating position $x + 1$. A schematic excerpt of the function instantiating a position (x, l) is shown in Alg. 11. It calculates the postconditions holding after the current position $\text{pc}_{x+1,l}$ (line 7) by computing the intersection of the postconditions $\text{pc}_{o,x,l}$ of all operations at the position x. This is correct because these literals will hold after the current position regardless of which of the operations is executed and because one of the operations at a given position must be part of every successful encoding representing a valid plan in Lilotane. For the computation of the possible facts $\text{pf}_{x+1,l}$ for the next position, the individual sets $\text{pfc}_{o,x,l}$ are combined (line 6) as before. Finally, after traversing all operations for the position, the ground literals in $\text{pc}_{x+1,l}$ are used to remove impossible literals from $\text{pf}_{x+1,l}$.

4. Evaluation

In this chapter we will evaluate our pruning techniques presented in the previous chapter. We start off by giving some notes and details on our implementation in Section 4.1 and presenting our evaluation setup in Section 4.2. We will then spend Section 4.3 optimizing the values for the parameters our approach introduces. After this, we continue with an in depth evaluation of our approach per domain on our benchmark set in Section 4.4. Lastly, we compare the new Lilotane version against the old Lilotane version and other state-of-the-art hierarchical planners in terms of performance and plan quality in Section 4.5.

4.1. Implementation

Our extensions of Lilotane were implemented in C++ 17 like the existing Lilotane codebase. Our source code is available at https://github.com/NikolaiLMS/lilotane. For hashsets and and hashmaps we also used the efficent implementation from [1]. As a SAT solver we used Glucose [2] for most runs except those specifically labeled with the suffix '(C)' in Section 4.5, for which we used CaDiCaL [7]. We will now elaborate on some implementation details that improve the efficiency of the techniques from chapter 3.

4.1.1. Reduction of Found Lifted Effects

Input: literal $l1$, literal $l2$
Result: Bool
1 $assert(l1.predicate = l2.predicate)$;
2 $assert(l1.polarity = l2.polarity)$;
3 **for** $argIndex, arg \in enumerate(l1.args)$ **do**
4 | **if** $\neg isFree(arg)$ ***and*** $arg \neq l2.args[argIndex]$ **then**
5 | | **return** $False$;
6 **return** $True$;

Algorithm 12: Function for checking whether the ground hull of a literal $l1$ is a superset of the ground hull of another literal $l2$.

In the implementation, the literals of which the ground hull is computed at the end of algorithms 2 and 8 (lines 13 and 18) are reduced so that less redundant computations are done when computing the ground hull. This is achieved by removing literals whose ground hull is the subset of the ground hull of a different literal in the set. The procedure for checking this for two given literals is shown in Algorithm 12. To give an example, consider a precondition of the form $prec(A, ?)$, and $prec(?, ?)$ where A is a constant and ? stands for

a free variable, i.e., the latter preconditions ground hull contains all possible combinations of possible constants for both parameters. In this case, the latter preconditions ground hull contains all the ground preconditions contained in the former preconditions ground hull, and as such we can remove the former precondition from our set before grounding. Depending on how big the domain of possible constants for the second parameter is, we can save many unnecessary computation steps with this technique and have empirically found this approach to bring a performance improvement.

4.1.2. Preprocessed Tree

The tree that is preprocessed as described in 3.1.1 and then substituted at runtime does not actually contain all the information (preconditions, postconditions, effects) that is needed at runtime. Rather, the effects, preconditions and postconditions exist only once in memory for every lifted operation and are substituted at runtime with the appropriate parameters using a substitution object found in the tree's nodes which makes up most of the node's content. Further, this substitution is only done for every precondition, postcondition and effect just in time when or if it is needed. If, for example, a node's preconditions turn out to be invalid, the postconditions and effects of said node are never substituted as this is not necessary.

4.1.3. Caching Rigid Preconditions for Variable Restrictions

For the restriction of variables described in 3.1.2.2, possible constants for variables belonging to rigid preconditions are determined using a special caching technique. In the preprocessing step of Lilotane, all rigid preconditions in the state are used to create a rigid precondition cache. An example of this computation can be seen in Fig. 4.1. For every rigid precondition in the initial state s_I, every version of this precondition with a single parameter as completely free variable (denoted by '?') is used as a key in a map, that maps to the possible constant values for the free parameter. Since the preconditions are rigid, this map always stays correct, and conversely this type of preprocessing is not possible for fluent preconditions. This map is then used in $restrictVariables$ to determine possible constants for a newly introduced variable. For example, given the precondition $rigid\_precondition(A, Var1)$ where $Var1$ shall be restricted, this map can be used to retrieve the possible constants (B, C) for $Var1$ in constant time instead of having to go through all possible values for $Var1$ and checking each ground precondition against the initial state.

4.1.4. Avoiding Redundant Precondition Checking

In algorithms 2 and 8, if $restrictVariables$ used a precondition to restrict variables and does not find it to be invalid, then this precondition has already implicitly been checked for reachability. Hence, preconditions with this property are not checked again in $checkPreconditions$.

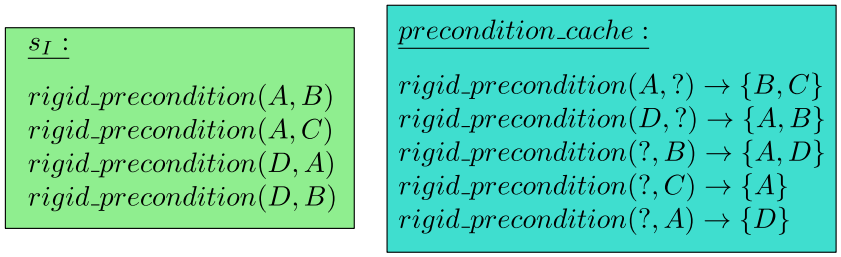

Figure 4.1.: Example computation of rigid precondition cache.

4.2. Evaluation Setup

For the evaluations we used the benchmarks of the 2020 IPC planning competition[4], which comprise a total of 24 diverse domains. We used two machines with 507GB and 550GB of RAM respectively, running Intel Xeon E5-4640 CPUs @ 2.40GHz with 32 cores. We executed up to 28 runs in parallel with a RAM limit of 16GB and a time limit of 1800 seconds. We always executed a single run for each configuration of our planner over the benchmark set. We used the *pandaPIparser*[1] to verify found plans. The experimental data is available at https://github.com/NikolaiLMS/mastersthesis-experimental-data.

4.3. Parameter Evaluation

As a reminder, our approach introduces four parameters that belong to three different concepts:

- 1. The limit used for variable restrictions as explained in 3.1.2.2; $restrictLimit$.
- 2. The limit for the number of nodes in the preprocessed tree for each operation mentioned in 3.1.1; $treeSizeLimit$.
- 3. The two parameters for the dynamic traversal explained in 3.1.2.3; $initialNodeBudget$ and $noceIncrease$.

We will evaluate these parameters in the above shown order in the following sections with different emphasis on performance and amount of pruning, although in the end our main goal is optimizing the performance of our planner. For the measurement of performance and amount of pruning we will use three metrics:

- 1. PAR2-Score: The 'Penalized Average Runtime 2'-Score, is used as our main performance metric. For a finished instance, this score is the number of seconds that the planner needed to find the solution. For an instance which did not finish, the instances score is computed as twice the time limit, so $2 * 1800 = 3600$ in our case. The PAR2-score for a domain is the arithmetic average of the scores of all its instances, and the total PAR2-score for the entire test-set is the arithmetic average over the PAR2-scores of all instances. For this score, lower is better, hence we will sometimes use $3600 - PAR2\text{-}Score$ to represent it.

[1] https://github.com/panda-planner-dev/pandaPIparser

- 2. IPC-Score: This score was used in the 2020 IPC planning competition and for an instance that was finished in t seconds, it is computed as 1.0 if $t < 1.0$ and as $1 - \log(t/1800)$ otherwise. For an instance for which no solution was found, the score is 0. A domains IPC-score is the arithmetic average over the scores of all of its instances, and the total IPC-score is the sum of the IPC-scores for every domain. This score is quite skewed towards solving instances very quickly rather than solving more instances, and we will only use it as a secondary performance metric, e.g., if the other metrics are the same between different configurations we will use the IPC-score as a tie breaker. For this score, higher is better.

- 3. ANC-Score: This is our main metric to compare amounts of pruning, and it is computed as the average number of clauses in the final encoded SAT formula over all finished instances. When we compare this score across the runs of multiple configurations we only use the finished instances that were solved by all configurations to prevent elusive skewing. For this score, lower is better.

For the evaluations in sections 4.3.1 through 4.3.3 we did not use the full IPC benchmark set, but rather a randomly chosen subset of about 70% of the total benchmark, and then use the remaining 30% to evaluate whether our parameters exhibit overfitting properties in Section 4.3.4. For the rest of the evaluations we will then use the complete IPC benchmark set. It should also be noted that the Lilotane version used in this section differs from the version presented in Section 3, in that preconditions were not used to initialize postcondition sets (Alg. 7 line 2, Alg. 8 line 11, Alg. 9 line 21).

4.3.1. Variable Restriction Limit

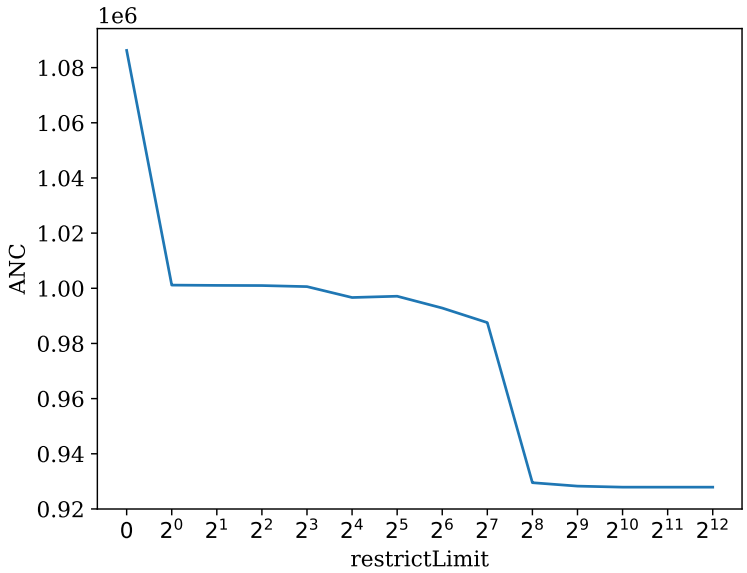

Figure 4.2.: ANC values for runs with different *restrictLimit* values. Lower is better.

In this section we will try to determine the optimal value for the *restrictLimit* parameter. Since the amount of possible combinations for all four parameters mentioned in 4.3 is too big to find the optimum combination by trying out many combinations, we will use a different approach where we will determine the 'optimum' value for each parameter isolated from the others by trying different values for the parameter to optimize and keeping the others fixed. To start, we will determine the optimum parameter for *restrictLimit*. We will try different values for *restrictLimit* while keeping the other parameters fixed. We will keep the parameter *treeSizeLimit* fixed at 1024 and set the dynamic parameters so that our dynamic approach is effectively not used by also setting *initialNodeBudget* to 1024. In this context we will then find the value for *restrictLimit* with the highest amount of pruning measured via ANC as the highest priority and performance as measured by PAR2- and IPC-score as secondary. We do it in this manner, because we want to maximize the potential for pruning by finding an upper limit for the work done in a single node of our tree traversal that still has a meaningful impact on pruning, and then optimize the performance of our planner by optimizing the number of nodes that are traversed in sections 4.3.2 and 4.3.3 using the other parameters.

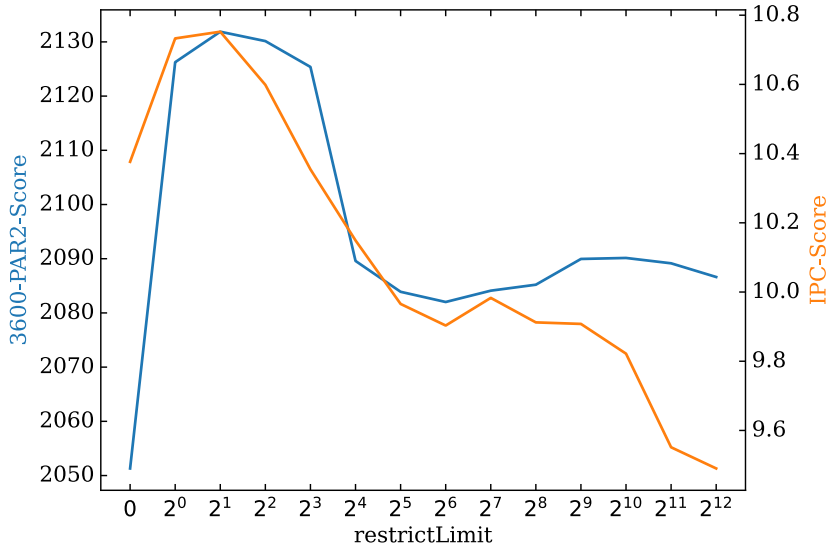

Figure 4.3.: PAR2-scores and IPC-scores for runs with different *restrictLimit* values. Higher is better.

Fig. 4.2 shows ANC for our runs with different values for the *restrictLimit* parameter. One can see two bigger 'jumps' at values 2^0 and 2^8. Since pruning is our main priority we consider only the values of 2^8 and higher as these have the highest ANC-score. Looking at this interval in Fig. 4.3, one can see that the PAR2-score reaches its best value with a *restrictLimit* of 2^9 and the IPC-Score dropping significantly with values 2^{10} and higher. Thus, we choose the value of $2^9 = 512$ for the *restrictLimit* parameter in the further evaluation.

4.3.2. Tree Size Limit

In this section we will evaluate different values for *treeSizeLimit*, while keeping the *restrictLimit* fixed with the value of 512 found in the previous section, and again not using the dynamic approach by setting *initialNodeTraversel* to the value of *treeSizeLimit* for every run. We will look at both performance and amount of pruning and look at differences between domains.

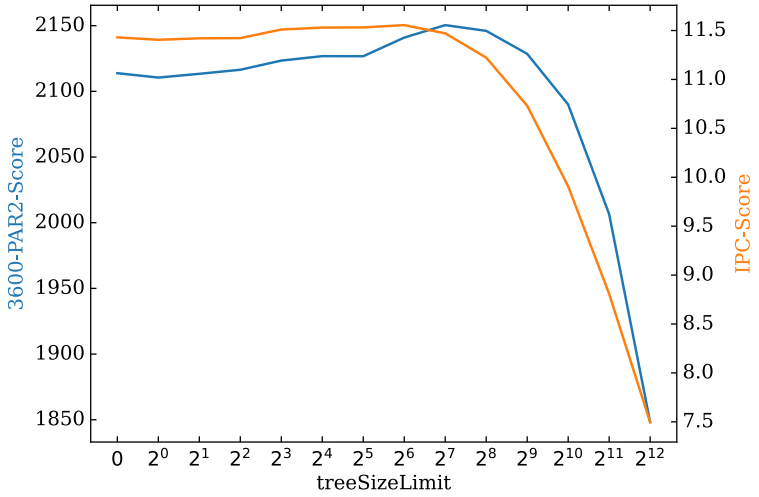

Figure 4.4.: PAR2-scores and IPC-scores for runs with different *treeSizeLimit* values. Higher is better.

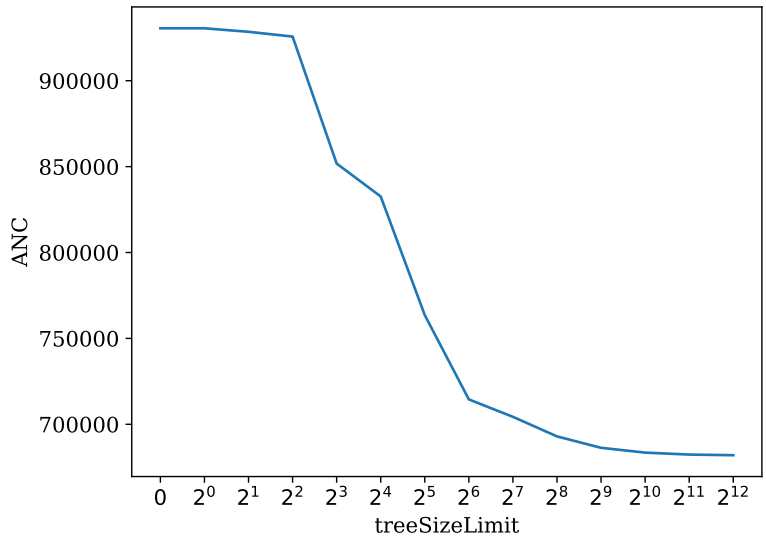

Figure 4.5.: ANC values for runs with different *treeSizeLimit* values. Lower is better.

Fig. 4.4 shows the performance for different *treeSizeLimit* values measured via PAR2- and IPC-score. One can see that the best PAR2 value of 1450 is reached with a *treeSizeLimit* of 2^7, going further away from this value in both directions seems to worsen both PAR2 and IPC-score. When looking at the amount of pruning measured via *ANC* in Fig. 4.5, one can see that even beyond a *treeSizeLimit* of 2^7 the amount of pruning increases further. We will try to exploit more of this pruning using our dynamic work approach in the next section.

4.3.3. Dynamic Work Adjustment Parameters

Our strategy for the determination of the dynamic paramter values goes as follows. Firstly, we will use a *treeSizeLimit* of 128 and determine the minimal values for *initialNodeBudget* and *nodeIncrease* that still achieve (most of) the pruning that happens with an *initialNodeBudget* of 128. These minimal values should also pose the optimum values for PAR2 performance with a *treeSizeLimit* of 128 since they traverse as little nodes as possible to achieve the same amount of pruning. We will then fix the found optimal *initialNodeBudget* and *nodeIncrease* values, and try values higher than 128 for the *treeSizeLimit* and see if we can reach even better performance as measured by PAR2-score.

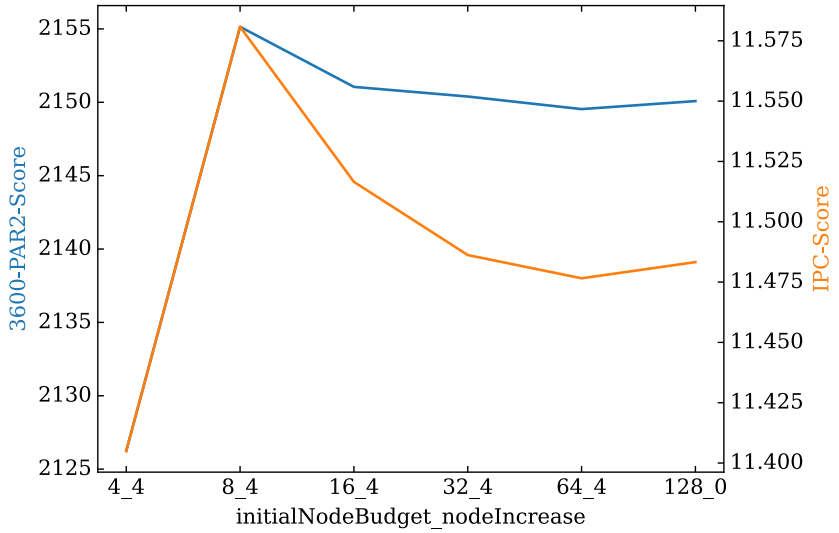

Figure 4.6.: PAR2-scores and IPC-scores for runs with different dynamic parameter values. Higher is better.

Starting with a *treeSizeLimit* of 128 we evaluated different *initialNodeBudget* and *nodeIncrease* values starting with 64 and 1 to try and reach or surpass the PAR2-scores for the 128 column of the table. Fig. 4.6 shows the PAR2-score of the best *initialNodeBudget* and *nodeIncrease* value combination for every *initialNodeBudget* value we tried. It shows that the optimal values regarding PAR2-score are *initialNodeBudget* = 8 and *nodeIncrease* = 4. For smaller values of *initialNodeBudget* we were not able to achieve

enough further pruning, regardless of how big we set *nodeIncrease* and as such we stopped there.

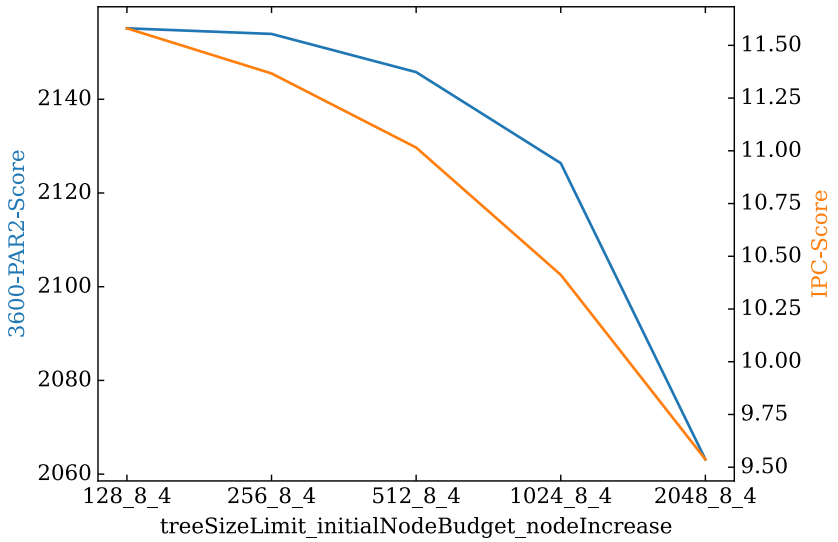

Figure 4.7.: PAR2-scores and IPC-scores for runs with different dynamic parameter values. Higher is better.

Using the values of $initialNodeBudget = 8$ and $nodeIncrease = 4$, we then tried bigger *treeSizeLimit* values. The results of these runs can be seen in Fig. 4.7. As one can see, we could not achieve a better overall *PAR2* and *IPC* score with this method. As such, the final values for our parameters and further evaluations are $restrictLimit = 512$, $treeSizeLimit = 128$, $initialNodeBudget = 8$, $nodeIncrease = 4$. In the further sections we will call our final planner using these parameter values *LilotaneP+* (spoken 'Lilotane Plus').

4.3.4. Evaluation of Overfitting

	Lilotane	LilotaneP+	Δ
Training set	1488.0	1444.9	-2.9%
Test set	1247.0	1245.0	-0.2%
Entire set	1409.9	1379.2	-2.2%

Table 4.1.: PAR2-score for Lilotane and LilotaneP+ on the different versions of the benchmark set. Lower is better.

To check how well our approach can extrapolate on unseen data that we did not use to optimize parameters, we evaluated the finished LilotaneP+ planner on a test set comprising about 30% of the IPC benchmark set. The results can be seen in T. 4.1. It shows that LilotaneP+ does not achieve the improvement over Lilotane measured by PAR2-score

on the test set as it does on the training set. On the entire set, the improvement is much bigger than on the test set but still smaller than on the training set. Our approach seems to not generalize well on the test set based on this data. We believe this to be the case because our approach achieves improvements on selected domains and for an overall improvement on a diverse benchmark set, the balance between exploiting the pruning on domains where our approach is effective and limiting the overhead on domains where it is not effective, is delicate. The data in T. 4.2 supports this hypothesis. Here we see that LilotaneP+ has advantages on multiple domains on both test and training sets, but only on the training set the advantages amount to an overall advantage. On the test set, the overhead on instances where LilotaneP+ is not effective offsets the advantages.

	Lilotane	LilotaneP+
Blocksworld-G	1288.0	**539.1**
Satellite	**547.3**	792.6
Hiking	691.1	**216.1**
Snake	**5.9**	9.2
Elevator	139.4	**109.5**
Minecraft-R	1204.0	**1001.0**
Total	1488.0	**1444.9**

	Lilotane	LilotaneP+
Transport	**564.9**	608.2
Hiking	1202.0	**872.9**
Barman	543.0	**498.8**
Childsnack	**146.9**	458.1
Snake	**0.3**	1.0
Elevator	163.3	**101.2**
Monroe-FO	2.1	**1.9**
Rover-GTOHP	351.4	**294.5**
Total	1247.0	**1245.0**

Table 4.2.: PAR2-scores for Lilotane and LilotaneP+ on the training set (left) and the test set (right). Only domains with differences > 5% in PAR2-score shown. Lower is better.

4.4. Per Domain Evaluation

In this section we will evaluate the effects of LilotaneP+ and how it compares to Lilotane on our benchmark set on each of its respective domains.

The left part of Fig. 4.8 shows the number of clauses for the final SAT formula for each finished instance of LilotaneP+ and Lilotane, where each domain is characterized with a unique symbol. Each datapoint below the diagonal means that the formula of LilotaneP+ was smaller than that of Lilotane and vice versa for points above the diagonal. Additionally, the datapoints beyond the timeout boundary are instances which one of the planners solved while the other one did not. One can clearly see four domains on which LilotaneP+ leads to much smaller formulas, for some instances up to two orders of magnitude smaller, namely Minecraft-Regular, Blocksworld-GTOHP, Hiking and Elevator. LilotaneP+ also solves seven instances more, mostly of these same domains. Surprisingly there are two domains with two obvious outliars in which the formula of LilotaneP+ is much bigger; Transport and Monroe-Fully-Observable. We found out that these increases in the size of the SAT formula are actually not caused by our approach directly. That is to say, for implementing our approach we actually did not start out with the exact source code of the

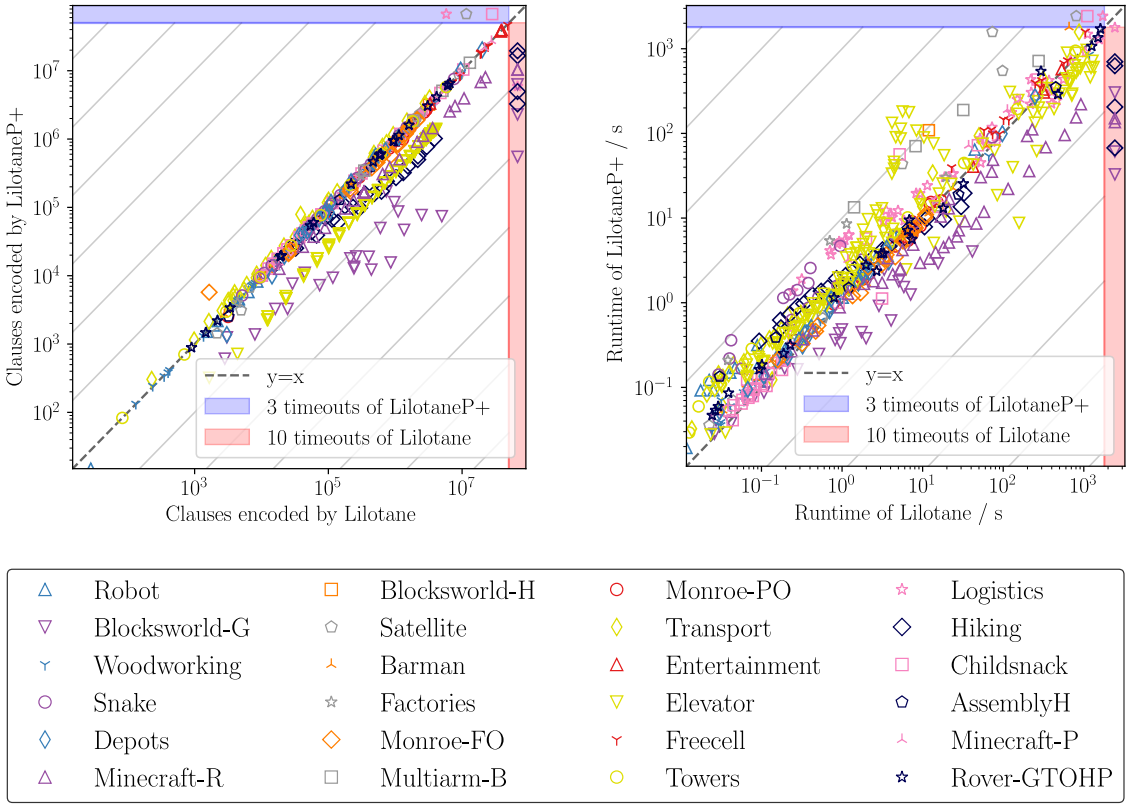

Figure 4.8.: Direct comparison of encoded clauses and runtimes of Lilotane and LilotaneP+. Diagonal lines denote orders of magnitude of difference.

planner we are referring to as 'Lilotane' in this chapter, but rather a slight refactoring of the source code which we will refer to as *LilotaneR*, which causes these increases in the number of encoded clauses. We will briefly discuss the differences of this refactoring and its effects on our evaluation at the end of this section in Section 4.4.2.

Next we will look at the differences in runtime for the finished instances, which are shown in Fig. 4.8 on the right. One can see that the previously identified four instances with the biggest reductions in the size of the SAT formula also lead to lower runtimes, with a reduction of up to about one order of magnitude. For the domains Minecraft-Regular and Blocksworld-GTOHP the improvement in runtime is consistent across most instances. For the Hiking domain, LilotaneP+ actually takes longer to solve the instances that are solved in about ten seconds or less by Lilotane, while solving instances with longer solve times faster than Lilotane and ultimately solving four more instances on the Hiking domain. The Elevator domain behaves similarly, where the advantage of LilotaneP+ only appears to be in the upper right corner of the graph, and otherwise the runtimes are quite scattered across the diagonal. In general, when dismissing the quadrant with runtimes below one second, LilotaneP+ seems to have a slight advantage in runtime, albeit not a consistent one across all domains.

We will now examine differences in clauses encoded and differences in PAR2-score, which are shown in Fig. 4.9 on the left. We chose LilotaneR instead of Lilotane as the

Domain	Δ#*clauses*	Δ*PAR2*
Blocksworld-G	-87.88%	-41.45%
Elevator	-68.92%	-23.78%
Minecraft-R	-51.79%	-17.92%
Hiking	-51.07%	-50.62%
Monroe-FO	-15.38%	-11.74%
AssemblyH	-13.77%	-0.29%
Depots	-12.66%	-0.50%
Monroe-PO	-9.09%	-15.28%
Robot	-7.46%	0.01%
Barman	-6.58%	12.43%
Factories	-5.74%	0.05%
Satellite	-5.09%	4.65%
Woodworking	-4.85%	-1.36%
Freecell	-3.33%	0.20%
Minecraft-P	-1.47%	0.28%
Logistics	-1.00%	-0.57%
Childsnack	-0.19%	3.01%
Entertainment	-0.17%	-1.20%
Rover-GTOHP	-0.08%	1.66%
Snake	0.00%	28.75%
…	…	…
Total	-14.44%	-3.07%

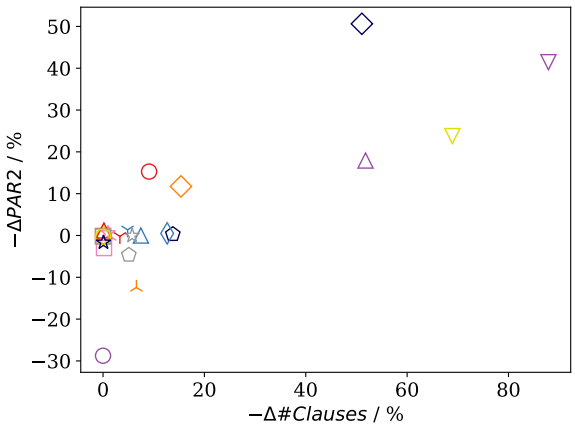

Figure 4.9.: Left: Difference in clauses and PAR2-score per domain of LilotaneP+ and LilotaneR, negative numbers mean lower (better) values for LilotaneP+, only domains with absolute differences > 1% shown, sorted by Δ#*clauses* column. Right: Correlation plot showing the reduction in PAR2-score and reduction in number of clauses encoded per domain. Positive numbers mean higher (better) values for LilotaneP+. Same legend see Fig. 4.8

comparison planner for this table to isolate the effects of our approach and not blur the data with the effects of the refactoring. The column Δ#*clauses* shows the relative difference in the final SAT formula as measured by clause size between LilotaneR and LilotaneP+, averaged over the finished instances (finished by both planners) of every domain. One can see that the four domains we previously identified in Fig. 4.8 as having the biggest difference in clauses encoded also have the highest negative values in the Δ#*clauses* column, and can be seen in the top four rows of the table. These reductions of 50% and higher in the final SAT formula seem to correlate with double digit improvements in the PAR2-score as shown in the Δ*PAR2* column. Beyond the top four instances, there are two more domains where a double digit improvement in PAR2-score can be measured, namely Monroe-FO and Monroe-PO. For the domains Barman and Snake, significant deteriorations of PAR2-score are measured. Interestingly, for the Barman domain this deterioration appears despite a significant decrease in clauses encoded. In general, when

plotting the correlation of reduction in PAR2-score and reduction of clauses, as can be seen in Fig. 4.9 on the right, reductions in clauses encoded of about 10% and higher seem to correlate with significant improvements in runtime measured by PAR2-score, while small reductions are either accompanied by little to no differences or by deteriorations in PAR2-score.

4.4.1. Impact of Postconditions

Domain	Δ#*clauses*	Δ*PAR2*
Blocksworld-G	-85.67%	-41.48%
Elevator	-68.92%	-26.68%
Monroe-FO	-4.34%	-5.65%
Rover-GTOHP	-0.06%	-4.16%
Monroe-PO	-2.52%	-4.02%
Logistics	-0.99%	-3.83%
Freecell	-3.31%	-0.54%
Depots	-2.52%	-0.21%
Hiking	-0.12%	1.92%
Childsnack	-0.19%	3.43%
Satellite	-5.09%	5.39%
Snake	0.00%	8.31%
Barman	-1.92%	12.76%
…	…	…
Total	-7.39%	-1.13%

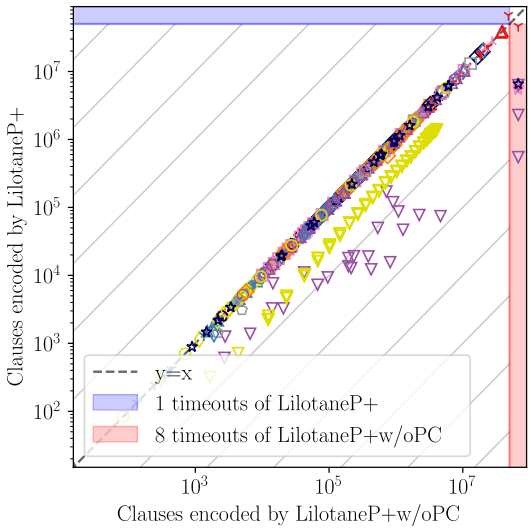

Figure 4.10.: Left: Difference in clauses and PAR2-score per domain of LilotaneP+ and LilotaneP+w/oPC, negative numbers mean lower (better) values for LilotaneP+, only domains with absolute differences > 1% shown, sorted by Δ*PAR2* column. Right: Direct comparison of encoded clauses of Lilotane and LilotaneP+w/oPC. Diagonal lines denote orders of magnitude of difference.

While LilotaneP+ consists of both the general approach described in Section 3.1 and the postcondition approach from Section 3.2, the general approach can be used without postconditions. We used this version without postconditions, which we will call *LilotaneP+w/oPC*, to evaluate the impact of our postcondition approach. We have drawn a scatter plot showing the difference in number of clauses encoded which can be seen in Fig. 4.10 on the right. It shows that the clause reduction on the domains Elevator and Blocksword-G that we have seen before when comparing LilotaneP+ and Lilotane, are caused by the postcondition approach and that seven additional instances are solved when using postconditions.

Further, when looking at differences in PAR2-score between LilotaneP+w/oPC and LilotaneP+ in the table in Fig. 4.10 on the left, it shows that the improvements in PAR2-score on Blocksworld-G and Elevator are also caused by the postconditions as expected. Further there are additional significant improvements in PAR2-score on 4 more domains, and

significant decreases in PAR2-score on 5 additional domains. Overall, using postconditions improves the PAR2-score by 1.13% and causes an average of 7.39% less clauses encoded per instance.

4.4.2. Effects of Refactoring

Domain	Δ#*clauses*	Δ*PAR2*
Transport	25.77%	3.97%
Monroe-FO	16.52%	8.43%
Depots	8.86%	0.64%
Rover-GTOHP	0.04%	-2.17%
Monroe-PO	0.00%	20.37%
Satellite	0.00%	19.62%
Entertainment	0.00%	1.37%
Minecraft-R	0.00%	6.08%
Snake	0.00%	7.27%
Childsnack	0.00%	49.64%
Elevator	0.00%	-3.58%
Barman	0.00%	-2.25%
Logistics	-2.39%	0.70%
Freecell	-3.04%	0.54%
...	...	...
Total	1.92%	1.17%

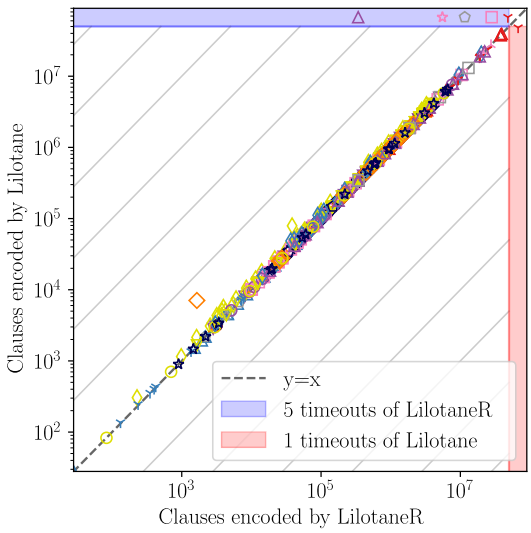

Figure 4.11.: Left: Difference in clauses and PAR2-score per domain of Lilotane and LilotaneR, positive numbers mean higher (worse) values for LilotaneR, only domains with absolute differences > 1% shown, sorted by Δ#*clauses* column. Right: Direct comparison of encoded clauses of Lilotane and LilotaneR. Diagonal lines denote orders of magnitude of difference.

When looking at the difference between Lilotane and LilotaneR in terms of the number of clauses encoded in Fig. 4.11 on the right, one can see that the previously identified domains of Monroe-FO and Transport have higher numbers of clauses encoded than Lilotane. Additonally, the domain 'Depots' exhibits this behaviour aswell, which can be seen in the table on the left of Fig. 4.11. We do not know why this is the case. However, the table also reveals significant worsenings in PAR2-score in multiple domains, e.g. Childsnack, Satellite and Monroe-PO, which do not exhibit an increase in the encoding size. A possible explanation for this is the lack of a certain caching technique in the refactored version: Because LilotaneR only uses state independent possible fact changes, the lifted set of possible fact changes is always the same for every operation which means that the ground hull can be partly precomputed and cached. This caching technique is not something we can adapt to our algorithm since our set of lifted possible fact changes is state dependent and different depending on when the operation is queried.

Either way, our approach needed to overcome this overhead of about 1.17% measured by PAR2-score to improve upon Lilotane overall. If the exact reasons for the overhead could be identified and mitigated, our approach could have a bigger impact.

4.5. State of the Art Comparison

In this section we will compare LilotaneP+ and Lilotane to other state-of-the-art TOHTN planners. These planners are the following:

- *HyperTensioN* [16], a greedy, lifted progression search planner written in Ruby.
- *PandaSAT* [3], a SAT-based planner that works on a grounded version of the problem.
- *PandaGBFS* [13], a ground planner that uses a heuristic progression search. We executed it using the 'RC2(add)' heuristic.

We will compare both planner performance and quality of the found plans. We executed one run per planner and instance of the entire IPC benchmark set using the parallel setup described in 4.2.

4.5.1. Performance

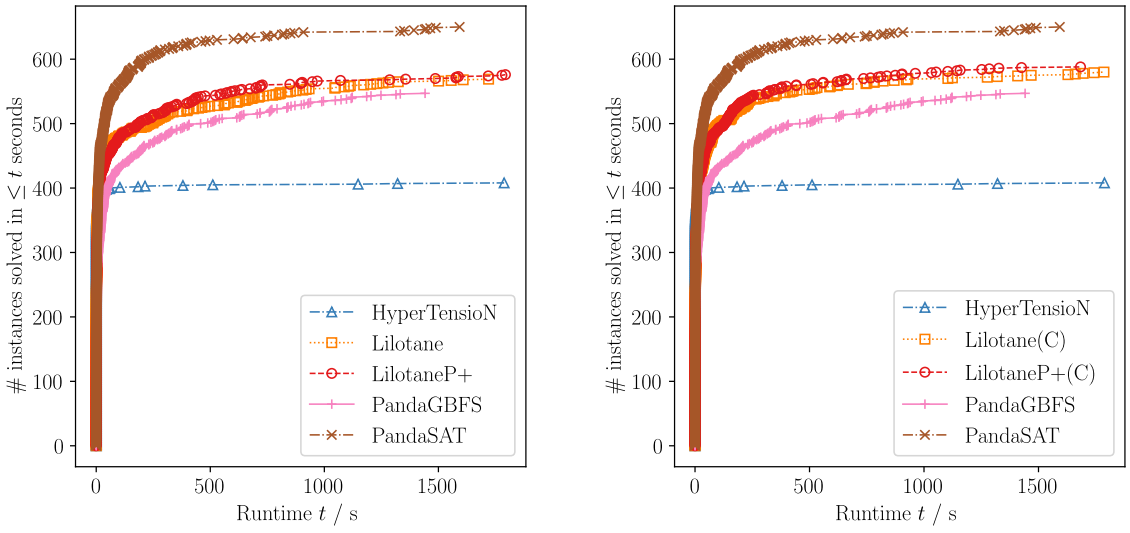

Figure 4.12.: Runtimes of evaluated planners.

For the initial performance evaluation we will look at the cactus graphs in Fig. 4.12. We used two different version of LilotaneP+ and Lilotane each, one using the SAT solver Glucose (Fig. 4.12 left) and one version using the SAT solver CaDiCaL (4.12 right). When using Glucose, LilotaneP+ shows a small but significant advantage over Lilotane, and it extends the advantage over PandaGBFS further. When using CaDiCaL, both planners

perform significantly better than with Glucose. While the advantage of LilotaneP+ over Lilotane seems to dwindle a bit when using CaDiCaL, LilotaneP+ can solve the same number of instances as Lilotane with an about 800 seconds lower time limit per instance. PandaSAT outperforms all other planners significantly, solving about 60 more instances than LilotaneP+. HyperTensioN comes in last, only solving about 400 instances.

Domain	HyperTensioN		Lilotane(C)		LilotaneP+(C)		PandaGBFS		PandaSAT	
Metric	PAR2	Cvg.	PAR2	Cvg.	PAR2	Cvg.	PAR2	Cvg.	PAR2	Cvg.
Robot	**1922.7**	**0.47**	2281.1	0.37	2281.1	0.37	2280.1	0.37	2280.3	0.37
Blocksworld-H	**960.2**	**0.73**	3480.4	0.03	3483.4	0.03	2803.6	0.23	2942.4	0.20
Monroe-PO	3600.0	0.00	**7.3**	**1.00**	7.8	**1.00**	1592.3	0.60	258.1	0.95
Logistics	2610.9	0.28	1593.9	0.61	1571.5	0.61	1890.3	0.50	**62.7**	**1.00**
Blocksworld-G	1866.1	0.50	843.4	0.77	494.1	0.87	**34.6**	**1.00**	772.0	0.80
Satellite	3600.0	0.00	266.2	0.95	529.4	0.90	109.8	**1.00**	**55.1**	**1.00**
Transport	2093.3	0.42	478.2	0.88	478.2	0.88	1152.6	0.70	**30.5**	**1.00**
Hiking	600.1	0.83	846.6	0.77	**414.4**	**0.90**	637.3	0.83	976.5	0.73
Woodworking	3064.6	0.15	460.6	**0.88**	**460.4**	**0.88**	1895.8	0.47	776.2	0.80
Barman	360.1	0.90	444.5	0.90	**317.7**	**0.95**	1810.2	0.50	324.5	**0.95**
Entertainment	3600.0	0.00	2217.1	0.42	2237.9	0.42	81.1	**1.00**	**1.8**	**1.00**
Childsnack	168.2	**0.97**	124.5	**0.97**	**124.3**	**0.97**	1102.4	0.70	843.5	0.77
Snake	**0.2**	**1.00**	5.0	**1.00**	6.4	**1.00**	16.3	**1.00**	26.2	**1.00**
Factories	3060.3	0.15	2881.3	0.20	2883.2	0.20	**2171.7**	**0.40**	2216.2	**0.40**
Elevator	1322.6	0.63	43.5	**1.00**	37.7	**1.00**	192.1	**1.00**	**4.6**	**1.00**
AssemblyH	3360.2	0.07	3008.4	0.17	3011.1	0.17	3122.0	0.13	**2930.6**	**0.20**
Depots	764.8	0.80	740.3	0.80	736.5	0.80	1115.2	0.70	**372.4**	**0.90**
Monroe-FO	3600.0	0.00	4.5	**1.00**	**4.2**	**1.00**	93.1	**1.00**	35.4	**1.00**
Freecell	3600.0	0.00	2970.1	**0.20**	**2956.7**	**0.20**	3600.0	0.00	3014.8	0.17
Minecraft-P	**2700.1**	**0.25**	2926.8	0.20	2934.1	0.20	3092.7	0.15	2800.0	**0.25**
Minecraft-R	**490.4**	**0.86**	1290.5	0.66	1238.4	0.66	832.6	0.80	859.1	0.80
Multiarm-B	3356.9	0.07	3412.0	0.05	3420.7	0.05	2932.3	0.19	**2801.9**	**0.24**
Towers	1980.1	0.45	1849.2	0.50	1857.4	0.50	**1463.6**	**0.60**	2345.9	0.35
Rover-GTOHP	**123.4**	**0.97**	845.2	0.80	713.8	0.83	1584.2	0.57	352.9	0.93
Total	2033.5	10.50	1375.9	15.11	1341.7	15.38	1483.6	14.44	**1128.5**	**16.81**

Table 4.3.: PAR2-scores and coverage scores for the different planners on the IPC benchmark set. For the PAR2-score, lower is better and for the Coverage higher is better. Best values per domain are marked in bold. The rows highlighted in blue (red) show which domains LilotaneP+(C) performs significantly better (worse) on than Lilotane(C) measured by PAR2-score.

Further, we have computed the PAR2-scores as well as the coverage (solved percentage of available instances) per domain and assembled these in T. 4.3, where the respective best values per domain are marked bold. For this table we used the SAT solver CaDiCaL for Lilotane and LilotaneP+ as the previous data showed an advantage over Glucose. The total

coverage score is simply the sum of all domain coverage scores, and the total PAR2-score was computed as the average of the domain scores.

The total PAR2-scores and coverage basically express the hierarchy of performance already seen in 4.12. For the the total scores in both metrics, PandaSAT is quite ahead of everyone, LilotaneP+ and Lilotane rank second and third respectively, then follows PandaGBFS and HyperTensioN comes in last. When looking at the different domains, the picture becomes more nuanced. In terms of PAR2-score, PandaSAT wins eight domains, HyperTensioN and LilotaneP+ win six domains each, Lilotane wins one domain and PandaGBFS wins 3 domains. One should note that LilotaneP+ would win 7 domains if not competing against Lilotane, and Lilotane would win 5 domains if not competing against LilotaneP+. The coverage scores generally follow the pattern of the PAR2-scores. To summarize, LilotaneP+ improves the overall performance of Lilotane by a small but significant margin, measured by better coverage and PAR2-score. On the Hiking and Barman domains, LilotaneP+ improves the performance by such a big margin that it now beats all other planners on these domains. The improvements on the Blocksworld-G, Elevator, Minecraft-R, and Rover-GTOHP domains are also significant, but not enough to win them. Interestingly, when using CaDiCaL as the SAT solver, LilotaneP+ achieves a significant PAR2-score improvement on the Rover-GTOHP domain, which could not be observed before when using Glucose (compare tables 4.3 and 4.9).

Finally, it should be noted that HyperTensioN and PandaGBFS performed significantly worse than they did in [16] and [13]. This could be due to a multitude of reasons, e.g., executing them in parallel on the same machine instead of sequentially, executing them on a different CPU, only using one run per instance and planner, not randomizing the planners seeds etc.. Since all planners were evaluated under the same conditions we consider our evaluation to be fair, but the difference for the named planners performance should still be noted and possibly investigated in future work.

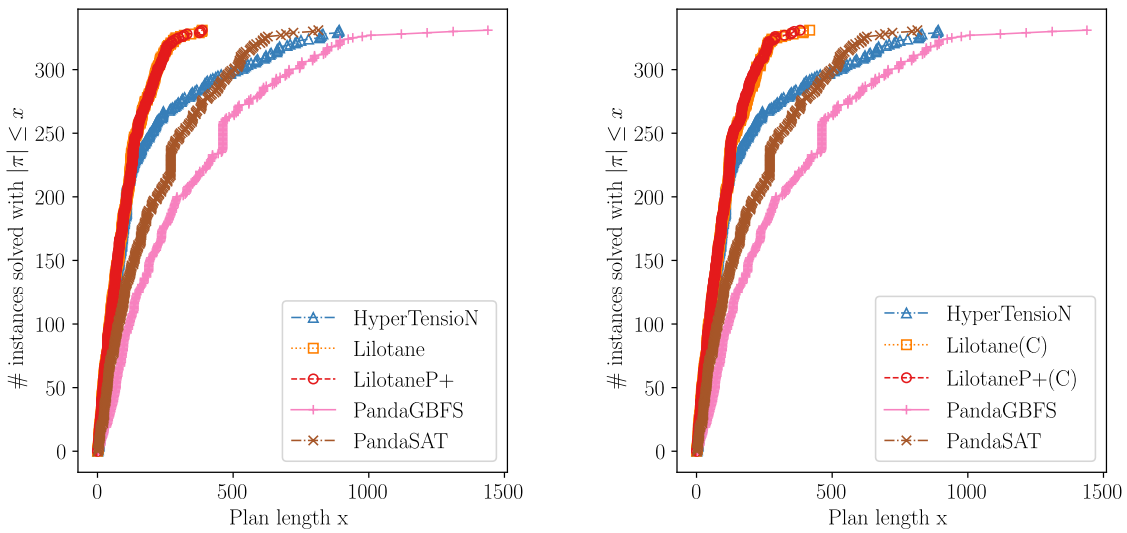

Figure 4.13.: Found plan lengths of the evaluated planners.

4.5.2. Plan Quality

For the evaluation of plan quality we have drawn a cactus plot in Fig. 4.13 showing the amount of plans found per plan length, using only the instances for which *every* planner found a solution. We again used two different versions of LilotaneP+ and Lilotane each, one using the SAT solver Glucose (Fig. 4.13 left) and one version using the SAT solver CaDiCaL (Fig. 4.13 right). In terms of quality, Lilotane and LilotaneP+ have a clear advantage over all other planners. The quality of plans output by PandaGBFS ranks last, while HyperTensioN and PandaSAT rank quite similarly about 'halfway' between the Lilotane planners and PandaGBFS.

5. Conclusion

We have presented pruning techniques for lifted SAT-based hierarchical planning with Lilotane. Our techniques work by improving the reachability analysis in Lilotane's instantiation phase to be more accurate, and ultimately leverage additional computation in this phase of the Lilotane algorithm to reduce the time that is spent encoding and SAT solving in later phases by pruning impossible operations before they are encoded. Our general approach works by finding more accurate possible effects for an operation via a dynamic, look-ahead traversal of the hierarchy originating from that operation, during which we check preconditions to identify impossible operations and to restrict the possible constants for variables occuring in that hierarchy with our concept called *variable restrictions*. In addition to this, we have introduced the novel concept of postconditions into Lilotane and successfully integrated it into our general approach.

We have evaluated our approach on a diverse benchmark set, and we have shown that our approach improves both the overall performance and coverage of Lilotane. On some domains of the benchmark set it achieves a reduction of the number of clauses in the encoded SAT formula of up to two orders of magnitude as well as runtime improvements of up to one order of magnitude. For an overall improvement in terms of performance, exploiting the improvements on the domains of the benchmark set where our approach is effective has to be carefully balanced against the overhead on domains where it is not effective. We have seen that this balance is fickle and heavily depends on the values given to our hyperparameters, which were prone to overfitting the data used to optimize them. Further, we have seen that the postcondition concept is an effective addition to our general concept and is responsible for much of our performance improvements. When comparing our approach to other state-of-the-art TOHTN planners, it improves Lilotane's advantage in terms of performance to previously beaten planners and makes Lilotane the best performing planner on seven to previously only five out of 24 domains. In terms of quality of the output plans, our approach is able to sustain Lilotane's previously held advantage over all other planners.

5.1. Outlook

There are multiple possibilities for future work to be done and extending our approach by using the insights we gained in this thesis.

Since we have found big differences in performance depending on which SAT solver was used, a possibility for future work is a SAT solver case study comparing the performance of different SAT solvers in conjunction with Lilotane and our extensions. Based upon this, our parameter optimization could then be redone using the best performing SAT

solver. The parameter optimization could also generally be redone using a state-of-the-art parameter optimizer like ParamILS [14] or SMAC [15] instead of our chosen approach.

In terms of balancing the amount of work that is done in the function for computing possible fact changes, an approach worth trying might contain shifting more work to earlier positions in a given layer, i.e., doing a deeper more thorough traversal of possible further operations at earlier positions. Since possible facts in the generalized world state are only removed if an according postcondition is found, which is rare, and otherwise are deemed reachable for the entire rest of the layer, pruning these, if possible, earlier could be more effective than our presented approach whose dynamic traversal algorithm is the same for every position in a layer.

Further, the dynamic part of our traversal could be extended. Currently, the dynamic traversal only uses found invalid nodes and restricted variables as information to cause further traversal via the node budget. An additional piece of information to consider could be the size of the set of already found, state dependent possible fact changes in comparison to the size of the preprocessed set of state independent possible fact changes of the queried operation: One could hypothesise that the smaller this difference is, the less possible fact changes can be pruned and the less further traversal might be worth it. For finding out what information to further consider, the domains of the benchmark set on which our approach caused the least pruning and the biggest overhead could be analysed. With a more sophisticated traversal algorithm, one could then also consider getting rid of the parameter that limits the size of the preprocessed tree, and allowing the algorithm to dynamically decide at runtime how deep to traverse the hierarchy.

One could also consider an alternative approach where multiple layers are instantiated in a single iteration instead of just one, and then a depth-first search is used to instantiate this number of layers. This could be viewed as an adaptation of our look-ahead traversal to the general Lilotane procedure. Since our approach has shown that a limited amount of ‘look-ahead’ work can bring improvements, changing the general procedure in a similar way might as well. A caveat of this approach would be that Lilotane might lose its feature of finding a solution at the minimum expansion depth of the hierarchy, since this limit might be ‘overshot’ when instantiating multiple layers.

Bibliography

[1] Martin Ankerl. *robin_hood unordered map & set*. `https://github.com/martinus/robin-hood-hashing` Accessed: 2021-11-13. 2021.

[2] Gilles Audemard and Laurent Simon. "Predicting learnt clauses quality in modern SAT solvers". In: *Twenty-first International Joint Conference on Artificial Intelligence*. 2009.

[3] Gregor Behnke. "Block Compression and Invariant Pruning for SAT-based Totally-Ordered HTN Planning". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 31. 2021, pp. 25–35.

[4] Gregor Behnke, Daniel Höller, and Pascal Bercher, eds. *Proceedings of 10th International Planning Competition: Planner and Domain Abstracts – Hierarchical Task Network (HTN) Planning Track (IPC 2020)*. 2021.

[5] Gregor Behnke, Daniel Höller, and Susanne Biundo. "totSAT-Totally-ordered hierarchical planning through SAT". In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[6] Gregor Behnke et al. "On succinct groundings of HTN planning problems". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 06. 2020, pp. 9775–9784.

[7] Armin Biere et al. "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020". In: *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*. Ed. by Tomas Balyo et al. Vol. B-2020-1. Department of Computer Science Report Series B. University of Helsinki, 2020, pp. 51–53.

[8] Mark S Boddy et al. "Course of Action Generation for Cyber Security Using Classical Planning." In: *ICAPS*. 2005, pp. 12–21.

[9] Kutluhan Erol, James Hendler, and Dana S Nau. "HTN planning: Complexity and expressivity". In: *AAAI*. Vol. 94. 1994, pp. 1123–1128.

[10] Alex Fukunaga et al. "Aspen: A framework for automated planning and scheduling of spacecraft control and operations". In: *Proc. International Symposium on AI, Robotics and Automation in Space*. 1997, pp. 181–187.

[11] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[12] Malte Helmert and Hauke Lasinger. "The Scanalyzer domain: Greenhouse logistics as a planning problem". In: *Twentieth International Conference on Automated Planning and Scheduling*. 2010.

[13] Daniel Höller and Gregor Behnke. "Loop Detection in the PANDA Planning System". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 31. 2021, pp. 168–173.

[14] Frank Hutter et al. "ParamILS: an automatic algorithm configuration framework". In: *Journal of Artificial Intelligence Research* 36 (2009), pp. 267–306.

[15] Marius Lindauer et al. "SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization". In: *arXiv preprint arXiv:2109.09831* (2021).

[16] Maurício C Magnaguagno, Felipe Rech Meneguzzi, and LAVINDRA DE SILVA. "HyperTensioN: A three-stage compiler for planning". In: *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS), 2020, França.* 2020.

[17] Amol Dattatraya Mali and Subbarao Kambhampati. "Encoding HTN Planning in Propositional Logic." In: *AIPS*. 1998, pp. 190–198.

[18] Dana Nau et al. "SHOP: Simple hierarchical ordered planner". In: *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*. 1999, pp. 968–973.

[19] Dana S Nau et al. "SHOP2: An HTN planning system". In: *Journal of artificial intelligence research* 20 (2003), pp. 379–404.

[20] Abdeldjalil Ramoul et al. "Grounding of HTN planning domain". In: *International Journal on Artificial Intelligence Tools* 26.05 (2017), p. 1760021.

[21] Dominik Schreiber. "Lilotane: A lifted SAT-based approach to hierarchical planning". In: *Journal of Artificial Intelligence Research* 70 (2021), pp. 1117–1181.

[22] Dominik Schreiber, Damien Pellier, Humbert Fiorino, et al. "Tree-REX: SAT-based tree exploration for efficient and high-quality HTN planning". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 29. 2019, pp. 382–390.

[23] Dominik Schreiber et al. "Efficient SAT encodings for hierarchical planning". In: *11th International Conference on Agents and Artificial Intelligence (ICAART 2019)*. 2019.

[24] Julia Wichlacz, Alvaro Torralba, and Jörg Hoffmann. "Construction-planning models in minecraft". In: (2019).

A. Appendix

A.0.1. Misc. Metrics of LilotaneP+ per Domain

Domain	# slv.	Δ#*clauses*	Δ*PAR2*	IRP	IFP	IFPPC	IOS	IOPC
Blocksworld-G	26.00	-87.88%	-41.45%	0.00	533.38	173.58	25.92	4.85
Elevator	147.00	-68.92%	-23.78%	0.00	258.72	201.46	0.00	31.97
Minecraft-R	41.00	-51.79%	-17.92%	19033.71	998.00	0.02	0.00	0.00
Hiking	27.00	-51.07%	-50.62%	10496.93	0.11	0.00	39.26	0.00
Monroe-FO	20.00	-15.38%	-11.74%	1037.40	199.70	52.90	4.60	0.00
AssemblyH	5.00	-13.77%	-0.29%	29.00	16.40	0.00	2.00	0.00
Depots	24.00	-12.66%	-0.50%	0.00	64.54	4.62	0.92	0.00
Monroe-PO	20.00	-9.09%	-15.28%	1532.95	238.75	98.70	3.60	0.00
Robot	11.00	-7.46%	0.01%	0.00	787.91	771.64	4.00	0.00
Barman	17.00	-6.58%	12.43%	0.00	32.35	4.12	0.00	4.12
Factories	4.00	-5.74%	0.05%	12077.00	46.25	1.00	0.00	0.00
Satellite	14.00	-5.09%	4.65%	0.00	362.14	357.86	0.00	1.00
Woodworking	35.00	-4.85%	-1.36%	2.46	249.34	0.00	0.00	0.00
Freecell	13.00	-3.33%	0.20%	737.38	49103.31	46355.00	5.46	0.00
Minecraft-P	4.00	-1.47%	0.28%	1096.00	709.00	0.25	0.00	0.00
Logistics	44.00	-1.00%	-0.57%	0.00	444.82	391.32	0.00	2.00
Childsnack	28.00	-0.19%	3.01%	0.00	0.00	0.00	0.00	0.00
Entertainment	5.00	-0.17%	-1.20%	10.40	31.20	0.00	0.20	0.00
Rover-GTOHP	23.00	-0.08%	1.66%	10.61	4.00	0.17	0.00	0.00
Transport	34.00	-0.04%	-0.02%	0.41	0.00	0.00	0.00	0.00
Multiarm-B	4.00	0.00%	0.26%	0.00	55.50	0.00	0.00	0.00
Snake	20.00	0.00%	28.75%	0.00	4.35	0.00	0.00	0.00
Towers	10.00	0.00%	0.30%	0.00	36.90	0.00	0.00	0.00
Blocksworld-H	1.00	0.00%	0.09%	0.00	92865.00	92823.00	0.00	0.00

Table A.1.: Different metrics for finished instances by LilotaneP+ averaged over finished instances per domain, sorted by column Δ#*clauses*. From left to right: # slv.: number of solved instances, Δ#*clauses*: difference in number of clauses of final SAT formula (vs LilotaneR), Δ*PAR2*: difference in PAR2-Score (vs LilotaneR), IRP: invalid rigid preconditions found, IFP: invalid fluent preconditions found, IFPPC: invalid preconditions found via postconditions, IOS: invalid operations found via invalid subtasks, IOPC: invalid operations found via postconditions

A.0.2. State of the Art Evaluation IPC-Scores

Domain	HyperTensioN	Lilotane(C)	LilotaneP+(C)	PandaGBFS	PandaSAT
Robot	**0.43**	0.35	0.35	0.37	0.36
Blocksworld-H	**0.73**	0.02	0.01	0.15	0.11
Monroe-PO	0.00	**0.78**	0.77	0.23	0.43
Logistics	0.25	0.28	0.26	0.33	**0.62**
Blocksworld-G	0.39	0.69	0.79	**0.86**	0.69
Satellite	0.00	0.69	0.59	0.68	**0.71**
Transport	0.36	0.75	0.74	0.58	**0.87**
Hiking	**0.83**	0.66	0.69	0.67	0.61
Woodworking	0.13	**0.81**	0.80	0.44	0.58
Barman	**0.90**	0.76	0.76	0.39	0.76
Entertainment	0.00	0.15	0.14	0.77	**0.93**
Childsnack	0.87	**0.91**	**0.91**	0.52	0.70
Snake	**1.00**	0.94	0.89	0.82	0.81
Factories	0.14	0.17	0.15	**0.32**	0.29
Elevator	0.63	0.69	0.71	0.57	**0.86**
AssemblyH	0.06	0.13	0.13	0.11	**0.14**
Depots	0.76	0.72	0.72	0.63	**0.83**
Monroe-FO	0.00	**0.84**	**0.84**	0.48	0.53
Freecell	0.00	0.05	0.05	0.00	**0.09**
Minecraft-P	**0.25**	0.09	0.08	0.05	0.08
Minecraft-R	**0.81**	0.41	0.48	0.52	0.47
Multiarm-B	0.06	0.03	0.02	**0.15**	0.13
Towers	0.45	0.39	0.37	**0.48**	0.30
Rover-GTOHP	**0.89**	0.54	0.56	0.42	0.61
Total	9.94	11.83	11.82	10.53	**12.51**

Table A.2.: IPC-Scores for all planners per domain.

A.0.3. Partition of Runtimes by Stage

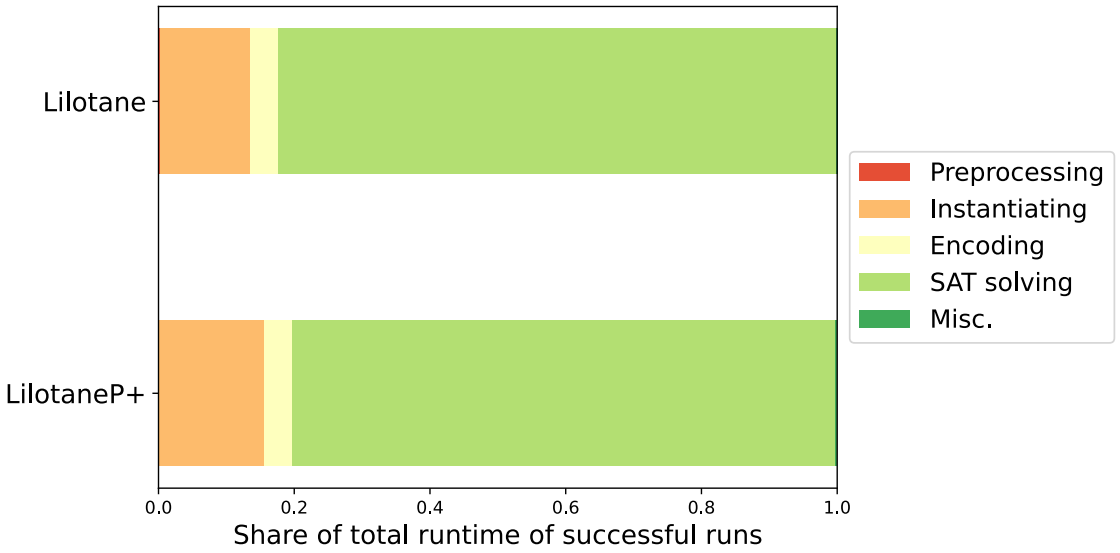

Figure A.1.: Partition of runtimes by stage