Master Thesis

Partially Instantiated Representations for Automated Planning

Marvin Williams

April 9, 2020

Supervisors: Prof. Dr. Peter Sanders
Dominik Schreiber
Tomáš Balyo

Institute of Theoretical Informatics, Algorithmics II
Department of Informatics
Karlsruhe Institute of Technology

Abstract

This thesis presents and evaluates techniques for classical planning with partially instantiated representations.

All successful state-of-the-art planners ground the problem at hand before they even start the actual planning. Therefore, sophisticated methods to efficiently generate compact ground representations have been proposed. In some cases, grounding still consumes a lot of resources (time as well as memory) and can even be the bottleneck of the planning procedure. We propose planning algorithms that circumvent this bottleneck and even handle planning tasks that other planners are not able to ground in the first place. They gradually ground the planning problem and solve encodings for partially instantiated representations of it using incremental SAT solvers. The experimental evaluation confirms that grounding is unnecessary or even counterproductive in some cases. Moreover, our planners are capable of solving planning problems from popular planning competitions that no state-of-the-art classical planner we tested is able to. However, due to the limited scope of this thesis, our work lacks some common optimizations that are necessary to be competitive with other SAT based planners across the broad variety of popular benchmark problem sets.

We also briefly look into parallelization of the grounding and planning routines. We achieve significant speedups for the grounding time on most complex tasks, while the imposed synchronisation affects the grounding of some tasks negatively. Solving encodings for different partially instantiated representations in parallel can be beneficial for a small number of parallel executions at the cost of increased memory consumption.

We conclude that our approach is promising and still has much potential for future work. Our planners might be useful in planner portfolios due to its unconventional approach.

Zusammenfassung

Diese Arbeit stellt Techniken für klassisches Planen mit partiell instantiierten Repräsentationen vor und bewertet diese anschließend empirisch.

Die erfolgreichsten aktuellen Planer müssen das gegebene Planungsproblem *grunden*, bevor sie mit dem eigentlichen Planungsprozess beginnen können. Daher wurde das Grunden in der Vergangenheit intensiv erforscht und Methoden entwickelt, um kompakte gegrundete Repräsentationen zu generieren. Das Grunden kann dennoch zu einem erheblichen Zeit- und Speicherverbrauch führen und zum Flaschenhals des gesamten Planungsprozesses werden. Wir stellen Techniken vor, um diesen Flaschenhals zu umgehen und sogar Probleme zu lösen, die andere Planer nicht einmal grunden können. Im Wesentlichen grunden wir das Problem schrittweise und stellen partiell instantiierte Repräsentationen des Problems als SAT-Formel dar, welche wir mittels eines SAT-Solvers versuchen zu lösen. Unsere Auswertungen zeigen, dass

das Grunden in einigen Fällen unnötig, in manchen sogar kontraproduktiv ist. Mit unserem Ansatz sind wir in der Lage, Planungsprobleme zu lösen, die keiner der anderen getesten Planer unter denselben Rahmenbedingungen lösen konnte.

Im Rahmen dieser Arbeit fehlt es der Implementierung dennoch an einigen bekannten Optimierungen, welche notwendig sind um auf den vielseitigen Problemtypen der Benchmarks gegen die Konkurrenz zu bestehen.

Wir befassen uns auch mit Möglichkeiten, unsere Ansätze für das schrittweise Grunden und Planen zu parallelisieren. Es stellt sich heraus, dass wir das Grunden insbesondere von komplexen Problemen beschleunigen können. Das nebenläufige Lösen von SAT-Formeln für verschiedene partiell instantiierte Repräsentationen des gleichen Problems beschleunigt zwar den Planer, erhöht jedoch den Speicherbedarf. Daher ist diese Art der Parallelisierung nur für wenige Threads geeignet.

Insgesamt hat sich der Ansatz als vielversprechend herausgestellt und bietet viele Möglichkeiten für weitere Forschung. Die vorgestellten Planer können dank der neuartigen Herangehensweise eine sinnvolle Ergänzung für Planer-Portfolios darstellen.

Acknowledgements

First and foremost I want to thank my supervisors Dominik Schreiber and Tomáš Balyo introducing me to the world of automated planning and SAT-solving in the first place. We had many insightful conversations and they guided my research with their knowledge and experience on these topics whenever needed. At the same time, they gave me the freedom to pursue my own research interests.

I would also like to thank Prof. Peter Sanders for the computing resources he provided to make the experimental evaluation of our research possible.

Last but not least, I would like to thank Nils Froleyks, a colleague and good friend of mine, who I accompanied to visit the Symposium on Combinatorial Search in 2019. His understanding of SAT solving and unconventional thinking in general led to many fruitful exchanges.

Eigenständigkeitserklärung

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, den 9. April 2020

Marvin Williams

Contents

1. Introduction

The study of intelligent agents is central in the field of Artificial Intelligence (AI). Those agents can be arbitrary devices that perceive their environment and can perform a set of predefined actions. Therefore, devising a valid sequence of actions to achieve a given goal is crucial to AI and has been subject of research since the early sixties. Historically, mostly domain-specific planners were used, which are intuitively more efficient compared to generic planning systems. We nowadays deploy autonomous systems using AI for a wide range of applications such as robot motion planning[LK05], aircraft assembly[Xu+12] and even mars rovers[Bre+05]. Given the high complexity and variety of tasks, efficient general-purpose planners are of high demand. While many different approaches towards those planners have been studied, describing planning problems as boolean satisfiability problem (SAT) and state-space searches have been the most popular.

As recent planning competitions[1] [2] suggest the dominance of state-space searches over satisfiability encodings for non-optimal planning, most research is nowadays focused on finding more sophisticated and advanced search strategies. Nevertheless, the individual performance of generic planners is highly domain dependant and planning as satisfiability still outperforms state-space searches on some domains.

The boolean satisfiability problem is one of the most famous and thoroughly researched NP-complete problems. Consequently, many highly performant general-purpose SAT-solvers have been developed. Naturally, SAT-based planning approaches benefit directly from the significant performance improvements over the past twenty years.

Nevertheless, typical encodings of complex planning tasks can quickly become too large to handle for SAT solvers. Especially when the plans to find are long, state-space searches have a natural advantage over SAT-based approaches. Many techniques for more compact encodings and efficient SAT solver scheduling have emerged over the time to tackle this problem.

1.1. Our Approach

In real world applications, planning tasks are described with predicates and action schemas, reminiscent of first-order logic. The majority of planners first compute a ground representation of the task by generating all possible actions instantiations. This method has proven to be very successful, as the ground representation allows

[1] `https://ipc2018-classical.bitbucket.io/#results`

[2] `http://ada.liacs.nl/events/sparkle-planning-19/results.html`

for faster, more precise heuristics and more aggressive pruning. Typically however, only a small fraction of all action instantiations is required to solve the planning problem. Thus, computing the ground representation often entails a lot unnecessary work. For the majority of problems this overhead is negligible, but can become a bottleneck for problems with too many complex operators.

In this thesis we incorporate SAT based planning in the grounding process. That way the SAT solver has the ability to instantiate actions it deems important. At the same time we don't neglect the advantages of a grounded representation to guide the SAT solver and prune the search space. We show several ways to combine SAT solving with grounding and discuss challenges and shortcomings arising with this approach. Also, we look into possibilities to parallelize our approach by concurrently grounding and encoding partially instantiated representations of the planning problem.

1.2. Structure of the Thesis

Chapter 1 introduces the general problem setting and motivates the approach taken in this thesis. In Chapter 2 we formalize definitions and notations used throughout this thesis. Additionally, we familiarize the reader with the trucking domain to illustrate some of the notations. Chapter 3 provides an overview over related work. We present notable publications relevant for or similar to our approach. Further, we refer the reader to literature to delve into related topics. Our contribution, SAT based planners using encodings of partially instantiated representations, is presented in Chapter 4. We then analyze and evaluate our planners in Chapter 5. Additionally, we compare configurations of our planners to other state-of-the-art planners. Chapter 6 concludes this thesis and provides suggestions for future work.

2. Notation and Preliminaries

In this chapter we introduce concepts and definitions used throughout this thesis. First, we give an intuition of the planning problem by providing a simple running example. Subsequently, we introduce a formal definition of the planning problem and briefly explain the Boolean satisfiability problem. We conclude this chapter with techniques and strategies for SAT-based planning.

2.1. Running Example: Trucking

The trucking domain describes logistic problems involving cities, trucks and parcels. Trucks can drive between cities that are connected by streets, load parcels and unload parcels. The goal is to deliver each parcel from their initial city to their destination city.

Figure 2.1 illustrates such a trucking problem. Here, the goal is to deliver the parcels located at cities A and B to city C, starting with the truck in city A. Assuming that the truck can only carry one parcel at a time, a possible solution to this problem would be to pick up the parcel, drive to city C, unload the parcel, drive to city B, pick up the parcel and again drive to city C and unload the parcel.

Some details, such as whether every parcel has the same destination, whether streets are one-way and how many parcels a truck can load depend on the exact formulation of the problem.

2.2. Planning

While many formalisms for automated planning exist, we will consider the most simple one, known as the *Classical Planning Problem* or *STRIPS-Planning*[1]. Our formalism roughly follows the notations of Ghallab et al. in *Automated Planning: Theory and Practice*[GNT04, pp. 20 sqq.].

Definition 1 (Planning Task)**.** A planning task is a 5-tuple $\Pi = (S, A, \gamma, s_0, S_g)$, where

- S is a finite set of possible *states* the world can be in,
- A is a finite set of *actions* that can be performed,

[1] STRIPS (Stanford Research Institute Problem Solver) is an early planner developed by Fikes and Nilsson in 1971[FN71].

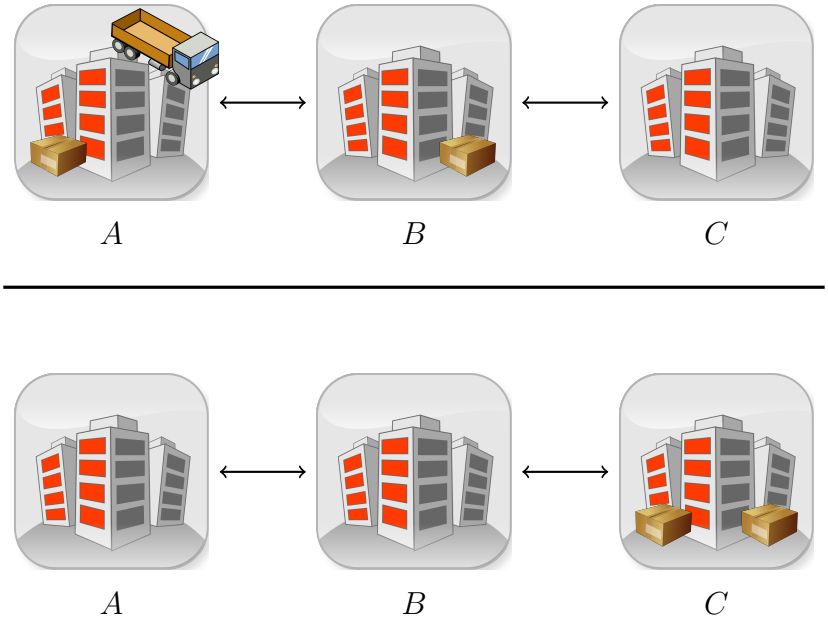


Figure 2.1.: A trucking problem with three cities, two parcels and one truck. Streets connect the city A with B and B with C (in both ways). The upper and lower configurations show the initial state and the goal, respectively. The truck's final location is irrelevant and thus omitted.

- $\gamma : S \times A \supseteq X \to S$ is the state transition function,
- $s_0 \in S$ is the *initial state*,
- $S_g \subseteq S$ is a set of *goal* states.

Conceptually, the world is in some initial state and actions can be applied to change the state. An action a is *applicable* in state s if γ is defined for (s, a). In classical planning, the world is

- *static*, i.e. the state does not change unless an action is performed,
- *deterministic*, i.e. actions have a fixed effect that is known a priori,
- *fully observable*, i.e. the current state of the world is always known,
- *timeless*, i.e. actions are performed instantaneously and their effect applies instantly.

Given a (possibly empty) sequence of actions $\tau = \langle a_1, \ldots, a_k \rangle$ with corresponding states $\langle s_0, \ldots, s_k \rangle$ such that $s_i = \gamma(s_{i-1}, a_i)$ for $1 \leq i \leq k$, we define

$$\gamma^*(s_0, \tau) \coloneqq \gamma(\cdots \gamma(\gamma(s_0, a_1), a_2) \cdots, a_k).$$

A state s is *reachable* (from s_0) if there is a sequence of actions τ such that $\gamma^*(s_0, \tau) = s$. Likewise, an action a is *reachable* (from s_0) if there is a sequence of actions τ with $a \in \tau$ and $\gamma^*(s_0, \tau)$ is well-defined.

Definition 2 (Plan)**.** A plan π of length k is a sequence of k actions such that $\gamma^*(s_0, \pi) \in S_g$.

Thus, a planning task has a plan if and only if at least one goal state is reachable. The *planning problem* is to find a plan for a given planning task. The *bounded planning problem* is to decide whether a plan of at most a given length exists and the *planning decision problem* asks whether a plan exists at all.

Example 1 (Trucking as Planning Task)**.** We formulate the trucking problem shown in Figure 2.1 as a planning task. In total we model 48 states, as each parcel can be at any of the three cities or loaded in the truck (L). Thus

$$S = \{(l_{P_1}, l_{P_2}, l_T) \in \{A, B, C, L\} \times \{A, B, C, L\} \times \{A, B, C\}\},$$

where l_{P_1}, l_{P_2} and l_T represent the locations of the parcels P_1, P_2 and the truck, respectively. Assuming that the truck can load one parcel at a time, the states $\{(L, L, l_T) \mid l_T \in \{A, B, C\}\}$ are unreachable.

The initial state and goal states are

$$s_0 = (A, B, A) \qquad \text{and} \qquad S_g = \{(C, C, A), (C, C, B), (C, C, C)\}.$$

Generally, the truck can perform 8 possible actions

$$A = \{\texttt{driveAtoB}, \texttt{driveBtoA}, \texttt{driveBtoC}, \texttt{driveCtoB}, \\ \texttt{loadP1}, \texttt{loadP2}, \texttt{unloadP1}, \texttt{unloadP2}\}.$$

The semantics of these actions are implicitly given by the state transition function, which is only defined in the following cases:

$$\gamma((l_{P_1}, l_{P_2}, l_T), a) = \begin{cases} (l_{P_1}, l_{P_2}, A) & \text{if } a = \texttt{driveBtoA} \wedge l_T = B, \\ (l_{P_1}, l_{P_2}, B) & \text{if } a = \texttt{driveXtoB} \wedge l_T = X \\ (l_{P_1}, l_{P_2}, C) & \text{if } a = \texttt{driveBtoC} \wedge l_T = B, \\ (L, l_{P_2}, l_T) & \text{if } a = \texttt{loadP1} \wedge l_{P_1} = l_T \wedge l_{P_2} \neq L, \\ (l_{P_1}, L, l_T) & \text{if } a = \texttt{loadP2} \wedge l_{P_2} = l_T \wedge l_{P_1} \neq L, \\ (l_T, l_{P_2}, l_T) & \text{if } a = \texttt{unloadP1} \wedge l_{P_1} = L, \\ (l_{P_1}, l_T, l_T) & \text{if } a = \texttt{unloadP2} \wedge l_{P_2} = L. \end{cases}$$

E.g., `loadP1` is only applicable if the truck is currently in the same city as P_1 and P_2 is not already loaded.

Finally, a possible plan of length 8 to solve this trucking problem is

$$\pi = \langle \texttt{loadP1}, \texttt{driveAtoB}, \texttt{driveBtoC}, \texttt{unloadP1}, \\ \texttt{driveCtoB}, \texttt{loadP2}, \texttt{driveBtoC}, \texttt{unloadP2} \rangle.$$

2.2.1. Lifted Representation

In practice, it is often infeasible to enumerate all possible states and actions of a planning task[2]. Instead, planning tasks are typically described with the *planning domain definition language* (PDDL). PDDL uses a compact and expressive *lifted representation* for planning tasks.

In a lifted representation, states are composed of first-order predicates and actions are generalized to operators. *Variables* (denoted in lowercase) in the predicates and operators take values from a finite domain, the *constants* (denoted in uppercase). A *term* is either a constant or a variable. Each *predicate* has a unique name, some valence and takes terms as arguments. If p is a predicate with valence n and $t_1, \ldots, t_n$ are terms then $p(t_1, \ldots, t_n)$ is called an *atom*. An atom is an *instance* of another atom if it can be obtained by replacing some of the variables of the other atom with constants. If $t_1, \ldots, t_n$ contain no variables, the atom is *ground*. Possible atoms for the trucking domain are

$$\text{street}(A, B),\ \text{truckAt}(l),\ \text{parcelAt}(p, A),\ \text{loaded}(P_2).$$

A *literal* is an atom with a polarity. It is either positive or negative, written as a or $\neg a$ for an atom a, respectively. Literals can be conjugated, that is $\overline{a} = \neg a$ and $\overline{\neg a} = a$. An operator is a 3-tuple $o = (\texttt{param}(o), \texttt{precond}(o), \texttt{effects}(o))$, where $\texttt{param}(o)$ is a list of variables, called *parameters*. All variables that appear in the operator must also appear in its parameters. $\texttt{precond}(o)$ and $\texttt{effects}(o)$ are sets of literals, called *preconditions* and *effects*, respectively. An operator from the trucking domain might look like this:

$\texttt{drive}(l_1, l_2)$
Preconditions: $\text{truckAt}(l_1)$, $\text{street}(l_1, l_2)$
Effects: $\neg\text{truckAt}(l_1)$, $\text{truckAt}(l_2)$

An assignment $\sigma = \{v_1 \mapsto c_1, \ldots, v_k \mapsto c_1\}$ maps distinct variables $v_1, \ldots, v_k$ to constants $c_1, \ldots, c_k$. An operator o can be *instantiated* with an assignment σ by replacing v in the preconditions and effects with c and removing v from the parameters for each $v \mapsto c \in \sigma$. The resulting operator is denoted as $\sigma(o)$ and called an *instance* of o. An operator is *ground* if its parameters are empty and thus every literal in the preconditions and effects is ground. Operator instances that are not ground are called *partially instantiated*. To give some examples,

(1) $\texttt{drive}(l_2)$
Preconditions: $\text{truckAt}(B)$, $\text{street}(B, l_2)$
Effects: $\neg\text{truckAt}(B)$, $\text{truckAt}(l_2)$

(2) $\texttt{drive}()$
Preconditions: $\text{truckAt}(A)$, $\text{street}(A, B)$
Effects: $\neg\text{truckAt}(A)$, $\text{truckAt}(B)$

[2] A trucking problem with 15 cities, 5 trucks and 10 parcels already has $\sim 10^{30}$ states.

are instances of the operator $\texttt{drive}(l_1, l_2)$, with $\sigma = \{l_1 \mapsto B\}$ for (1) and $\sigma = \{l_1 \mapsto A, l_2 \mapsto B\}$ for (2). (1) and (2) can be denoted as $\texttt{drive}(l_1, A)$ and $\texttt{drive}(A, B)$, respectively. $\texttt{drive}(A, B)$ is a ground instance, since it has no parameters left.

Definition 3 (Lifted Representation)**.** A planning task in lifted representation is given by $\Pi_L = (C, O, s_0, g)$ with

- the finite domain, a set of constants C,
- a finite set of operators O,
- a set of ground atoms, the initial state s_0,
- a set of ground literals, the goal g.

The set of predicates appearing in the description of O, s_0 and g is denoted as $P(\Pi_L)$. The finite set of possible ground atoms over C of all predicates in $P(\Pi_L)$ is denoted as $\overline{P}(\Pi_L)$. Finally, $\overline{O}(\Pi_L)$ is the set of all ground operators instantiatable from operators in O. We omit Π_L and write P, $\overline{P}$ and $\overline{O}$ if the representation is clear from the context. The states of the planning task then are subsets of $\overline{P}$. A ground atom *holds* in a state if and only if it is contained in the state (*closed world assumption*). For any set of literals L we define L^+ to be all atoms that are positive in L and likewise L^- to be all atoms that are negated in L. L is *consistent*, if and only if it does not both contain both a and $\neg a$ for any atom a. A state s *satisfies* L if and only if

$$L^+ \subseteq s \qquad \text{and} \qquad L^- \cap s = \emptyset.$$

Specifically, a state can satisfy the goal g and preconditions of ground operators. A ground operator is *applicable* in a state if its preconditions are satisfied by the state. Applying a ground operator in a state removes all negative effects from the current state and adds all positive effects. We therefore assume that the effects of all operators are consistent[3]. Formally, the planning task $\Pi = (S, A, \gamma, s_0, S_g)$ described by Π_L has the following properties.

- $S \subseteq 2^{\overline{P}}$
- $A = \overline{O}$
- For all $(s, a) \in S \times A$ with a applicable in s,

  $$\gamma(s, a) = (s \setminus \texttt{effects}(a)^-) \cup \texttt{effects}(a)^+$$

  and γ undefined for all other $(s, a) \in S \times A$.
- $s_0 \in S$

[3]PDDL does not have this restriction. Here, negative effects are removed from the state before positive effects are added.

- $S_g = \{s \in S \mid s \text{ satisfies } g\}$

A predicate is *rigid* if it does not appear in the effects of any operator in O. Ground atoms of rigid predicates are therefore always true or always false in every reachable state. Consequently, a planning problem is trivially unsolvable if the subset of literals of rigid predicates in g is not satisfied in s_0.

Example 2 (Lifted Representation for Trucking)**.** We show a lifted representation of the planning task depicted in Figure 2.1. The constants in this representation are $C = \{P_1, P_2, A, B, C\}$, analogous to Example 1. The operators in schematic form are

`drive`(l_1, l_2)
Preconditions: truckAt(l_1), street(l_1, l_2)
Effects: $\neg$truckAt(l_1), truckAt(l_2)

`load`(l, p)
Preconditions: truckAt(l), parcelAt(p, l), $\neg$loaded(P_1),$\neg$loaded(P_2)
Effects: $\neg$parcelAt(l), loaded(p)

`unload`(l, p)
Preconditions: truckAt(l), loaded(p)
Effects: parcelAt(p, l), $\neg$loaded(p)

with the initial state and goal

$$\begin{aligned} s_0 = \{&\text{truckAt}(A),\ \text{parcelAt}(P_1, A),\ \text{parcelAt}(P_2, B),\\ &\text{street}(A, B),\ \text{street}(B, A),\ \text{street}(B, C),\ \text{street}(B, C)\}\\ g = \{&\text{parcelAt}(P_1, C),\ \text{parcelAt}(P_2, C)\}. \end{aligned}$$

2.2.2. Grounding

A lifted representation is an *instance* of another lifted representation if all its operators are instances from operators of the other representation while the constants, initial state and goal stay the same. The operators of an instance of a lifted representation must be such that at least one plan for the initial planning task must remain findable. In particular, instances of representations may contain no operators if there was no plan for the corresponding planning task. A representation is *ground* (also called a ground representation) if all operators are ground. Thus, it contains no variables and atoms decay to Boolean propositions. *Grounding* describes the process of generating an instance of a lifted representation that is ground. A representation instance is called *partially instantiated* if it is not ground.

Generally, the number of possible ground instances of an operator is exponential in its valence[4]. Thus, enumerating all ground operators can be infeasible in practice if the maximum valence or the number of constants is large. However, oftentimes

[4] An n-ary operator has $n^{|C|}$ ground instances.

only a small fraction of all ground instances is needed to solve a given problem. Therefore, partially instantiated representations do not necessarily preserve all possible ground instances but can omit operators that are unreachable or otherwise not useful. Pruning operators during grounding can oftentimes mitigate the enumeration problem and potentially speed up the actual planning due to the smaller search space. Many grounding strategies have been proposed (see Section 3.1).

2.2.3. Complexity

The theoretical complexity of the planning decision problem depends on the used representation. With an explicit representation (i.e. with lists of states and actions) of the planning task, the problem is in P, as it can be reduced to a reachability test in a directed graph. The vertices and edges of this graph indicate states and possible state transitions, respectively. A plan exists if and only if any vertex representing a goal is reachable from the vertex representing the initial state. Both the graph construction and the reachability test are polynomial.

However, the planning decision problem is most commonly studied with a ground representation and referred to as PlanSat. While actions in a ground representation correspond to actions in the explicit representation, the set of possible states is given implicitly with the set of (ground) atoms. Many complexity results about PlanSat and variants thereof have been presented by Bylander[Byl91], most notably that PlanSat is PSPACE-complete. The outline of the proof is as follows. We show that PlanSat is in NPSPACE and PSPACE-hard. Since NPSPACE = PSPACE (see [Sav70]), this proves PSPACE-completeness. Let n be the number of atoms that appear in the ground representation of a planning task. Then, the number of states of the task is at most 2^n since every atom is ground. As the shortest plan (if one exists) revisits no state, it has at most 2^n actions.

- PlanSat is in PSPACE, since a nondeterministic Turing machine can decide PlanSat with polynomial space complexity by iteratively selecting an action and applying it until either a goal state is reached (accept) or 2^n actions have been applied (reject). The current state can be stored with n cells, an action can be selected and applied in polynomial space and the action counter requires n cells to count up to 2^n.

- PlanSat is PSPACE-hard, since each problem decidable by a Turing machine with polynomial space complexity can be reduced to PlanSat in polynomial time by simulating the corresponding Turing machine (w.l.o.g., assume a binary alphabet). The relevant cells of the tape, possible head positions and internal states of the Turing machine are represented by 0-ary predicates. Each combination of head position and transition is represented by an operator. Preconditions ensure that the operator is only applicable if the current head position, tape content and internal state are as intended. The effects model the internal state transition, tape modification and head movement. The number

and size of the operators is polynomial in the size of the Turing machine and since all predicates are 0-ary, all operators are ground.

The planning problem with the planning task given in lifted representation is EXPSPACE-complete (see [GNT04, pp. 61 sqq.]). Intuitively, this is because the number of possible states is doubly exponential in the number of atoms. The proof is similar to the proof for the PSPACE-completeness of PlanSat but requires additional work to represent the tape of the Turing machine.

2.3. Boolean Satisfiability Problem

A *Boolean formula* is an formal expression with Boolean variables, conjunctions ($\wedge$), disjunctions ($\vee$) and negations ($\neg$). A Boolean variable can take values from $\{\texttt{true}, \texttt{false}\}$. A literal is either a Boolean variable or the negation of a Boolean variable. An assignment of values from `true` and `false` to each variable is a *model* for a Boolean formula if the formula evaluates to `true` under that assignment. A formula is *satisfiable* if there is a model for it.

The Boolean satisfiability problem (Sat) is to determine whether a given Boolean formula is satisfiable. Sat instances are typically given in *conjunctive normal form* (CNF). That is, the formula is a conjunction of clauses. A *clause* is a disjunction of literals. A *unit clause* is a clause containing exactly one literal. A Boolean formula in CNF is satisfiable if there is a model that satisfies every clause. For example, the formula $(a \vee \neg b \vee c) \wedge (\neg a \vee b)$ with variables $\{a, b, c\}$ is in CNF, the clauses are $(a \vee \neg b \vee c)$ and $(\neg a \vee b)$. The formula is satisfiable, a possible model is $\{a \mapsto \texttt{false}, b \mapsto \texttt{false}, c \mapsto \texttt{true}\}$.

Sat was the first problem proven to be NP-complete (see [Coo71]) and is still actively researched. Sophisticated heuristics and solving strategies enable modern Sat algorithms to solve formulae with millions of clauses, making them feasible for many practical combinatorial problems. In practise, it is commonly required for SAT-solvers to provide a model in case the given formula is satisfiable.

2.4. SAT-based Planning

Encoding planning tasks as Sat instances was first proposed by Kautz, Selman, et al. in 1992[KS+92]. It can be assumed that constructing a Boolean formula that is satisfiable if and only if a corresponding PlanSat instance is `true` is inherently exponential, since PlanSat is PSPACE-hard and Sat is in NP. Therefore, generating and solving such a formula can quickly exceed time and memory limits. To mitigate this problem, the planning task can be encoded incrementally: Given a fixed horizon h, we construct Boolean formulae F_h that are true if and only if a plan within the horizon h exists. Intuitively, F_h can be organized in h *steps*, where each step contains a state and actions to be applied to constitute the state of the next step. A SAT solver successively tries to solve F_h with increasing h until it finds a satisfiable

formula. This method is especially efficient if a plan exists that is significantly shorter than theoretically possible (see Section 2.2.3). We usually want F_h to be *constructive*, that is one can easily obtain a plan from a model for F_h.

2.4.1. Step Semantics

If an encoding allows only one action to be applied in each step, it uses *sequential* step semantics. The horizon (*makespan*) then directly corresponds to the maximum length of a plan that can be found. This restriction can be alleviated to allow multiple actions to be performed in one step, yielding *parallel* step semantics. Parallel step semantics have two major advantages over sequential step semantics[RHN06]. First, plans can be found within a smaller horizon, so fewer formulae have to be solved and the encodings stay smaller. Second, total orderings of actions in one step are not encoded and thus considered by the SAT-solver, so fewer possible action orderings have to be checked. For parallel step semantics, the concept of *parallel plans* is useful.

Definition 4 (Parallel Plan)**.** A parallel plan is a sequence of sets of actions $S = \langle A_1, \ldots, A_k \rangle$, such that there is a sequence of states $\langle s_0, \ldots, s_k \rangle$ with

1. $s_i = \gamma^*(s_{i-1}, \tau_i)$ for some ordering τ_i of A_i for $1 \leq i \leq k$ and
2. the concatenation of all τ_i is a plan.

The parallel plan S has a makespan of k.

A model for an encoding with parallel step semantics corresponds to a *parallel plan*, where for each step there is a set of actions. An encoding has $\forall$-*step* semantics if every ordering within each action set of the parallel plan obtained by a model yields the same successor state. For that to hold, all actions have to be applicable in the state and no two actions must *interfere* in each step. Two ground operators o_1 and o_2 interfere if either

(a) $\exists l \in \texttt{precond}(o_1) : \bar{l} \in \texttt{precond}(o_2)$,

(b) $\exists l \in \texttt{effects}(o_1) : \bar{l} \in \texttt{effects}(o_2)$,

(c) $\exists l \in \texttt{effects}(o_1) : \bar{l} \in \texttt{precond}(o_2)$,

(d) $\exists l \in \texttt{precond}(o_1) : \bar{l} \in \texttt{effects}(o_2)$.

In case (c) o_1 *disables* o_2 and in case (d) o_2 disables o_1.

Example 3 (Parallel Plans for Trucking)**.** For the sake of this example we assume that the truck can load multiple parcels at once.

A valid parallel plan with makespan 3 for the example in Figure 2.1 then would be

$$\begin{aligned} S = \langle &\{\text{load}(A, P_1), \text{drive}(A, B)\}, \\ &\{\text{load}(B, P_2), \text{drive}(B, C)\}, \\ &\{\text{unload}(C, P_1), \text{unload}(C, P_2)\}\rangle. \end{aligned}$$

However, S does not adhere to $\forall$-step semantics, as $\text{load}(A, P_1)$ is not applicable after $\text{drive}(A, B)$. The shortest parallel plan (in terms of the makespan) with $\forall$-step semantics is

$$\begin{aligned} S = \langle &\{\text{load}(A, P_1)\}, \\ &\{\text{drive}(A, B)\}, \\ &\{\text{load}(B, P_2)\}, \\ &\{\text{drive}(B, C)\}, \\ &\{\text{unload}(C, P_1), \text{unload}(C, P_2)\}\rangle \end{aligned}$$

with makespan 5.

Rintanen et al.[RHN06] proposed $\exists$-*step* semantics, which are more relaxed. An encoding has $\exists$-step semantics if the actions in each step can be applied in *some* order to get to the state of the next step. All actions have to be initially applicable and the union of all effects has to be consistent[5]. The first parallel plan in Example 3 adheres to $\exists$-step semantics.

Other step semantics such as relaxed $\exists$-step semantics and R^2-exists-step semantics are discussed in Section 3.2.

2.4.2. Base Encoding

Most SAT encodings for planning found in the literature assume a ground representation. These representations are unsuited for our use case. We introduce a base encoding for lifted representations of planning tasks. The formulae of this encoding are shared by all subsequent encodings. The base encoding allows for multiple actions to be applied in one step, provided they are applicable and the union of their effects is consistent. Also, at most one instance of each operator can be applied in each step. However, no further restrictions on the operators are imposed. Therefore, the encoding itself is not incomplete and further restrictions are required for a proper encoding. The following naming conventions are taken from [BS] and adjusted accordingly for lifted representations.

Let $\Pi_L = (C, O, s_0, g)$ be the planning task to encode and h the horizon. We construct F_h such that we can extract the applied actions from a model of F_h efficiently. Let L be the set of all ground literals of $\overline{P}$. For each ground atom $a \in A$

[5]Rintanen et al. proposed other, less restricted $\exists$-step semantics, which they deemed impractical.

we have the Boolean variables is_a^t for $t \in \{0, \ldots, h\}$. They indicate whether a holds in step t (and after the last step). The variables do_o^t for $t \in \{0, \ldots, h-1\}$ express whether an instance of operator o is applied in step $t \in \{0, \ldots, h-1\}$. Finally we have the Boolean variable $m_{o,v\mapsto c}^t$ for each operator o, $v \in \texttt{param}(o)$ and $c \in C$. It indicates whether c is assigned to v in the instance of o that is applied in step $t \in \{0, \ldots, h-1\}$.

The *precondition support* PS_l for a ground literal l is the set of all pairs of operators and inclusion-wise minimal assignments (o, σ) such that l is contained in the preconditions of $\sigma(o)$. Some precondition supports induced by the operators from Example 2 are:

$$\begin{aligned}
PS_{\text{street}(B,C)} &= \{(\texttt{drive}(l_1, l_2), \{l_1 \mapsto B, l_2 \mapsto C\})\} \\
PS_{\neg\text{street}(A,B)} &= \{\} \\
PS_{\text{truckAt}(A)} &= \{(\texttt{drive}(l_1, l_2), \{l_1 \mapsto A\}), (\texttt{load}(l, p), \{l \mapsto A\}), \\
&\qquad (\texttt{unload}(l, p), \{l \mapsto A\})\} \\
PS_{\neg\text{loaded}(P_1)} &= \{(\texttt{load}(l_1, l_2), \{\})\}
\end{aligned}$$

The *effect support* ES_l is defined analogously as the set of all pairs of operators and inclusion-wise minimal assignments (o, σ) such that l is contained in the effects of $\sigma(o)$. Given an operator o and an assignment σ, we define

$$Do_{o,\sigma}^t \coloneqq do_o^t \wedge \bigwedge_{v\mapsto c \in \sigma} m_{o,v\mapsto c}^t, \tag{2.1}$$

which expresses that an instance of $\sigma(o)$ is applied in step t.

F_h is the conjunction of the formulae given below. Each of them constrains the Boolean variables in order to establish their semantics. We do not necessarily give the formulae in CNF and use implications ($\rightarrow$), thus the formulae have to be converted to CNF using trivial transformations in order to be valid input for SAT solvers.

1. The initial state holds in step 0:

$$\bigwedge_{a \in A} \begin{cases} is_a^0 & \text{if } a \in s_0, \\ \neg is_a^0 & \text{otherwise.} \end{cases} \tag{2.2}$$

2. Ensuring that the goal literals are satisfied after the last step:

$$\bigwedge_{l \in g} \begin{cases} is_l^h & \text{if } l \text{ is positive,} \\ \neg is_l^h & \text{if } l \text{ is negative.} \end{cases} \tag{2.3}$$

The following formulae are instantiated for each step $t \in \{0, \ldots, h-1\}$.

3. At most one constant can be assigned to each parameter of any operator o:

$$\forall v \in \texttt{param}(o) : \bigwedge_{\substack{c_1, c_2 \in C \\ c_1 \neq c_2}} \neg m_{o,v\mapsto c_1}^t \vee \neg m_{o,v\mapsto c_2}^t \tag{2.4}$$

4. Each operator o needs to be ground in order to be applied:

$$\forall v \in \texttt{param}(o) : do_o^t \to \bigvee_{c \in C} m_{o,v \mapsto c}^t \tag{2.5}$$

5. If an operator is applied, its preconditions need to be satisfied:

$$\forall l \in L, (o,\sigma) \in PS_l : Do_{o,\sigma}^t \to \begin{cases} is_l^t & \text{if } l \text{ is positive,} \\ \neg is_{\bar{l}}^t & \text{if } l \text{ is negative.} \end{cases} \tag{2.6}$$

6. Likewise, the effects change the state of the next step:

$$\forall l \in L, (o,\sigma) \in ES_l : Do_{o,\sigma}^t \to \begin{cases} is_l^{t+1} & \text{if } l \text{ is positive,} \\ \neg is_{\bar{l}}^{t+1} & \text{if } l \text{ is negative.} \end{cases} \tag{2.7}$$

7. *Frame axioms.* Variables representing the same atom in consecutive steps only differ if an action is applied to support the change from `false` to `true`

$$\forall a \in A : (\neg is_a^t \wedge is_a^{t+1}) \to \bigvee_{(o,\sigma) \in ES_a} Do_{o,\sigma}^t \tag{2.8}$$

or from `true` to `false`:

$$\forall a \in A : (is_a^t \wedge \neg is_a^{t+1}) \to \bigvee_{(o,\sigma) \in ES_{\neg a}} Do_{o,\sigma}^t. \tag{2.9}$$

2.4.3. Step scheduling

Trying to solve every formula $F_0, F_1, \ldots$ until a solvable formula is found additionally proves that no plan within a smaller horizon exists. This method therefore yields a shortest plan if sequential step semantics are used. In general however, the horizon does not carry any practical meaning. Since proving unsolvability for the last unsolvable formula is often much more difficult to than it is to find a model for the first solvable formula ([RHN06]), we can employ more efficient step scheduling strategies. On the one hand, one wants to skip all hard unsolvable formulae and find an easy solvable formula. On the other hand, skipping solvable formulae can make the formula unnecessarily large. Therefore, heuristics have to be employed to predict the easiest solvable formula. A common step scheduling strategy introduced by Rintanen[Rin14] to increase the horizon by a constant factor $\gamma > 1$ and solve the formulae $F_{\lceil \gamma^0 \rceil}, F_{\lceil \gamma^1 \rceil}, \ldots$ successively.

2.4.4. Incremental SAT Solving

We expect to prove unsatisfiability for multiple formulae in succession. If these formulae are very similar in that they mostly build up on each other, we speak of

incremental SAT solving. Incremental SAT solving is a common pattern for many applications, among them is planning. Thus, SAT solvers have been developed for this specific use case. They maintain a growing set of clauses to satisfy and a list of *assumptions*. Assumptions are literals that are added as unit clauses only for the next solving attempt and removed afterwards. The solver "carries over" knowledge from previous solving attempts.

3. Related Work

The literature about automated planning in general is very broad. In this chapter we give an overview of the work that covers similar topics to this thesis and discuss their relevance to our approach.

3.1. Grounding

Almost all state-of-the-art planners ground the PDDL planning task in an early stage prior to the actual planning. The Fast Forward planning system by Hoffmann and Nebel[HN01] grounds operators step by step and prunes them as soon as their rigid preconditions become unsatisfiable. Once all operators are ground, it filters out unreachable operators utilizing a planning graph (Blum and Furst introduced planning graphs for their novel Graphplan[BF95] planner). Edelkamp and Helmert proposed an efficient method for generating a ground finite domain representation [EH99] for planning tasks. In contrast to propositional representations, variables in finite domain representations can take a finite number of values instead of only `true` and `false`. Finite domain representations are generally more expressive and are often more concise. Their original method was only applicable to a small subset of PDDL features but was later extended and refined by Helmert[Hel09]. The grounding procedure itself is based on delete relaxation, a concept introduced by Hoffmann and Nebel[HN01] and converts the planning task into a *datalog program* for the actual grounding. Datalog is a logic programming language, which allows the generation of the reachable ground operators without first generating all possible ground operators. Their grounding procedure performs very well in practice, thus many planners from the latest International Planning Competition[1] (IPC) and Sparkle Planning Challenge[2] rely on it (e.g., [Kat+18][HRK11][FBS19]). The advantages of finite domain representations are evident to us, yet for our foundational research we rely on Boolean atoms throughout this thesis.

Above grounding strategies are most effective when a ground representation is desired. Especially the reachability analysis is only tangible with ground operators. Less research has been published regarding partial grounding and planning with lifted representations in general. Ridder and Fox presented a technique to derive a partially grounded representation from a lifted relaxed planning graph[RF14]. In particular, they showed how to handle the complexity blowup entailed with generating the relaxed planning graph in a lifted setting. In a recent publication,

[1] `https://ipc2018-classical.bitbucket.io/`

[2] `http://ada.liacs.nl/events/sparkle-planning-19`

Gnad et al. described how to employ machine learning for grounding[Gna+19]. Their algorithm predicts which operators are relevant for the planning process und thus should be grounded next. At some point, they try to solve the planning problem with the already grounded operators. This might fail if the ground operators are insufficient to find a plan. In that case, they continue the grounding process and retry. The main difference to our approach is that their representations are incomplete and are extended when needed. Also, the actual planning is exclusively done with ground operators, while we allow partially instantiated operators in the representation to encode.

3.2. SAT-based Planning

As already mentioned in Section 2.4, Kautz, Selman, et al. were the first to represent planning problems as SAT formulae. They initially proposed an encoding based on a ground representation using sequential step semantics, but later ([KMS96]) added encodings that supported ∀-step semantics and lifted representations. Using SAT solvers for planning has since then been a popular approach, given the fast advances in SAT solving techniques (e.g., CDCL[3]). Consequently, many improvements to the original encodings as well as entirely novel approaches have been proposed. Besides the ability to find shortest plans, SAT-based planning was popular to find shortest parallel plans with ∀-step semantics. Rintanen et al.[RHN06] argued that the makespan of parallel plans lacks practical importance and proposed ∃-step semantics (see Section 2.4.1), based on ideas from Dimopoulos et al.[DNK97]. ∃-step semantics allow more actions to be applied in one step, given they could be applied one after the other. They presented multiple encodings for ∃-step semantics with various restrictions, but all required the actions of each step to be initially applicable. *Relaxed ∃-step* semantics, introduced by Wehrle and Rintanen[WR07], expand this idea by dropping this requirement. Still, effects had to be consistent in each step. Balyo lifted this restriction by proposing a further relaxation [Bal13]: With R^2-*exists-step* semantics, effects are allowed to disable effects of previous actions in the same step.

Step scheduling is an important factor for the efficiency of SAT-based planners if optimal makespans are not required. Many makespan scheduling strategies, such as parallel solving and preemption, are discussed in [RHN06].

An entire different approach to planning as SAT was also presented by Kautz and Selman with the planner Blackbox[KS98]. This planner generates a planning graph for a fixed number of layers and encodes that graph as a SAT formula. If this formula is unsolvable, the planning graph needs to be extended to generate a solvable formula.

Although SAT based planning has recently lost popularity in favor of state-space searches, it is still competitive in many areas. The Madagascar planner family[Rin14] by Rintanen is a set of highly optimized SAT based planners with a handcrafted

[3]Conflict driven clause learning (CDCL) is based on DPLL and was first proposed in [MS96]

SAT solving backend. It ranked highly in the IPC 2014[4] and is often used in planner portfolios[Rina].

3.2.1. Lifted encodings

Kautz et al. proposed the first encodings for lifted representations, but they were limited to sequential step semantics[KMS96]. Transferring their encodings with parallel step semantics to a lifted representation allows for only one instantiation per operator in each step. Robinson et al. were the first to propose a lifted encoding that was able to find makespan-optimal parallel plans with ∀-step semantics[Rob+09]. They achieve this by selecting a set of preconditions and effects to be applied in each step. Each operator whose preconditions and effects are selected is then considered as applied. Selecting consistent sets of preconditions and effects however entails the complexity of grounding the operators and requires a much more complex encoding.

3.2.2. Parallel Planning

There are a few different approaches towards parallel SAT based planning. One is to solve multiple formulae simultaneously. Rintanen et al. describes in [RHN06] how multiple horizons can be tested in parallel. The hope is that not all processes try to solve hard, unsatisfiable formulae. The algorithm finishes as soon as one satisfiable formula is solved. Since generating a model for a solvable formulae generally requires much less time than proving that a formula is unsolvable, this approach can theoretically scale arbitrarily. Another approach is to use a portfolio of different encodings and/or SAT solvers. This approach does not scale very well but is applied successfully for hard combinatorial problems (e.g. [BSS15]). Finally, one can trivially parallelize SAT based planning by using parallel SAT solvers such as Glucose-Syrup[AS14] or Plingeling[Bie13].

[4] `https://helios.hud.ac.uk/scommv/IPC-14`

4. Planning with Partially Instantiated Representations

In this chapter we present our approaches to utilize partially instantiated representations for the planning process in detail. At first, we outline the scope of our approaches and introduce our algorithms. We then describe its key components and justify critical design decisions.

4.1. Outline

The main idea is to ground, encode and solve the planning problem in an interleaving or parallel fashion. We therefore eliminate the need to ground the planning task beforehand and can take advantage of encodings for lifted representations. The grounder maintains a partially instantiated representation of the initial planning task. This representation can then be repeatedly refined. These intermediate representations can serve multiple purposes. Firstly, we can add constraints about the pruned operators to an existing encoding in order to guide the SAT solver. Secondly, we can re-encode these representations as they might yield more compact formulae for the SAT-solver.

We further present a parallel variant of our planner. This planner concurrently grounds the planning task and tries to solve formulae for intermediate representations. Also, we also parallelize the grounding process itself.

4.2. Grounding

In this chapter we describe the grounding algorithm in detail and present the options we explored for its subroutines. The key feature of our grounder is the stepwise grounding of the initial lifted representation. More precisely, given a target *groundness*, the grounder refines the representation until this groundness is reached. The grounding can then be resumed at a later time aiming for a higher groundness. Operators are pruned while grounding to reduce the size of the representations. We give a tractable definition of prunablility and present different methods to identify prunable operators.

As the grounding routine stops when a given groundness is reached, we first define the metric to measure the groundness.

Definition 5 (Groundness)**.** Given a lifted representation Π_L, the *groundness* r_{Π_L} of representation Π'_L instantiated from Π_L is

$$r_{\Pi_L}(\Pi'_L) := \frac{|O'| + |\overline{O}(\Pi_L) \setminus \overline{O}(\Pi'_L)|}{|\overline{O}(\Pi_L)|}, \tag{4.1}$$

where O' are the operators of Π'_L. Intuitively, r_{Π_L} expresses the fraction of all operators that are present in Π'_L or pruned from Π_L to all possible ground operators. Thus, $r_{\Pi_L}(\cdot) \in [0, 1]$. We write r for r_{Π_L} if Π_L is clear from context.

The higher r, the more operators are instantiated or pruned. It is $r(\Pi'_L) = 1$ if and only if all operators in Π'_L are ground[1]. The grounding procedure iteratively *refines* and prunes operators until the desired groundness is reached. Refining an operator is to select a subset of the parameters and replace the operator with all its instances that have constants assigned to exactly these parameters. These instances are *refined* operators of the original operator. Algorithm 1 shows this procedure. It takes a lifted representation $\Pi_L = (C, O, s_0, g)$ of a planning task and the target groundness r_t as input and outputs a partially instantiated representation Π'_L with $r(\Pi'_L) >= r_t$. Since the intermediate groundness (line 2) increases monotonically and converges towards 1 with refining and pruning, the algorithm always terminates. In the

Algorithm 1: Iterative Grounding

Data: Lifted representation $\Pi_L = (C, O, s_0, g)$ and groundness $r_t \leq 1$
Result: Partially instantiated representation Π'_L such that $r(\Pi'_L) \geq r_t$

```
1  O' ← O;
2  while r((C, O', s_0, g)) < r_t do
3  |   o ← selectOperator(O');
4  |   V ← selectParameters(o);
5  |   O' ← O' \ {o};
6  |   forall assignments σ with variables from V do
7  |   |   if prune(σ(o)) = false then
8  |   |   |   O' ← O' ∪ {σ(o)};
9  |   |   end
10 |   end
11 end
12 filter(O');
13 Π'_L ← (C, O', s_0, g);
14 return Π'_L
```

following, we discuss the main parts of the algorithm. The method `selectOperator` in line 3 selects the next operator to refine and `selectParameters` (line 4) chooses

[1]This is not true for the pathological case where $|C| = 1$, since each operator then only has one instantiation.

the parameters to refine with. The instances of o are added back to O' unless they can be pruned (line 7). The filtering (line 12) again removes prunable operators from the final representation. Although the partially instantiated representations obtained with $r_t < 1$ might differ depending on the operator and parameter selection, the ground representation obtained with $r_t = 1$ is the same regardless of these selections.

4.2.1. Pruning

Oftentimes, a large portion of the operators is not needed to find a plan. Pruning operators not only reduces the encoding size and therefore planning time, but also can make grounding itself feasible in the first place. On the one hand, we want to prune as many operators as possible, on the other hand we must not prune operators that might be required to find a plan. An operator can be pruned if no instance of it is reachable or has useful effects to reach the goal. As it is PSPACE-hard to decide whether an operator is reachable (consider the preconditions of a ground operator as goal), we relax these constraints and only identify a subset of all operators that can safely be pruned. We extend the definition of rigidness to ground atoms and introduce the concept of uselessness.

Given a lifted representation of a planning task, a ground atom a is *rigid true* if it holds in the initial state and no operator can be instantiated such that it has $\neg a$ as effect. Similarly, it is *rigid false* if it does not hold in the initial state and no operator can be instantiated such that it has a as effect. Thus, all ground atoms of rigid predicates are either rigid true or rigid false. Rigid atoms either hold in every reachable state or in no reachable state.

A ground atom a is *useless* if neither a nor $\neg a$ are a goal and no operator can be instantiated such it has a or $\neg a$ as precondition. Whether or not a useless atom holds is irrelevant for the goal and the applicability of any operator.

We call an operator *prunable* if all its ground instances

a) have at least one precondition that requires a rigid true atom to not hold, or

b) have at least one precondition that requires a rigid false atom to hold, or

c) have all effects changing useless atoms or uphold rigid atoms.

We check only if one of the cases applies to all ground instances of an operator so that we do not need to generate all ground instances but can instead test the ground literals of each precondition and effect independently. However, this method does not detect all prunable operators as early as possible. Consider the following operator:

`do`(x)
Preconditions: precond(x)
Effects: eff(x)

Given that the constants are $\{A, B\}$ and only precond(A) is rigid false and only eff(B) is useless, the operator can be pruned because either case b) or case c) applies for each ground instance but neither case is applicable to all ground instances. Nevertheless, prunable ground operators are always detected, thus the instances of prunable operators are pruned eventually.

Since it might be computationally expensive to generate all ground literals for each precondition and effect, we present three pruning policies.

Eager generates all ground literals of all preconditions and effects to check if they render the operator prunable. This policy detects prunable operators earlier than the other policies but is the most computationally expensive.

Ground only checks preconditions and effects that are already ground. This policy does not necessarily detect all prunable operators found by the **Eager** policy unless the operator is ground. In particular, case c) can only be considered if all effects are ground. Thus, detecting a prunable operator might be delayed until it is sufficiently refined.

Rigid is a relaxed version of the **Ground** policy. It only considers ground preconditions of rigid predicates and ground effects. As ground atoms might be rigid true or rigid false even though the corresponding predicate is not rigid, this policy does not detect all prunable operators. Nevertheless, it is the least expensive policy to compute.

Pruning operators might render other operators prunable. Therefore, O' might still contain prunable operators after the loop finished. The filtering in line 12 iteratively removes all prunable operators from O'.

4.2.2. Operator Priority

We use a first-in-first-out (FIFO) queue to determine which operator is selected to be refined next. After refining an operator, the refined operators are inserted into the queue. Ground operators cannot be refined anymore and are therefore not inserted back.

This method aims to *balance* the partially instantiated representation. That is, we want to avoid some operators to be ground while others still have a high valence. Although not further considered, we can think of more sophisticated operator selection algorithms. For example, one could try to prioritize operators that have many instances applicable in one step of the encoding. Since only one instance per operator can be applied in each step (see Section 2.4.2), this strategy could effectively shorten the makespan required to find a plan.

4.2.3. Parameter Selection

The parameters to refine the operator with have to be selected carefully. If few parameters are selected, the chance to prune operators after refinement is low.

Thus, the pruning of other operators might be delayed. This delay can lead to the refinement of operators that could have been pruned if another operator was further refined. Also, the groundness progresses slowly in each iteration and operators might be required to be refined multiple times. Selecting many parameters at once however might be infeasible because refining generates exponentially many operator instances in terms of the selected parameters. Additionally, generating the instances entails unnecessary work if many of the instances are pruned anyway. We propose multiple strategies to deal with parameter selection.

Most Frequent selects the parameter that occurs most often within the operator. Grounding with this strategy is potentially slow but avoids combinatorial blowup while refining. The idea is that selecting the most frequent parameter is more likely to yield ground preconditions than other parameters. Ground preconditions are desirable for detecting prunability since they can be checked for rigidness efficiently. This strategy is used as fallback for the other strategies if they otherwise would not select any parameter.

Min Ground identifies the precondition that has the fewest ground instances and selects all parameters occurring in that precondition. Thus, the strategy aims to keep the number of refined operators low but guarantees that one precondition will be ground in every refined operator.

Min New tries to ground preconditions early while keeping the number of refined operators low. To do so, it identifies the precondition that has the fewest *potentially satisfiable* ground instances and selects all parameters occurring in that precondition. A ground literal is potentially satisfiable if it is either positive and its atom not rigid false or negative and its ground atom not rigid true. All refined operators with preconditions that are not potentially satisfiable are thus never applicable. If the **Rigid** pruning policy is used, only preconditions of rigid predicates are accounted for when counting the potentially satisfiable ground instances. The number of potentially satisfiable ground instances of the selected precondition is an upper bound for the actual number of refined operators that are not pruned. A refined operator might still be detected as prunable because of other preconditions.

Max Pruned is similar to **Min New**. This strategy prunes operators early to avoid refining operators that are later pruned anyways. It therefore identifies the precondition that has the *most unsatisfiable* ground instances and selects all parameters occurring in that precondition. A ground literal is unsatisfiable if it is either positive and its ground atom is rigid false or negative and its ground atom is rigid true. Again, if the **Rigid** pruning policy is used, only preconditions of rigid predicates are accounted for when counting the unsatisfiable ground instances. The number of unsatisfiable ground instances of the selected precondition is a lower bound for the number of pruned refined operators since all refined operators that are prunable because of the selected

precondition are detected as such. Additionally, refined operators might be prunable independently of this precondition.

Effects selects the parameters occurring in an arbitrary effect that is not ground. As the frame axiom formulae (Equations (2.8) and (2.9)) are in DNF (disjunctive normal form, a disjunction of conjunctions of literals), they grow exponentially when converting them to CNF. This strategy aims to remove non-ground effects to make the frame axiom encoding more efficient (Equations 2.8 and 2.9). Ground effects do not contribute to the combinatorial explosion when converting the frame axiom encoding from DNF to CNF. In fact, if all assignments in one effect support are empty (i.e., the corresponding effects are ground), the frame axiom formula for that effect support is in CNF.

The different grounding strategies are evaluated regarding grounding time and operators of intermediate representations in Section 5.4.1.

4.2.4. Parallel Grounding

Multiple aspects of our grounding algorithm can be subject to parallelization. We focused on concurrently refining multiple operators. The parallel algorithm is given in Algorithm 2. The parallel grounding algorithm is nondeterministic and might yield

Algorithm 2: Parallel Grounding

Data: Lifted representation $\Pi_L = (C, O, s_0, g)$, threads p, groundness $r_t \leq 1$
Result: Partially instantiated representation Π'_L such that $r(\Pi'_L) \geq r_t$

```
1  O' ← O;
2  foreach i ∈ [0, ..., p − 1] parallel do
3      while r((C, O', s_0, g)) < r_t do
4          o ← selectOperator(O');
5          V ← selectParameters(o);
6          O' → O' \ {o};
7          forall assignments σ with variables from V do
8              if prune(σ(o)) = false then
9                  O' ← O' ∪ {σ(o)};
10             end
11         end
12     end
13 end
14 filter(O');
15 Π'_L ← (C, O', s_0, g);
16 return Π'_L
```

different partially instantiated representations for $r < 1$ than the sequential version. The differences regard the operator selection (line 4) and the final groundness.

The parallel version selects multiple operators concurrently and thus might refine operators that would not yet be considered by the sequential version. Also, the parallel version generally refines more operators than necessary to reach the given groundness due to the refinement of multiple operators in parallel.

4.3. Encodings

In this section we present multiple SAT encodings that are designed for lifted representations and feature different step semantics. All encodings share the formulae of the base encoding described in Section 2.4.2 thus we only describe modifications of the base encoding for each encoding. We discuss advantages and shortcomings as well as the encoding size of each modification. Let $\Pi_L = (C, O, s_0, g)$ be the planning task and h the horizon to encode. We adopt the notations introduced for the base encoding. The formulae below are instantiated for every $t \in \{0, \dots, h-1\}$ unless otherwise specified.

As discussed in Section 4.2.3, frame axioms grow exponentially when converting to CNF. One option is to introduce new variables to linearize the conversion, similar to the Tseytin transformation [Tse83]. As this is a trade-off between the number of clauses and number of variables, we introduce the DNF threshold $d_{\max}$ to control the maximum number of disjunctions allowed in a DNF before linearizing it. We also investigate the option to reduce this exponential blowup by implying the operator for each parameter and removing do_o^t from $Do_{o,\sigma}^t$ whenever possible. To do so, we add these formulae for each operator $o \in O$ in each step $t \in \{0, \dots, h-1\}$

$$\forall v \in \texttt{param}(o) : \bigwedge_{c \in C} m_{o,v \mapsto c}^t \rightarrow do_o^t \tag{4.2}$$

and change the definition of $Do_{o,\sigma}^t$ in Equation (2.1) to

$$Do_{o,\sigma}^t \coloneqq \begin{cases} do_o^t & \text{if } \sigma = \emptyset, \\ \bigwedge_{v \mapsto c \in \sigma} m_{o,v \mapsto c}^t & \text{otherwise.} \end{cases} \tag{4.3}$$

We evaluate the performance of the following encodings and the presented options to handle conversions to CNF in Section 5.4.5.

Sequential. This encoding realizes sequential step semantics and only allows one action in each step. Thus, for every two distinct operators o_1 and o_2 we add

$$\neg do_{o_1}^t \vee \neg do_{o_2}^t. \tag{4.4}$$

Sequential step semantics generally require greater horizons in order to find plans compared to parallel step semantics and have several other disadvantages, as discussed in Section 2.4.1. Nevertheless, this encoding only adds $\mathcal{O}(|O|^2)$ clauses, so it thus might be preferable if interference is expensive to encode.

Foreach. To encode ∀-step semantics, we need to forbid interfering actions to be applied in one step. Note that only cases (c) and (d) of interference are relevant, since cases (a) and (b) can never occur in one step due to Equations (2.6) and (2.7). We establish ∀-step semantics with

$$\forall l \in L : \bigwedge_{\substack{(o_1,\sigma_1)\in ES_l,\\ (o_2,\sigma_2)\in PS_{\overline{l}},\\ o_1\neq o_2}} \neg Do^t_{o_1,\sigma_1} \vee \neg Do^t_{o_2,\sigma_2}. \tag{4.5}$$

The number of added clauses is in $\mathcal{O}(|C|^{v_\mathrm{p}} \cdot |O|^2)$, with v_p denoting the maximum valence of all predicates. These interference clauses can dominate the overall size of the encoding. Note that this encoding is generally more restrictive than ∀-step semantics allow. It is only possible to apply one ground instance for each encoded operator in each step, while ∀-step semantics allow multiple ground instances of one operator in one step. As the potential parallelism of this encoding increases if multiple instances of the same operator are encoded as distinct operators, the horizon required to find a plan decreases with representations of higher groundness.

Restricted Foreach. We can further restrict the ∀-step semantics described above by only allowing operators to be applied in the same step if they have no interfering instances. This restriction yields a more compact encoding: For operators o_1 and o_2 that have interfering instances we add

$$\neg do^t_{o_1} \vee \neg do^t_{o_2}. \tag{4.6}$$

This encoding requires only $\mathcal{O}(|O|^2)$ additional clauses but forbids potentially non-interfering actions to be applied in the same step. The restricted foreach encoding is equivalent to the sequential encoding if all operators have interfering instances. However, it approaches the step semantics of the foreach encoding with increasing groundness. In fact, it equals the foreach encoding if all operators are ground since no two ground operators can have interfering instances without being interfering themselves.

Exists. We encode ∃-step semantics similar to the idea of Rintanen et al. in [RHN06, Section 3.4.4]. We impose a fixed ordering $o_1 < \cdots < o_{|O|}$ on the operators in O and define $\texttt{rank}(\sigma(o_i)) := i$ for any assignment σ. This ordering determines the order in which actions in each step are applied. Hence, we allow ground instances o and o' to be applied in the same step if o disables o' only if $\texttt{rank}(o) > \texttt{rank}(o')$. The encoding uses the concept of *disabling chains*. Operators activate disabling chains for their effects when applied. At the same time, they cannot be applied if a disabling chain for any precondition was activated by an operator with higher precedence (hence the name). We introduce helper variables $c^t_{a,o}$ and $c^t_{\neg a,o}$ for every $a \in A$, $o \in O$ and $t \in \{0,\ldots,h-1\}$ to encode the disabling chains for every ground literal. Figure 4.1 illustrates the encoding of disabling chains. Since $\neg l$ is a precondition of $\sigma_k(o_k)$, this

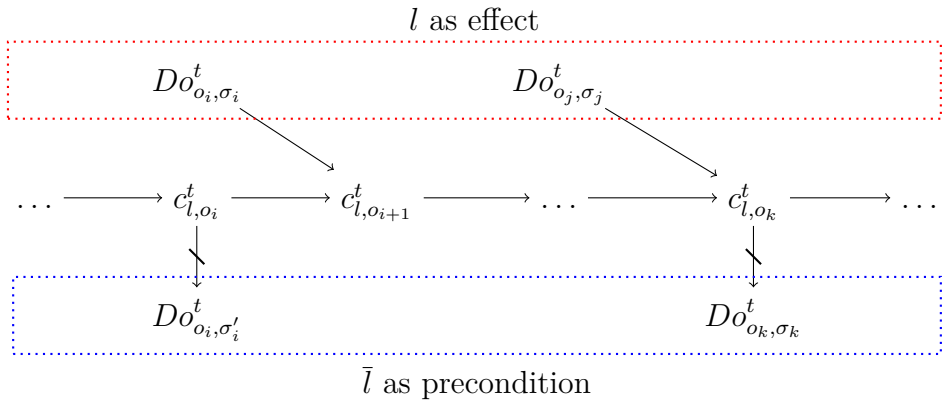


Figure 4.1.: Disabling chain for a ground literal l with $i < j = k - 1$. Arrows between formulae indicate implication, crossed out arrows indicate implication of the negation. In this example, $(o_i, \sigma'_i), (o_k, \sigma_k) \in PS_{\bar{l}}$ and $(o_i, \sigma_i), (o_j, \sigma_j) \in ES_l$.

operator must not be applied if another operator with lower rank has l as effect. In this case, both $\sigma_i(o_i)$ and $\sigma_j(o_j)$ have lower rank than $\sigma_k(o_k)$ and l as effect. Thus, if any ground instance of $\sigma_i(o_i)$ or $\sigma_j(o_j)$ is applied, c^t_{l,o_k} is set to `true` and prevents any ground instance of $\sigma_k(o_k)$ from being applied in the same step. The encoding then is as follows. The helper variables form a chain

$$\forall l \in L, i \in \{1, \ldots, |O| - 1\} : c^t_{l,o_i} \to c^t_{l,o_{i+1}}, \tag{4.7}$$

effects activate a chain

$$\forall l \in L : \bigwedge_{\substack{(o,\sigma) \in ES_l, \\ \texttt{rank}(o) < |O|}} Do^t_{o,\sigma} \to c^t_{l,o_{\texttt{rank}(o)+1}} \tag{4.8}$$

and active chains deactivate operators with corresponding preconditions

$$\forall l \in L, (o, \sigma) \in PS_{\bar{l}} : c^t_{l,o} \to \neg Do^t_{o,\sigma}. \tag{4.9}$$

These constraints add $\mathcal{O}(|A| \cdot |O|)$ clauses, therefore the exists encoding is smaller than the foreach encoding on most instances. Also, this encoding is always at least as permissive as the foreach encoding, because every two operators that can be applied in one step according to $\forall$-step semantics can also be applied in one step with this encoding. However, this encoding requires $\mathcal{O}(|A| \cdot |O|)$ additional variables, which can harm the SAT solver's performance. Compared to the encoding proposed by Rintanen et al. for ground representations, this encoding is more restrictive. Since the ordering in our encoding is imposed on the operators of O, every ground instance of o_1 has lower rank than any ground instance of o_2. The higher the groundness, the more precise orderings are allowed.

4.4. Planning Algorithms

In this section we present our planning algorithms. They each use different strategies to schedule the grounding process and SAT solving. These strategies involve heuristics to determine whether to proceed grounding, trigger a new SAT solving attempt and when to stop a SAT solver. The planning algorithms are evaluated regarding time to find plans and memory usage in Section 5.4.3.

4.4.1. Smallest Encoding

The idea of this planning algorithm is to scan partially instantiated representations of different groundness to find the one yielding the smallest encoding. The SAT solver then uses this encoding to find a plan. More precisely, given a granularity $k \geq 1$, we generate F_1 for each partially instantiated representation with groundness $\frac{i}{k}, i \in [0, \ldots, k]$ and select the smallest formula of them. Experiments suggest that using the number of variables is a sensible metric for encoding size (see Section 5.4.3). The pseudo-code for this planning algorithm is given in Algorithm 3. A time limit can be imposed to the loop in line 3 to ensure that solving is attempted for planning tasks where grounding would exceed the total time limit.

Algorithm 3: Smallest Encoding

Data: Lifted representation $\Pi_L = (C, O, s_0, g)$, granularity $k \geq 1$, step factor $\gamma > 1$
Result: Plan P

```
1  Π̂_L ← Π_L;
2  i ← 1;
3  while i ≤ k do
4  |  Π_L ← proceedGrounding(Π_L, i/k);
5  |  if |encode(Π_L)| < |encode(Π̂_L)| then
6  |  |  Π̂_L ← Π_L;
7  |  end
8  |  i ← i + 1;
9  end
10 m ← solve(Π̂_L, γ);
11 return toPlan(m);
```

The method `encode` takes the representation to encode as an argument and returns the corresponding boolean formula. `proceedGrounding` takes a representation and a target groundness as argument and grounds the given representation until the target groundness is reached (see Section 4.2). Note that the groundness is calculated with respect to the initial lifted representation. The `solve` routine repeatedly encodes the representation with horizons $\lceil\gamma^0\rceil, \lceil\gamma^1\rceil, \ldots$ until a satisfiable formula is found and the model can be returned. As Rintanen mentioned in [Rin14], proving the

unsatisfiability of formulae is much harder for SAT solvers than showing satisfiability. To prevent the planner from spending too much time on unsatisfiable formulae, we introduce *step skipping*. Initially, each SAT solving attempt is interrupted after a given timeout. If a solving attempt is interrupted, the current formula is deemed unsolvable and the encoding for the next horizon is solved. The step skipping parameter controls how many attempts in a row may fail due to the timeout before the timeout is lifted.

We can also make use of information about pruned operators if the representation with the smallest encoding is not the representation with the highest groundness. For each tuple of operator o in the representation to encode and assignment σ where $\sigma(o)$ is pruned in the representation with the highest groundness, we can add

$$\neg Do^t_{o,\sigma} \tag{4.10}$$

for each step t to the encoding.

4.4.2. Interruptive Planning

This approach tries to solve encodings for partially instantiated representations with increasing groundness. Basically, we alternate between grounding and SAT solving. If the SAT solver takes too long to solve a formula, the encoding is deemed infeasible and dismissed in favor of an encoding for a representation with higher groundness.

This approach can be advantageous over the previous one if the encoding size does not correlate with the difficulty to find a model for the encoding. Also, it might be necessary to encode a representation with high groundness albeit it has a large encoding if plans are very long and the increased step parallelism is required to keep the horizon low (see Section 4.3). A disadvantage is that solving attempts might be interrupted prematurely. That is, time is spent solving encodings that might later be discarded in favor of inferior encodings. Algorithm 4 illustrates the described approach. The `trySolve` method essentially does the same as `solve` but terminates after the given timeout. If the timeout occurs before a model to a satisfiable formula can be found, the returned model is empty.

We can interleave the individual SAT solving attempts with grounding instead of grounding after `trySolve` times out. Information about pruned operators can then be incorporated in subsequent SAT solving attempts for higher horizons analogous to Equation (4.10).

Algorithm 4: Interruptive SAT Planning

Data: Lifted representation $\Pi_L = (C, O, s_0, g)$, granularity $k \geq 1$, solving timeout t, step factor $\gamma > 1$
Result: Plan P

```
1  i ← 1;
2  while i < k do
3  |   m ← trySolve(Π_L, γ, t);
4  |   if m ≠ ∅ then
5  |   |   return toPlan(m);
6  |   else
   |   |   // trySolve timed out
7  |   |   Π_L = proceedGrounding(Π_L, i/k);
8  |   |   i ← i + 1;
9  |   end
10 end
11 Π_L = proceedGrounding(Π_L, 1);
12 m ← solve(Π_L, γ);
13 return toPlan(m);
```

4.4.3. Parallel Planning

We extend the idea of the interruptive planning by parallelizing the approach. Concurrently solving multiple encodings at once mitigates the problem with premature interruption of solving attempts. However, this approach consumes more memory since multiple SAT solvers have to be maintained simultaneously. The parallel version is given in Algorithm 5. This approach can also make use of the parallel version of the grounding routine. Since the planning task needs to be further grounded prior to a new solving attempt (line 7), all threads that are not currently occupied with solving an encoding can participate in the grounding process. Specifically, $p-i-1$ threads can concurrently ground the task to groundness $\frac{i+1}{p-1}$. The algorithm terminates as soon as any thread returns a plan (line 4).

Algorithm 5: Parallel Grounding and SAT solving

Data: Lifted representation $\Pi_L = (C, O, s_0, g)$, threads $p \geq 2$, step factor $\gamma > 1$
Result: Plan P

```
1  foreach i ∈ [0, ..., p − 1] do
2      in new thread do
3          m ← solve(Π_L, γ);
4          return toPlan(m);
5      end
6      if i ≠ p − 1 then
7          Π_L = proceedGrounding(Π_L, (i+1)/(p−1));        // By p − i − 1 threads
8          ;
9      end
10 end
```

5. Experimental Results

We first describe implementation details of the components of our planners and present our hardware and software setup. We then introduce the planning tasks we used for the experimental evaluation. We also show how we attain a reasonable configuration of the tuning parameters mentioned in the implementation details and the previous chapter. We further evaluate the performance of the proposed grounding procedure (see Section 4.2) and the planning algorithms (see Section 4.4) in combination with the different encodings presented in Section 4.3. Finally, we compare the performance of our planners with other successful, openly available planning systems.

5.1. Implementation

We implemented our planners with C++17. We compiled the programs using g++ version 9.2.0 with optimizations `-O3 -march=native` enabled.

The planning process consists of reading and parsing the input, normalizing and grounding the planning task, encoding the planning task as SAT instances and solving these instances with a SAT solver. We implemented all but the SAT solver ourselves and have no third-party dependencies. Our implementation supports IPASIR, a C header to interface with incremental SAT solvers [Bal+16]. Thus, our implementation can utilize any SAT solver that implements the IPASIR interface by linking against it.

The time and memory resources required to perform input processing as well as the normalizing are negligible for all tested inputs and therefore not explicitly considered in our experiments.

As PDDL has type support for constants, variables and parameters, we use that additional information when available to restrict the number of possible ground atoms and ground operators.

5.1.1. Grounding

Implementing the grounding routine efficiently is crucial to the performance of our planners. As discussed earlier, the number of operators in a ground representation might be exponential in the number of parameters of the operators in the lifted representation. Therefore, both time efficiency and memory consumption have to be considered. In the following we present how we select the next operator to refine,

compute the groundness and speed up the pruning test. Lastly, we discuss the measures we take to parallelize the grounding process.

Operator Selection. The grounder maintains a list of the operators comprising the current partially instantiated representation. The grounding itself consists of repeatedly refining each operator in the list. The refined operators are first buffered in a new list. After all operators of the list are refined, we replace the operator list with the buffer. For the purpose of fast iterating through the list and keeping allocation costs low, both the current list and the buffer are continuous in memory. The number of all refined operators of the operators currently in the list can be overestimated to preallocate the buffer.

Computing the Groundness. The groundness of the partially instantiated representation maintained by the grounder is computed online. Whenever an operator is pruned, we add the number of possible ground operators of that operator to a counter. Then the groundness is the size of the operator list plus that counter divided by the total number of possible ground operators.

Detecting Prunability. Depending on the pruning policy (see Section 4.2.1), testing whether an operator is prunable is computationally expensive and can dominate the time required to ground a planning task. Unless rigid pruning is used, we repeatedly need to check if a ground atom is useless, rigid true or rigid false in order to test an operator for prunability. Thus, a naïve approach is to check if any operator in the operator list contradicts the rigidness or uselessness of the ground atom to check. We speed up this check by caching information about previously checked ground atoms.

- *Successful Caching.* The idea is to cache rigid and useless atoms. Once we determine that a ground atom has any of those properties, we insert it into a corresponding hash map. If we later check this ground atom again, we find it in the hash map and can skip iterating over all operators. Thus, repeated checks of rigid or useless ground atom do not require to iterate over all operators again. Once a ground atom is determined to be rigid or useless, it keeps this property for the whole grounding process, therefore the caches do not need to be cleared.

- *Unsuccessful Caching.* We extend this idea to also cache unsuccessful checks. That is, whenever we determine that a ground atom is not rigid or useless, we insert it into corresponding hash maps. In order to be accurate, this cache would have to be cleared whenever an operator is pruned as this can render the effects rigid and the preconditions useless. At the cost of occasional false negatives when checking ground atoms for rigidness or usefulness, we clear the cache every time after all operators in the current operator list have been refined.

While checking if an operator can be pruned, we additionally simplify it by removing unnecessary preconditions and effects. A precondition is unnecessary if it always holds (i.e., requires either a rigid true atom to hold or a rigid false atom to not hold). An effect is unnecessary if it either affects useless atoms or upholds rigid atoms. Simplifying operators reduces the memory usage during grounding, can speed up the tests for rigidness and usefulness and reduces the size of the encoding, thus benefiting the SAT solver.

Parallel Grounding. The implementation of our parallel grounding approach is straightforward. The list of operators is equally divided up between all participating threads. To avoid contention on the buffer of refined operators, each thread maintains its own buffer. Each time after all operators in the list have been refined, the threads' buffers are merged to replace the operator list.

Since multiple threads access the caches concurrently, we need to make them thread-safe. We do so by guarding every reading and writing access to them with mutexes. To reduce the contention when accessing the caches, we use one mutex for each predicate that guards only the part of the cache containing ground atoms of this predicate.

5.2. Experimental Setup

We conduct all experiments on a computer equipped with an AMD EPYC 7702P processor with 64 cores running at 2.0 GHz and 1 TB of DDR3 RAM. The operation system is Ubuntu 19.10 using version 5.3.0-40-generic of the linux kernel.

Each experiment for parameter tuning was executed on a dedicated core and repeated three times. All data for repeated runs is arithmetically averaged. For the comparison with other planners we ran each planner once for each planning task. We impose a time limit of 5 minutes and a memory limit of 10 GB for each run and inform the planners about these limits when possible. We run as many experiments in parallel as possible such that each planner has a dedicated core and enough memory is exclusively available.

All experiments were scheduled, supervised and evaluated using the Downward Lab [Sei+17]. All plans have been validated with the external plan validator VAL [HLF04].

5.3. Planning Tasks

All planning tasks are given in a lifted representation in the PDDL format to the planners. Planning tasks in PDDL are comprised of two files. The *domain file* contains the operators as well as information about the types and predicates (the *domain* of the task). The *problem file* contains the constants, the initial state and the goal. Typically, there are multiple planning tasks that share the same domain

file. Tasks using the same domain file naturally share many characteristics and are therefore often grouped together by their domain. Unless otherwise noted, we use the geometric mean to summarize data from experiments with different problems of the same domain.

The main sources of planning tasks we use for our experiments are the *International Planning Competition* and the *Sparkle Planning Challenge*. Detailed information about the planning tasks can be found in Appendix A. In total we conducted experiments on 558 planning tasks.

We only use the 508 planning tasks from the IPC 2014 and the IPC 2018 (the IPC set) for the parameter tuning. While we also compare our planners to the competition on these tasks, the 50 tasks from the Sparkle Planning Challenge 2019 (the Sparkle set) are exclusively used for the comparison with other planners. Note that our planners are optimized for the imposed time and memory constrains.

All planning tasks we use in our experiments are solvable. SAT based planning as presented is particularly unsuited to prove that no plan exists for a given task. Thus, we try to find a plan until the resources are exhausted. Every task for which we fail to find a plan is counted as unsolved.

5.4. Parameter Tuning

Before evaluating the performance of our planners and comparing them to the competition, we need to find optimal settings for the available tuning parameters. Our grounding routine as well as the planning algorithms themselves offer many opportunities for tuning and configuration. We first perform tests to analyze our grounding routine independently from the actual planning. We then analyze the influence of tuning parameters and encodings to our planning algorithms. Finally, we test different SAT solvers for our planner.

In this section, the term *coverage* is, depending on the context, either the number of planning tasks the grounder is able to ground or the number of planning tasks the planner is able to solve within the time and memory constraints. We often use *cactus plots* to display the coverage. The coverage is on the x-axis and the time on the y-axis. A point on a curve then gives the number of tasks grounded/solved within the corresponding time per task.

The *size* of a representation is the number of operators the representation uses. We use the same resource limitations for the parameter tuning as we do for the other experiments.

5.4.1. Grounding

For the purpose of tuning the grounding process we consider the time it takes to fully ground the planning tasks, the size of intermediate representations as well as the size of the fully grounded representation. We also look at memory consumption,

Pruning policy	Rigid
Parameter selection	Most frequent
Caching	None

Table 5.1.: Grounder baseline

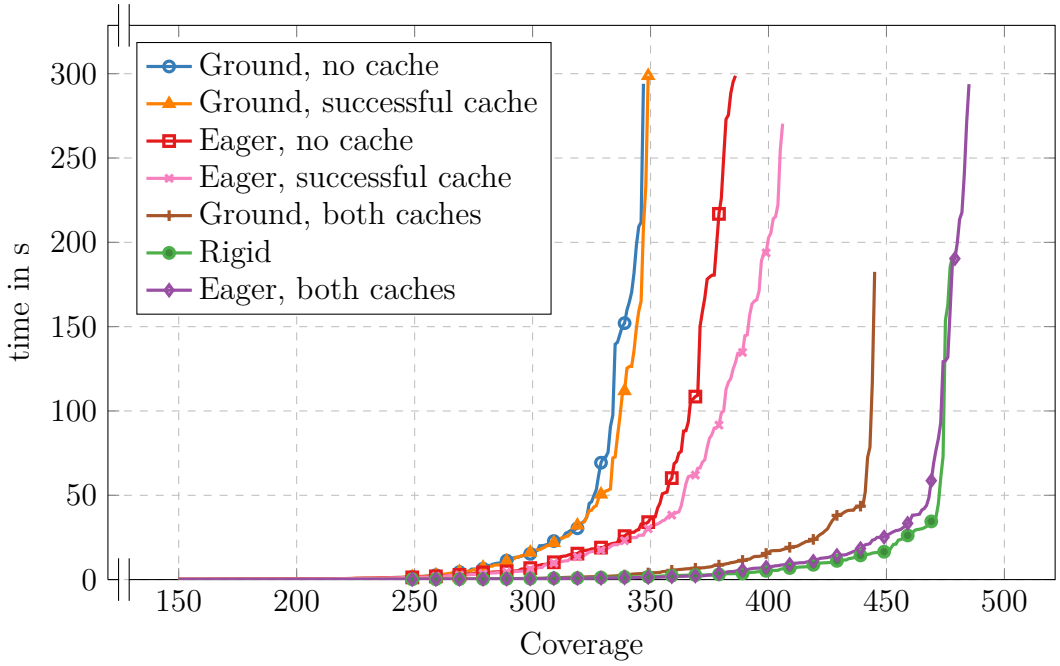


Figure 5.1.: The impact of caching and different pruning policies to the grounding coverage. The eager and ground pruning policies are tested without caching, the successful caches and both caches, respectively. Rigid pruning cannot make use of caching.

although no measures are taken by our grounder to react to memory limitations. The configuration of our baseline grounder is given in Table 5.1.

Pruning Policy. To evaluate the presented pruning policies, we first look into the effects of caching. Since rigidness and uselessness can be computed very efficiently with the rigid pruning policy, we only consider caches for the other two policies. Figure 5.1 shows the coverage for the pruning strategies with different caching methods as a cactus plot. Table 5.2 gives the coverage broken down by domain. Without the use of caching, rigid pruning clearly dominates the other policies in terms of coverage. Generally, the ground policy performs worst. Using only the successful cache does not significantly improve any policy. When using both caches, eager pruning outperforms rigid pruning slightly (7 more tasks grounded). We further consider only configurations that either use both caches or the rigid pruning policy. While the ground pruning policy is inferior on every domain, both rigid

Domain	Pruning Policy		
	Rigid	Ground	Eager
barman (34)	34	34	34
childsnack (30)	30	30	30
data-network (40)	22	22	**33**
floortile (30)	30	30	30
ged (40)	40	40	40
hiking (40)	40	40	40
openstacks (40)	40	40	40
organic-synthesis (40)	40	40	40
snake (40)	40	40	40
termes (40)	40	40	40
tetris (37)	**37**	7	36
thoughtful (20)	**8**	5	5
transport (37)	37	37	37
visitall (40)	40	40	40
Sum (508)	478	445	**485**

Table 5.2.: Comparison of the number of grounded problems per domain with different pruning policies. Ground and eager pruning both use successful and unsuccessful caching. The *most frequent* parameter selection strategy is used for all policies. Bold numbers indicate single best values.

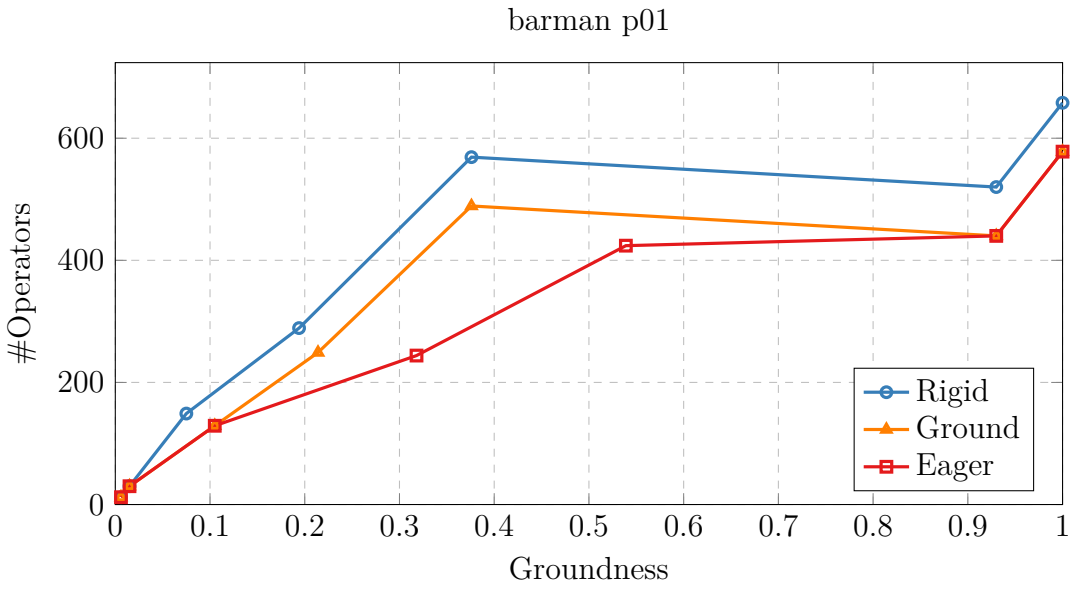


Figure 5.2.: The groundness progress during grounding for the first task of the domain "barman". Each data point represents the groundness after all operators in the operator list have been refined.

and eager pruning excel on some domains. Only the domains "data-network" and "thoughtful" contain tasks that could not be grounded by any configuration.

The pruning policy controls if and when operators are detected as prunable during grounding, so the number of operators generated during grounding as well as in the size of the final representation depend on the policy. The number of operators for any given groundness is in increasing order for the eager, ground and rigid policy. However, the ground representation is the same with eager and ground pruning. With the rigid policy, only a subset of operators pruned by the other policies are pruned, thus the final representation generally has more operators. Figure 5.2 shows this behaviour exemplary for one task from the domain "barman". Note that the number of operators does not necessarily increase monotonically. If this is the case, all refined instances of an operator are detected as prunable, however the original operator was not detected as prunable before (see Section 4.2.1 for details). The ground pruning policy evidently does not offer any advantage over any other policy and thus is not considered further.

Figure 5.3 shows that only very few tasks can be grounded with rigid pruning that cannot also be grounded with eager pruning. Finally, Table 5.3 lists the size of the ground representation and memory consumption by domain. The additional memory required by the eager policy is marginal in most cases. The only outlier is the domain "visitall", where the memory usage is higher by a factor of 3.45. However, the memory conserved by pruning more operators with the eager policy can be significant. Most notably is the domain "data-network", where only 2.4% of the memory is required compared to rigid pruning. While the number of operators

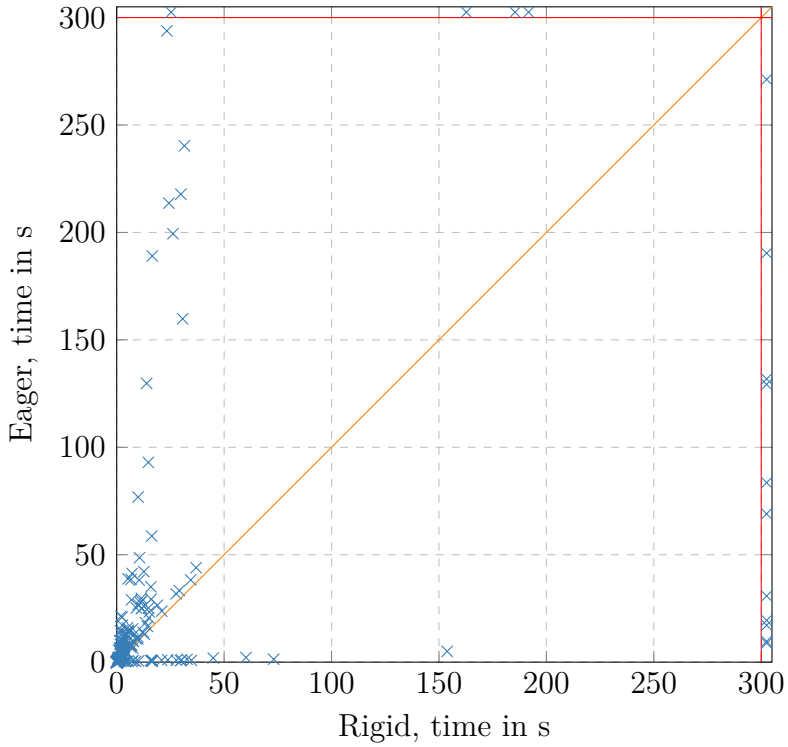


Figure 5.3.: Comparison of the time to ground each task with the eager and rigid pruning policy. Marks beyond the red lines indicate that this task was not grounded with the corresponding policy and tasks that were grounded by neither approach are omitted.

is the same for some domains with both policies, it is notably lower for the domains "snake" and "tetris" (factor ∼10).

Parameter Selection. The parameter selection for refinement influences the number of refined operators during the grounding process. However, it does not affect the size of the final ground representation. Figure 5.4 shows the impact of parameter selection to the number of grounded tasks. We are able to ground every planning task in the test suite (508) with the *rigid* pruning policy using the *max pruned* parameter selection for refinement. Every configuration using *min ground*, *min new* or *max pruned* is superior to any configuration with *effects* or *most frequent*. The *eager* pruning policy yields the highest coverage with the *min ground* parameter selection (505), closely followed by *max pruned* selection (499).

Figure 5.5 shows that the size of intermediate representations can vastly differ depending on the selection strategy. Also, the shapes of the curves for each selection strategy are dissimilar across the domains. As no strategy dominates the others, we cannot predict their performance in our planners domain-independently.

Domain	Rigid		Eager	
	Operators	Mem. (MiB)	Operators	Mem. (MiB)
barman (34)	2165.02	17.77	1794.52	**17.51**
childsnack (30)	3714.38	**23.42**	3714.38	23.44
data-network (22)	2085.27	1306.17	1868.92	**30.80**
floortile (30)	371.30	**15.32**	371.30	15.50
ged (40)	2008.91	**18.21**	1966.46	18.27
hiking (40)	9673.10	**44.27**	9673.10	44.28
openstacks (40)	13 897.78	**49.81**	13 897.78	50.14
organic-synthesis (40)	170 203.22	127.57	65 908.24	**70.95**
snake (40)	272 776.55	486.53	22 716.04	**61.99**
termes (40)	1072.22	16.32	958.75	**16.21**
tetris (7)	43 255.28	92.35	4511.25	**23.15**
thoughtful (5)	40 308.00	932.45	38 885.40	**928.80**
transport (37)	15 022.36	59.02	15 022.36	**54.56**
visitall (40)	1810.56	**21.10**	1810.56	72.97

Table 5.3.: Comparison of the number of grounded problems as well as required memory per domain. Eager pruning uses both successful and unsuccessful caching. The *most frequent* parameter selection strategy is used for both policies. Only tasks groundable with both policies are considered. The number of these tasks is indicated in parentheses in the domain column.

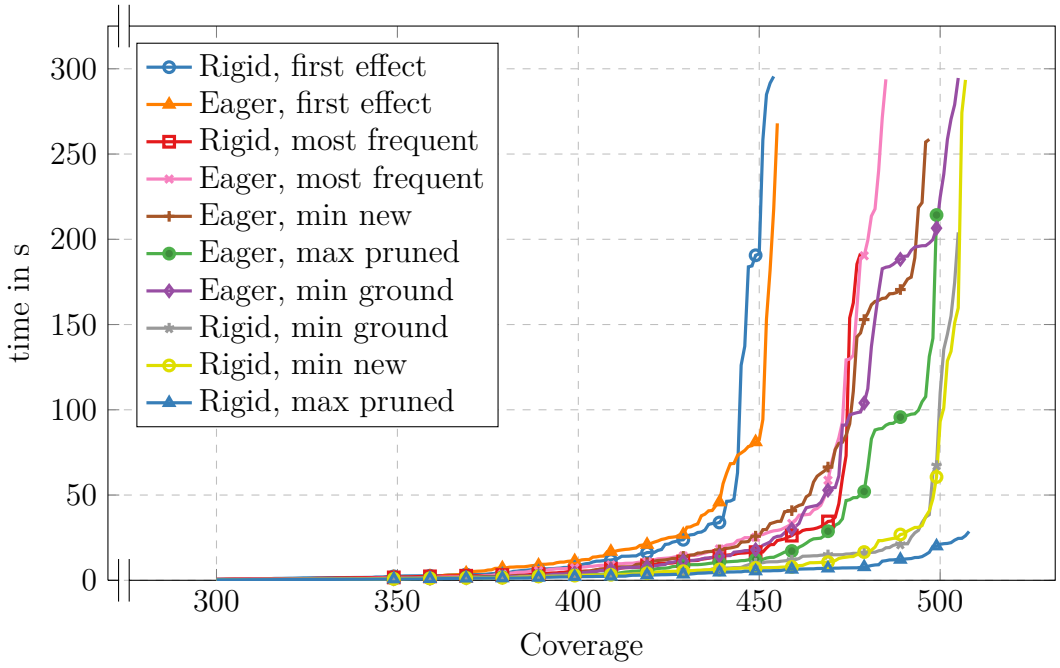


Figure 5.4.: The impact of parameter selection strategies to the number of grounded tasks.

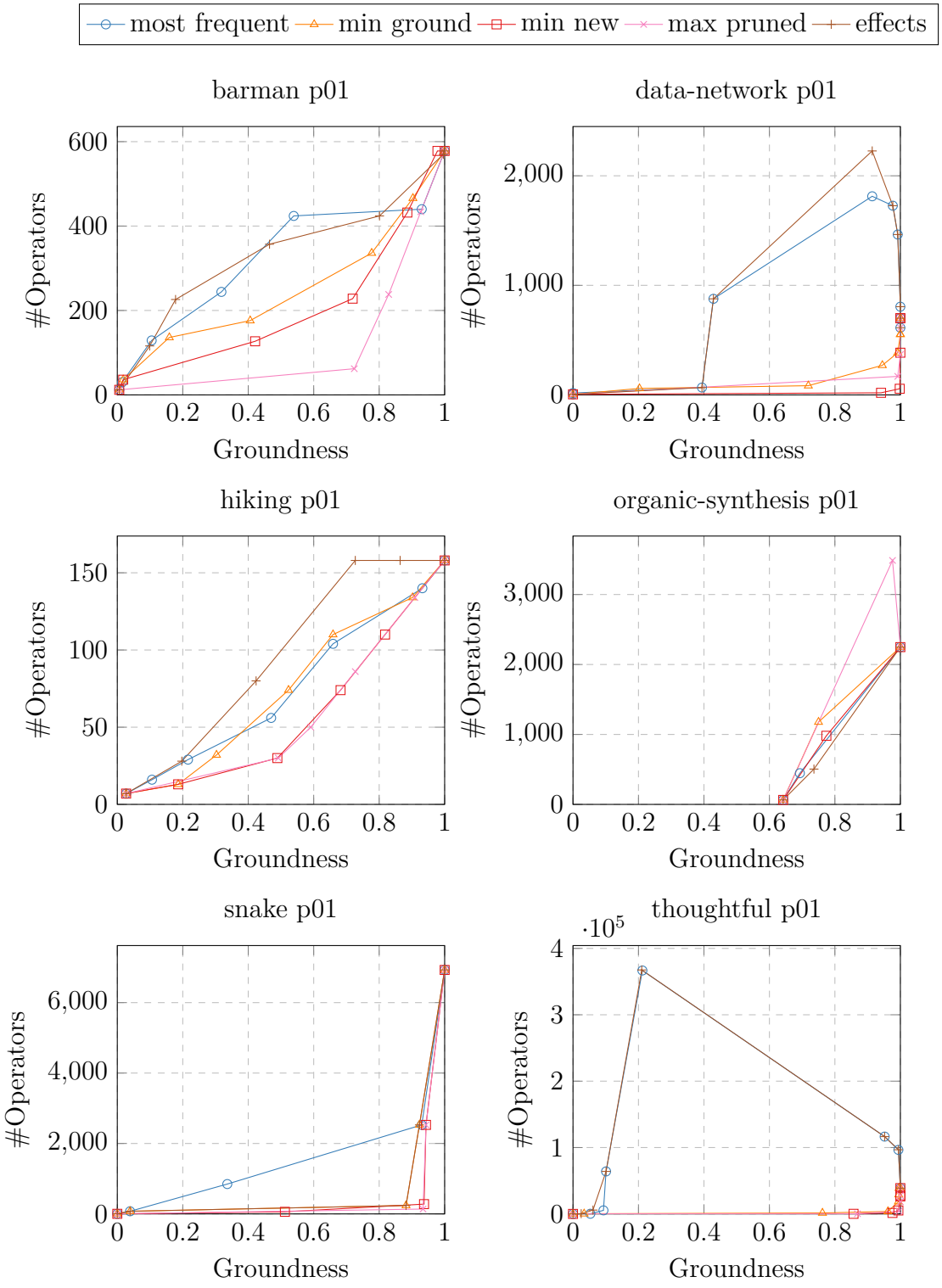


Figure 5.5.: The change of the number of operators in intermediate representations for different parameter selection strategies shown for selected tasks.

5.4.2. Parallel Grounding

We conduct experiments with the parallel grounder presented in Section 4.2.4 with eager pruning and *min ground* parameter selection. Figure 5.6 shows that parallelization improves the overall grounding time. Table 5.4 gives the grounding times exemplary for some tasks, the speedup on these tasks for up to 16 threads is shown in Figure 5.7. As Figure 5.8 shows, we can achieve significant speedups on tasks that are difficult to ground. For some tasks, mostly from the "tetris" domain, we achieve almost linear speedup. On some tasks of the "tetris" and "thoughtful" domains, the grounding time is more than halved with 2 threads. This effect is most likely due to caching the rigid and useless ground atoms. Refining multiple operators at once fills the caches more quickly, making pruning more efficient. Also, parallel grounding is more robust against an unfavorable operator selection order.

However, the parallel grounder solves fewer tasks the more threads are used. While the sequential grounder is able to ground almost all tasks, the imposed synchronisation overhead for parallelization impacts the performance negatively on some domains. This overhead is most severe for the "data-network" domain where we only can ground 29 tasks with 4 threads compared to 39 tasks with sequential grounding. For most tasks that take only a short time ($<$1 s) to ground sequentially, the performance gets worse the more threads are used.

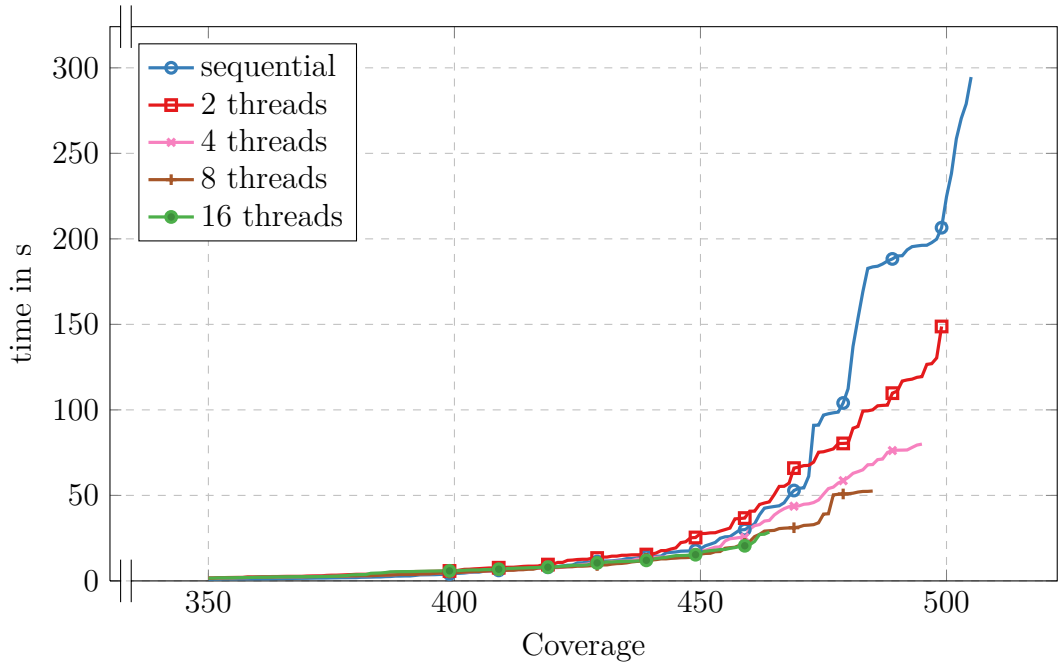


Figure 5.6.: Parallel grounding with eager pruning and *min ground* parameter selection. The legend shows the number of threads used. The data of the "sequential" curve is obtained by the sequential grounder with the same configuration.

Task		Threads			
	seq	2	4	8	16
data-network p21	4.45	7.55	13.79	11.58	13.75
data-network p24	29.98	75.51	53.99	-	-
organic-synthesis p05	0.29	1.42	1.94	3.36	5.67
snake p33	7.54	5.49	3.91	2.42	2.58
tetris p40	193.56	77.17	47.33	25.93	17.98
thoughtful p06	90.98	67.53	40.60	30.59	28.54
thoughtful p20	169.09	80.39	67.90	50.64	-

Table 5.4.: The time needed to ground various tasks depending on the number of used threads. The values for the "seq" column are obtained from the sequential grounder.

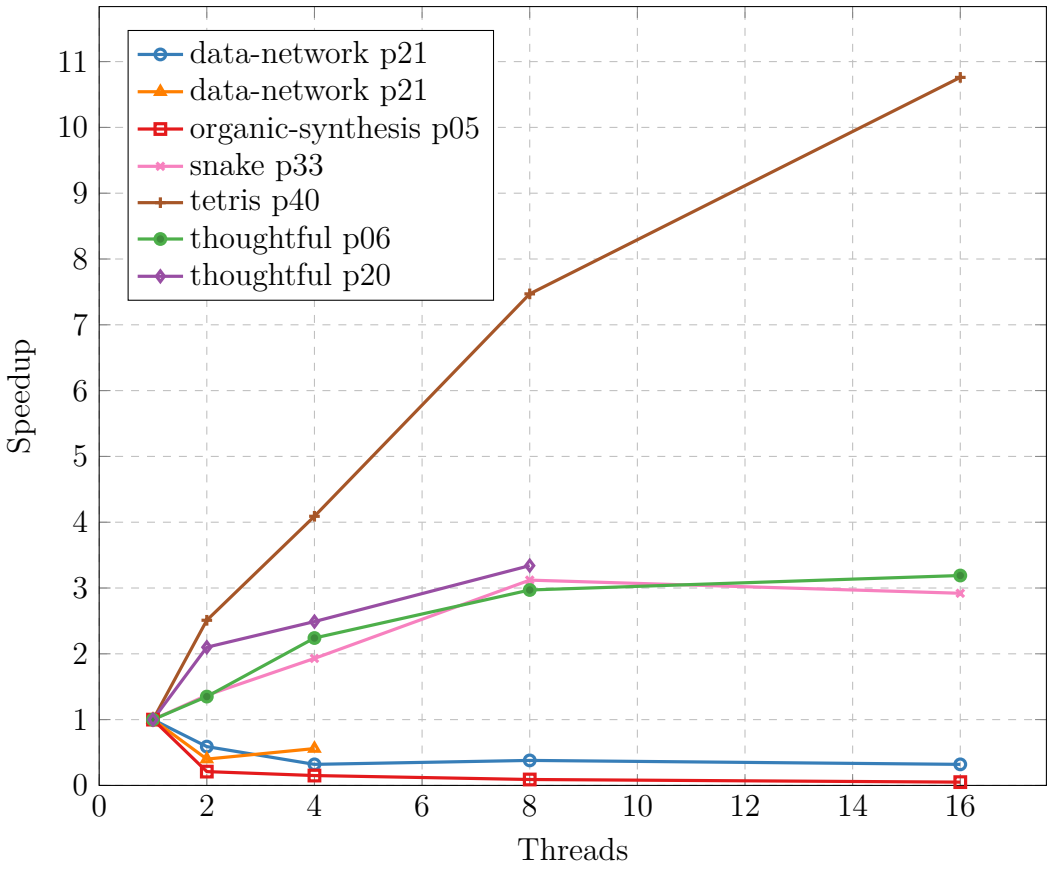


Figure 5.7.: The speedup for the tasks listed in Table 5.4.

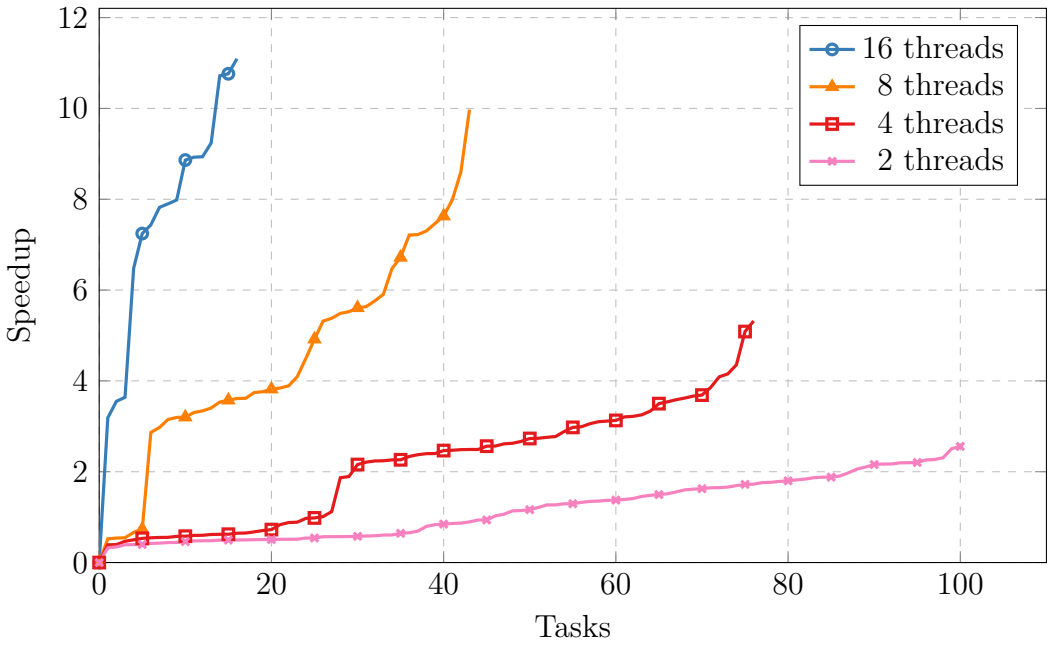


Figure 5.8.: Weak scaling on "hard" tasks. That is, the curve for t threads shows the speedup for tasks that take the sequential grounder more than $2t$ s to ground and are also grounded by the parallel grounder.

5.4.3. Baseline Planner

In this section we establish a baseline to tune our planners. Our planners share many tuning parameters, which we tune in the following sections using our baseline planner.

Table 5.5 lists parameters shared by all planners along their baseline configuration. Using these settings, we first examine planning with fixed groundness. We therefore sample representations with groundness $g \in [0, 0.2, 0.4, 0.6, 0.8, 1]$ for each task and solve them using the baseline configuration. The groundness of some representations can differ from g, since the groundness increases in discrete steps during grounding. The coverage for the planners with different grounder configurations is given

Step factor γ	1.4
Max step skip $s_{\max}$	0
Step timeout t_{step}	-
Encoding	foreach
DNF threshold $d_{\max}$	4
Parameter implies action	no
Sat solver	*Glucose* [AS09] version 4.0

Table 5.5.: Baseline configuration

Configuration		Groundness						Virtual
		0	0.2	0.4	0.6	0.8	1	best
Rigid	Min Ground	**112**	92	105	107	103	108	144
	Effects	**121**	113	111	104	103	114	155
	Max Pruned	120	123	**125**	112	109	115	156
	Min New	120	111	102	**121**	116	114	153
	Most Frequent	**121**	109	115	108	106	109	157
Eager	Min Ground	124	110	120	115	110	**136**	175
	Effects	124	**125**	121	121	116	123	160
	Max Pruned	123	128	133	118	114	**138**	174
	Min New	123	119	112	126	118	**135**	172
	Most Frequent	123	128	125	121	115	**129**	173

Table 5.6.: The planner coverage with different grounder configurations. The groundness is fixed for each run. "Virtual best" indicates the coverage that a planner would achieve if the optimal groundness was chosen for each task individually. Bold values indicate the best groundness for each configuration.

in Table 5.6.

The results show that eager pruning is evidently superior to rigid pruning. Full grounding performs best with eager pruning for all but the *effects* parameter selection. The configuration using *max pruned* yields the overall highest coverage for any fixed groundness (138). However, the *min ground* parameter selection enables the planner to potentially solve the most tasks (175) if the optimal groundness was chosen for each task. Figure 5.9 indicates that not grounding the task at all is the best option for many tasks. Overall, only few tasks are solved best with groundness other than 0 or 1. With a groundness of 1, more tasks are solved the fastest with *min ground* (56) than with *max pruned* (43). Nevertheless, two tasks are solved more using the latter selection strategy. Generally, both selection strategies seem to perform similarly well. We only consider eager pruning with the *min ground* parameter selection strategy for further experiments.

Grounding can also be counterproductive, as Figure 5.10 shows. The tasks solved with either no grounding or full grounding are very dissimilar. This observation matches the fact that the number of tasks solvable with the optimal individual groundness is 25 % to 40 % higher than the highest achieved coverage, depending on the grounder configuration. Our planners try to exploit this discrepancy by solving planning tasks with partially instantiated representations of varying groundness. Very few tasks are solved with no grounding in the latter half of the running time.

For tuning the parameters mentioned in Table 5.5, we use the smallest encoding planner (see Section 4.4.1). It determines the groundness yielding the representation with the smallest encoding and tries to solve the task using that representation.

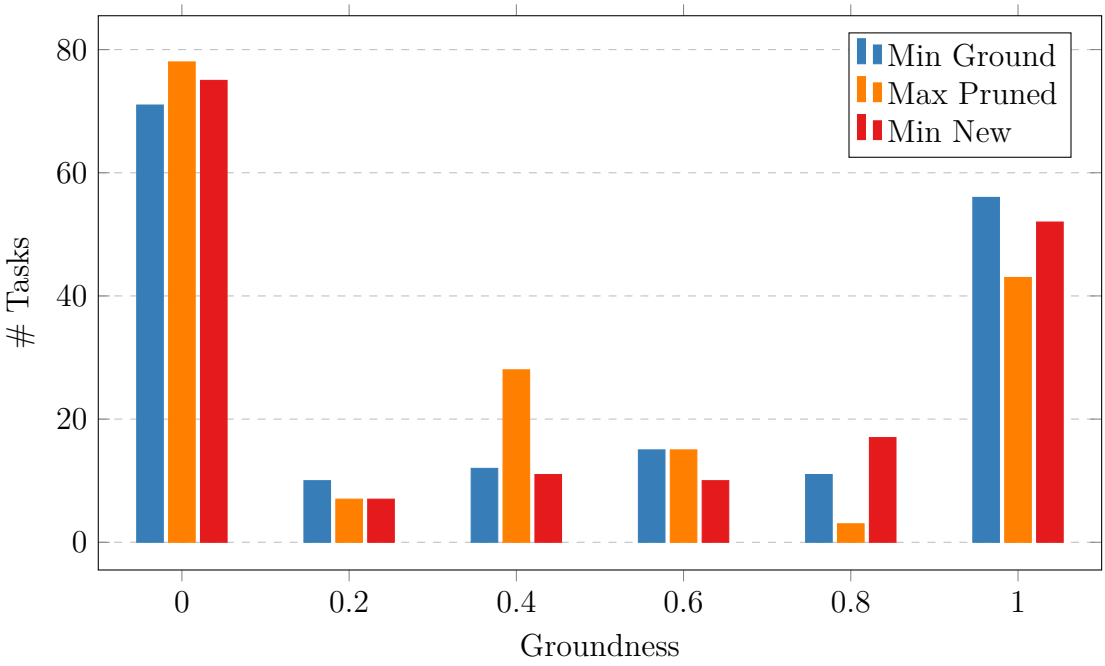


Figure 5.9.: The bars indicate the number of tasks that are solved the fastest with the corresponding groundness. All shown configurations use eager pruning.

The number of tasks solved using the smallest encoding depends on the measure of encoding size. Table 5.7 lists the number of potentially solvable tasks with different size measures. It shows that the encoding size in general is a good indicator for the success of the planner. All of the listed measures yield a better coverage than the best fixed groundness. Our smallest encoding planner uses the number of variables as measure. We use a sampling granularity $k = 3$ and a time limit of 60 s to find the smallest encoding as baseline.

Configuration	Clauses	Variables	Geom. mean
Min Ground	145	151	149
Max Pruned	145	150	150

Table 5.7.: The number of hypothetically solved tasks using different measures of encoding size. A task is counted as solved if it is solved by the planner using the fixed groundness that also yields smallest encoding. “Clauses” and “Variables” use the number of clauses and variables in the formula as metric, respectively. “Geom. mean” is the geometric mean of clauses and variables.

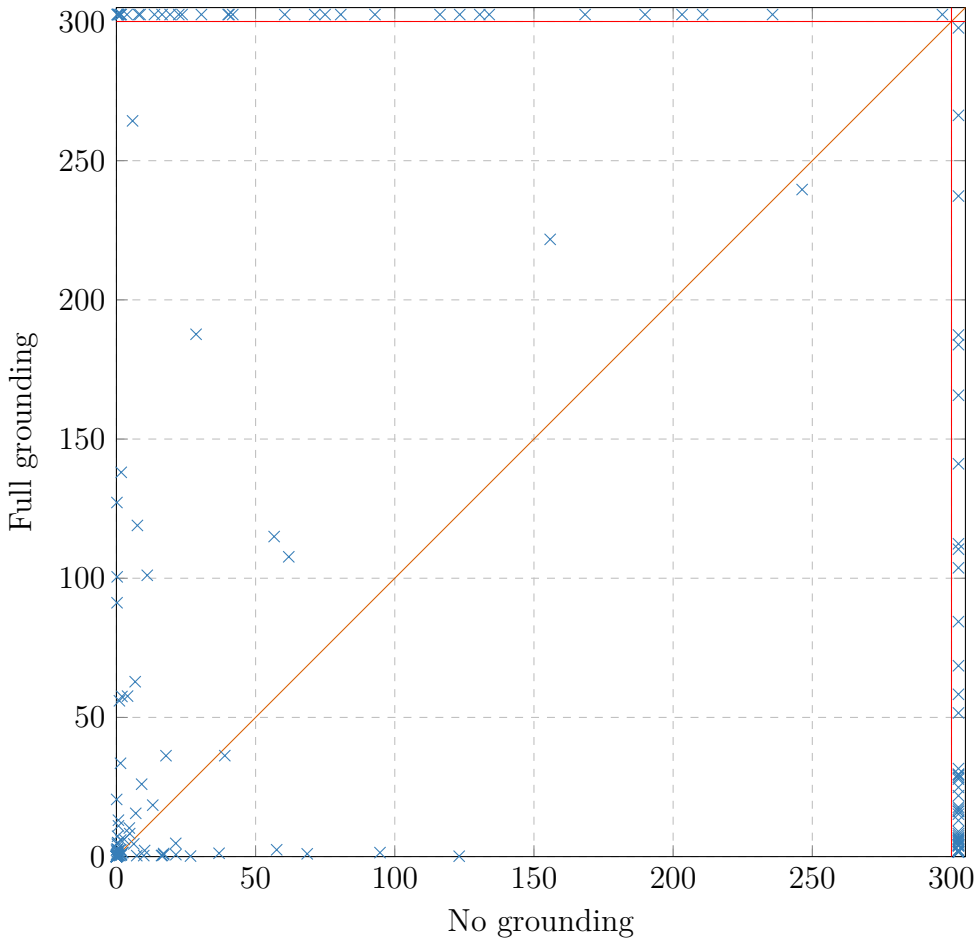


Figure 5.10.: Comparison of the time to solve each task with no grounding versus full grounding using eager pruning and the *min ground* parameter selection. Marks beyond the red lines indicate that the corresponding task could not be solved within the time limit.

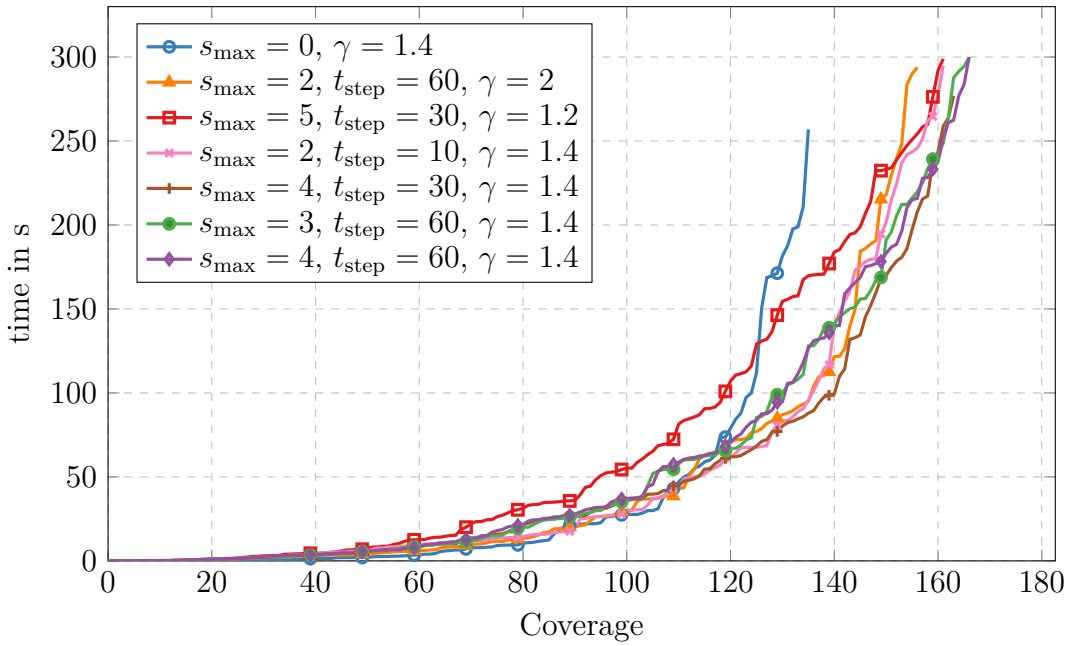


Figure 5.11.: The coverage of the smallest encoding planner for some combinations of values for $s_{\max}$, t_{step} and γ.

5.4.4. Step Scheduling

The three tuning parameters γ, $s_{\max}$ and t_{step} controlling the step scheduling are intuitively strongly interdependent. On the one hand, if a high horizon is needed to find a plan, either the step factor must be high or many unsolvable steps have to be skipped. On the other hand, if a tasks requires only a low horizon but is hard to solve, it is disadvantageous to skip steps early. We experiment with skipping 2 to 5 steps, step timeouts 10 s, 20 s, 30 s and 60 s and step factors 1.2, 1.4, 2 and 3. The coverage for some of the combinations including the baseline without step skipping are shown as a cactus plot in Figure 5.11. The best configurations in terms of coverage are given in Table 5.8. All of them use a step factor of $\gamma = 1.4$, which is therefore used by all our planners. For the smallest encoding planner and the parallel planner we use $s_{\max} = 4$ and $t_{\text{step}} = 60$. This configuration not only solves the most tasks, but also has the highest diversity of solved tasks across the domains. The interruptive planner uses considerably lower time limits for each solving attempt. We therefore use $s_{\max} = 4$ and $t_{\text{step}} = 30$ for the interruptive planner, as this configuration is well suited for shorter time limits according to Figure 5.11. Figure 5.12 compares the two step scheduling configurations as scatter plot and confirms that most tasks are solved faster with the configuration used by the interruptive planner despite fewer tasks are solved with that configuration.

Configuration			Coverage
$s_{\max}$	t_{step}	γ	
2	10	1.4	161
2	30	1.4	165
2	60	1.4	162
3	30	1.4	165
3	60	1.4	**166**
4	30	1.4	163
4	60	1.4	**166**
5	60	1.4	163

Table 5.8.: The best step scheduling configurations for the smallest encoding planner. The highest coverage is bold.

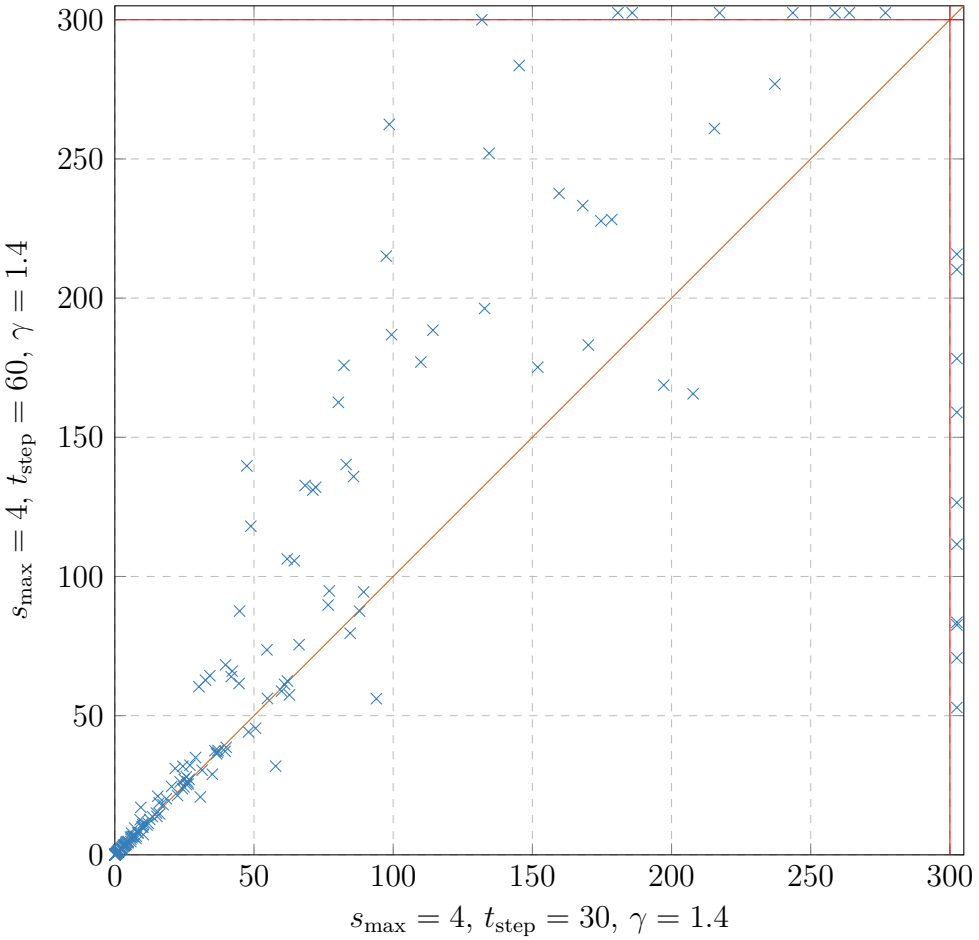


Figure 5.12.: Comparison of the two configurations used by our planners tested with the smallest encoding planner.

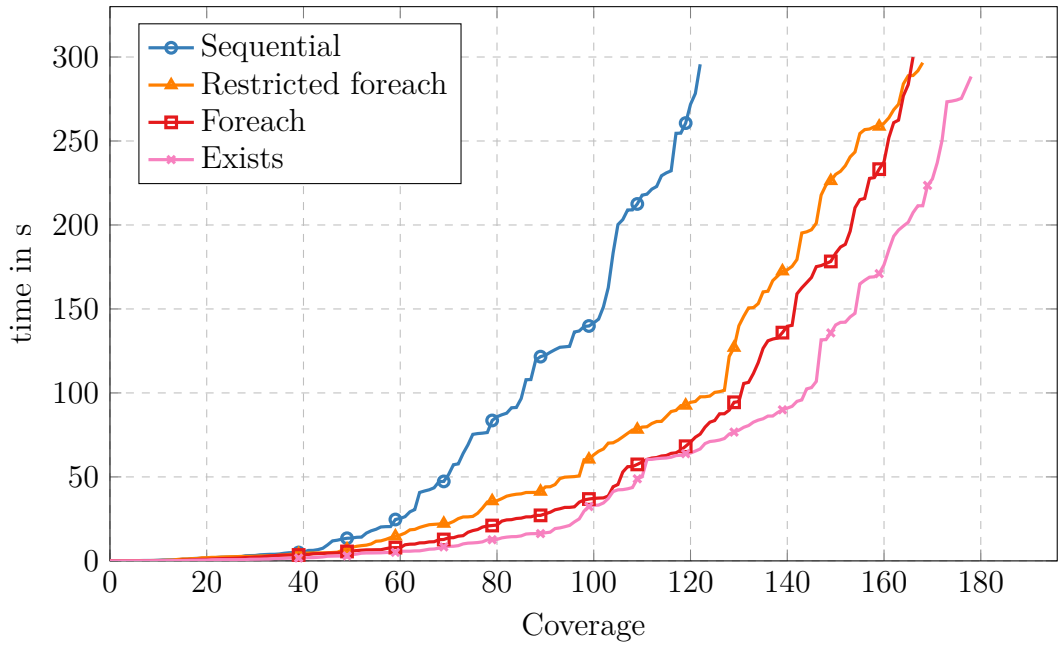


Figure 5.13.: The coverage using different encodings. The legend is sorted ascending after the allowed step parallelism.

5.4.5. Encoding

We have presented multiple encodings for partially instantiated representations in Section 4.3. To build the implication chain for the exists encoding, we need to order the operators. For our experiments we use the order in which our grounding routine outputs the operators. We first compare the performance of the encodings and then analyze the impact of the techniques to mitigate the combinatorial explosion due to the frame axioms (as discussed in Section 4.3).

Figure 5.13 shows that the exists encoding is clearly superior to the others, thus all our planners use that encoding. Even though the foreach encoding allows a superset of actions to be applied in one step compared to the actions allowed by the restricted foreach encoding, the latter surpasses the former eventually.

The coverage for varying DNF thresholds $d_{\max}$ with and without the option to imply the action from its parameters (see Equations 4.2 and 4.3) is shown in Figure 5.14. The techniques to handle the frame axioms are clearly required, as the graph for $d_{\max} = 0$ without the "imply action" option is not competitive with any other configuration. The plot suggests that parameters implying the corresponding action generally improves the performance. However, with $d_{\max} = 4$, the difference is negligible. The configuration using $d_{\max} = 8$ and the option to imply the action from its parameters enabled is the most successful and therefore selected for all our planners.

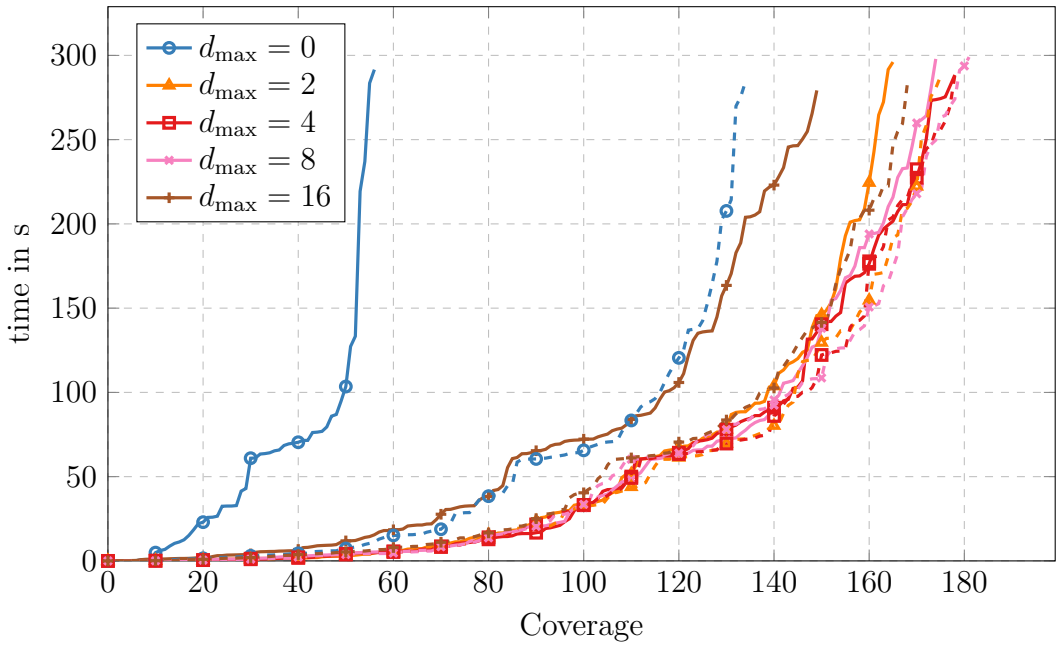


Figure 5.14.: Comparison for different values for $d_{\max}$. Solid and dashed plots of the same color use the same value for $d_{\max}$. The latter use the "imply action" option, which means that Equations 4.2 and 4.3 are used in the encoding.

5.4.6. SAT Solver

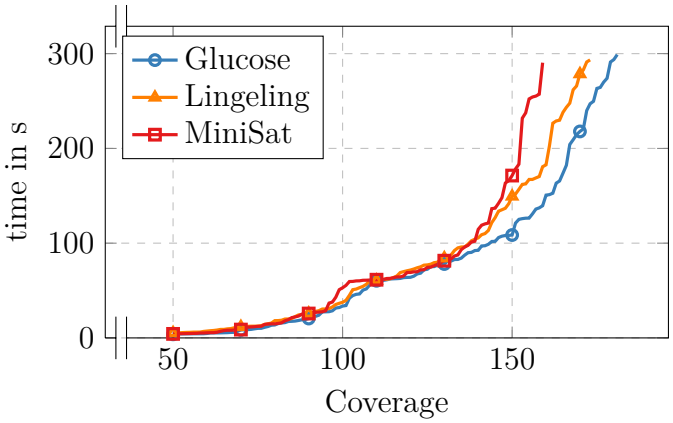
All our planners can be used with every SAT solver that implements the IPASIR interface. We compare the coverage of the smallest encoding planner using different SAT solvers. Besides Glucose, we also test *Lingeling* [Bie13] version bcj and *MiniSat* [ES03] version 2.2.0. Figure 5.15 shows that Glucose is the superior SAT solver for our application, solving 8 tasks more than the runner-up, Lingeling.

5.4.7. Smallest Encoding Planner

Table 5.9 shows the average selected groundness by domain with granularity 3 (the representations are sampled with groundness 0, 0.33, 0.67 and 1). Either groundness 0 or 1 yields the smallest encoding for almost all tasks of one domain consistently, with exception of the "organic-synthesis" domain. We therefore set the sampling granularity g to 3.

5.4.8. Interruptive Planner

The granularity (i.e., the number of times we interrupt the current solving attempt and start a new one) is a decisive parameter of the interruptive planner. If set too low, we might subsample the groundness and only try to solve suboptimal representations.



SAT solver	Coverage
Glucose	**181**
Lingeling	173
MiniSat	159

Figure 5.15.: Performance of the smallest encoding planner using all three SAT solvers.

Domain	Groundness
barman	0.00
childsnack	0.00
data-network	1.00
floortile	1.00
ged	0.03
hiking	0.00
openstacks	1.00
organic-synthesis	0.35
snake	0.00
termes	0.03
tetris	1.00
thoughtful	0.00
transport	0.00
visitall	1.00

Table 5.9.: The arithmetic mean of the groundness selected by the smallest encoding planner by domain.

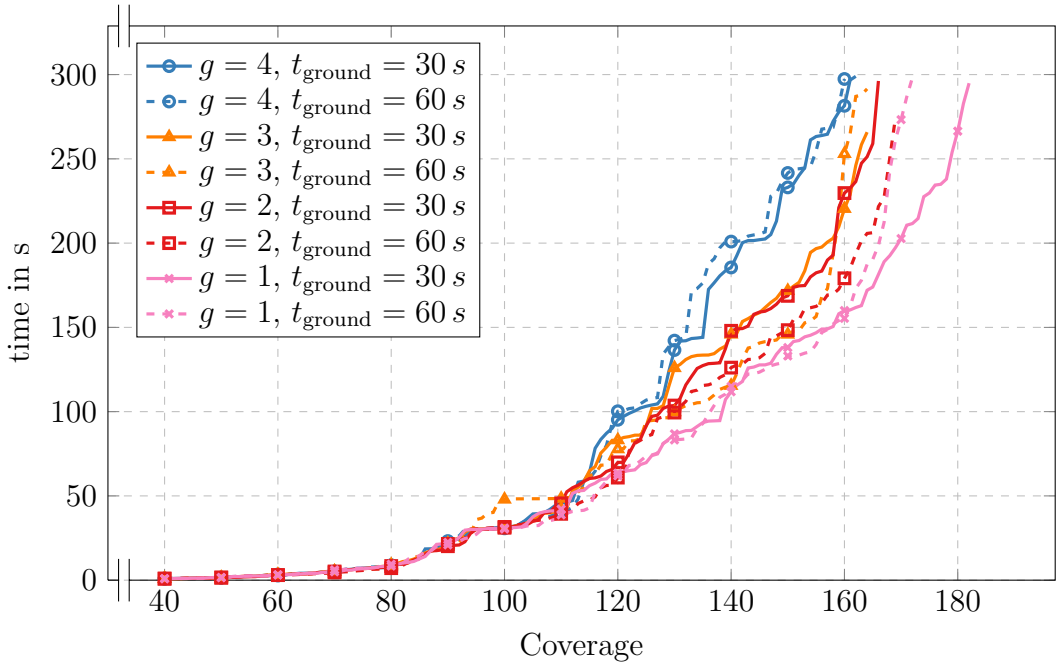


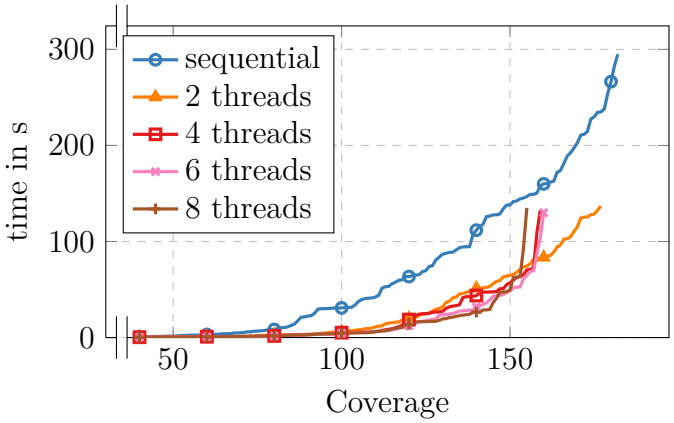
Figure 5.16.: The coverage of the interruptive planner using different values for granularity g and grounding timeout t_{ground}.

Oversampling on the other hand reduces the time for each solving attempt. The peaks at groundness 0 and 1 in Figure 5.9 suggest that the interruptive planner has the best chances to solve tasks either with no grounding or with full grounding. The planner allocates t_{ground} seconds for each grounding iteration (see line 7 of the interruptive planner algorithm). The remaining time is split equally between the solving attempts. Thus, every solving attempt has a timeout of $\frac{300 - t_{\text{ground}} * g}{g+1}$ seconds with granularity g. Figure 5.16 shows the coverage for different granularities $g \in [1, 2, 3, 4]$ and t_{ground} of 30 s and 60 s. We use $g = 1$ and $t_{\text{ground}} = 30$ s, since this configuration yields the highest coverage.

5.4.9. Parallel Planner

The parallel planner does not impose further tuning parameters. The more threads we use to concurrently solve encodings for representations of different groundness, the more likely we find a representation that can efficiently be solved. However, this parallelization does not scale well in practise, since only few tasks are best solved with groundness other than 0 or 1. The parallel planner utilizes the parallel grounder as described in Section 4.4.3.

Figure 5.17a shows the coverage for 2, 4, 6 and 8 threads compared to the interruptive planner in its final configuration. The more threads we use, the faster the tasks get solved. However, the parallel planner does not solve more tasks than the interruptive planner. This is due to the memory limitations, as each thread



(a) The coverage of the parallel planner with various threads.

Threads	Time in s
1	5500.57
2	2064.21
4	1749.37
6	1453.57
8	1403.50

(b) The accumulated time to solve all tasks that are solved with every number of threads (154 tasks in total).

Figure 5.17.: The performance of the parallel planner with different threads. The plot for one thread is the interruptive planner in its final configuration.

holds one SAT solver instance, each of which requires increasing memory the longer the solving takes (the formulae only increases with higher steps). As expected, the increase in speedup using two threads is significantly higher than for any higher number of threads (see Table 5.17b).

5.4.10. Summary

We briefly summarize the final parameter configuration for all our planners in Table 5.10. Figure 5.18 compares their performance with a cactus plot. The smallest encoding planner and the interruptive planner behave very similarly, while the parallel planner is generally faster but solves slightly fewer tasks.

	SE	I	P
Grounding			
Pruning policy		Eager	
Parameter selection		Min ground	
Grounding timeout	60 s	30 s	60 s
SAT			
Encoding		Exists	
DNF threshold $d_{\max}$		8	
Parameter implies action		Yes	
SAT solver		Glucose	
Step scheduling			
Step factor γ		1.4	
Max step skip $s_{\max}$		4	
Step timeout t_{step}	60 s	30 s	60 s
Granularity/Threads	3	1	2

Table 5.10.: Final configuration for the smallest encoding planner (**SE**), the interruptive planner (**I**) and the parallel planner (**P**). The grounding timeout for the interruptive planner and the parallel planner are per iteration.

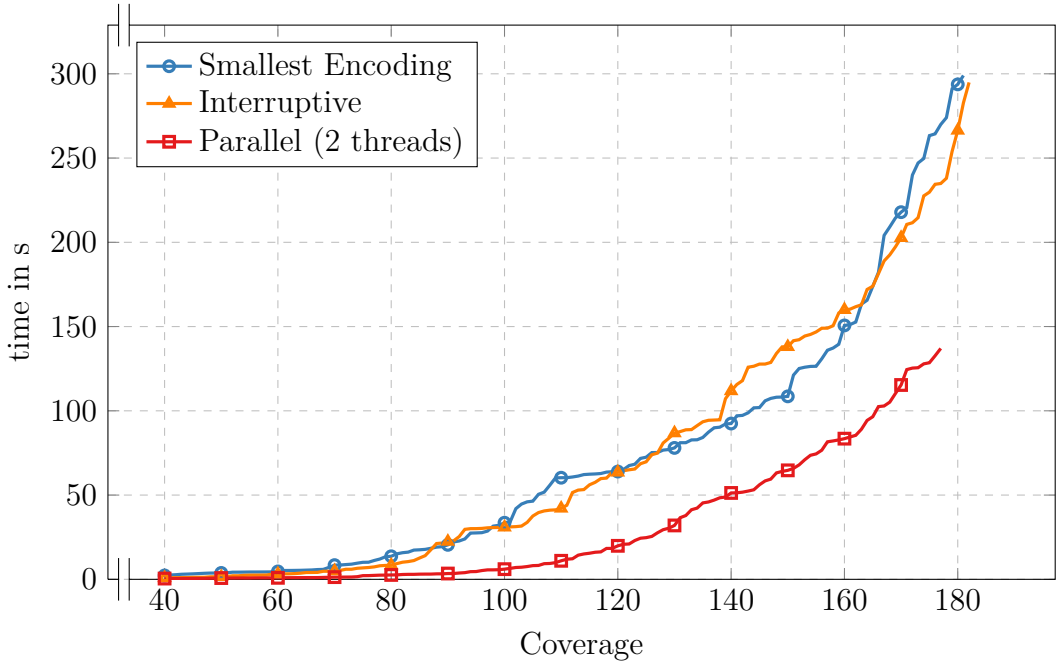


Figure 5.18.: Comparison of our planners in their final configuration.

5.5. Comparison

We compare our planners to the following planners on both the IPC set we used for parameter tuning as well as the Sparkle set.

LAMA 2011 is a configuration of the Fast Downward planning system [RWH11]. It placed first in the sequential satisficing track of the *International Planning Competition* 2011. The planner is based off of a state-space search. We follow the remarks from Rintanen in [Rin11] and set the time limit for LAMA's invariant synthesis to 60 s to adjust the planner to our time limit of 5 min.

Fast Forward (FF) [HN01] participated very successfully in the *International Planning Competition* 2000 and 2002. Fast Forward is a state-space planner that popularized the relaxed planning graph heuristic.

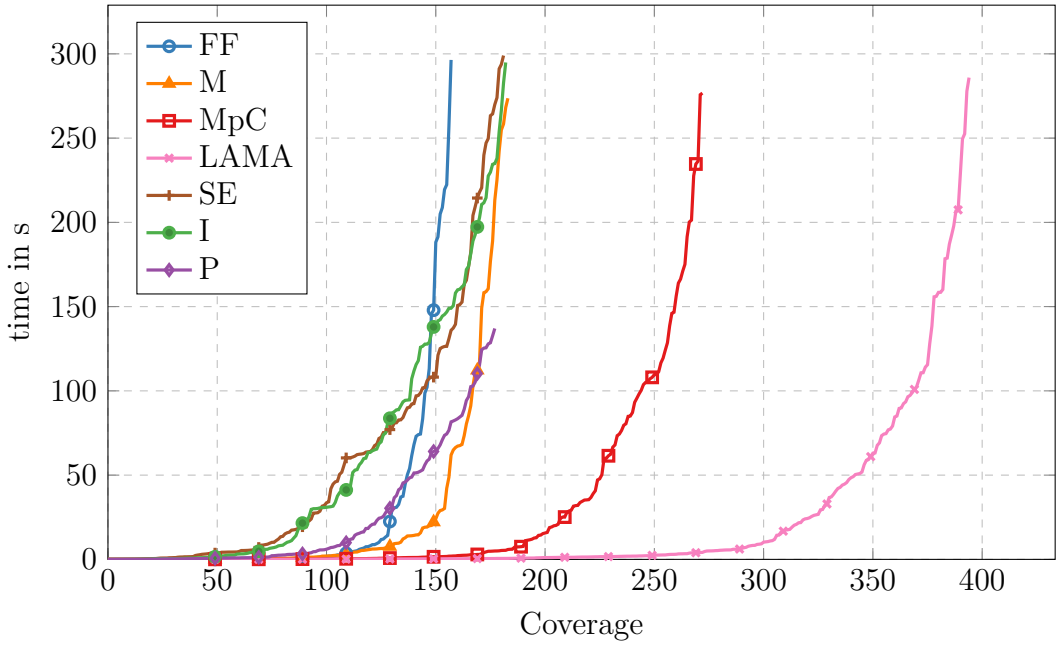
Madagascar [Rin14] is a SAT based planner with hand-crafted SAT solver. We include the versions **M** and **MpC** in our comparison, which placed third and second in the agile track of the *International Planning Competition* 2014, respectively.

The coverage for each planner is given domain-wise in Table 5.11. The results are visualized in Figure 5.19a and Figure 5.19b for the IPC set and the Sparkle set, respectively. We cannot compete with LAMA 2011 with regards to the number of solved tasks on the IPC set, as it solves more than twice the tasks we can solve. However, we outperform FF and are on par with version M of Madagascar. There are multiple reasons why it is difficult for our planner to compete with LAMA and comparable planners. Rintanen states that "The planning competition benchmarks are in general quite favorable to planners that use explicit state space search [...], in comparison to other types of planning problems"[Rinb]. Also, the planners we developed are in a state that is not as optimized as today's state-of-the-art planners yet. Madagascar is currently the state of the art in SAT based planning. Nevertheless, the approaches we explored have proven to be beneficial on many domains in comparison to the other planners. All our planners are able to solve the most tasks of the domain "childsnack" compared to the competition. On the domain "hiking" we significantly outperform both Madagascar variants by solving more than twice as many tasks, while still being subpar in comparison to LAMA. On the IPC set, we perform worst on the domains "openstacks", "transport" and "visitall".

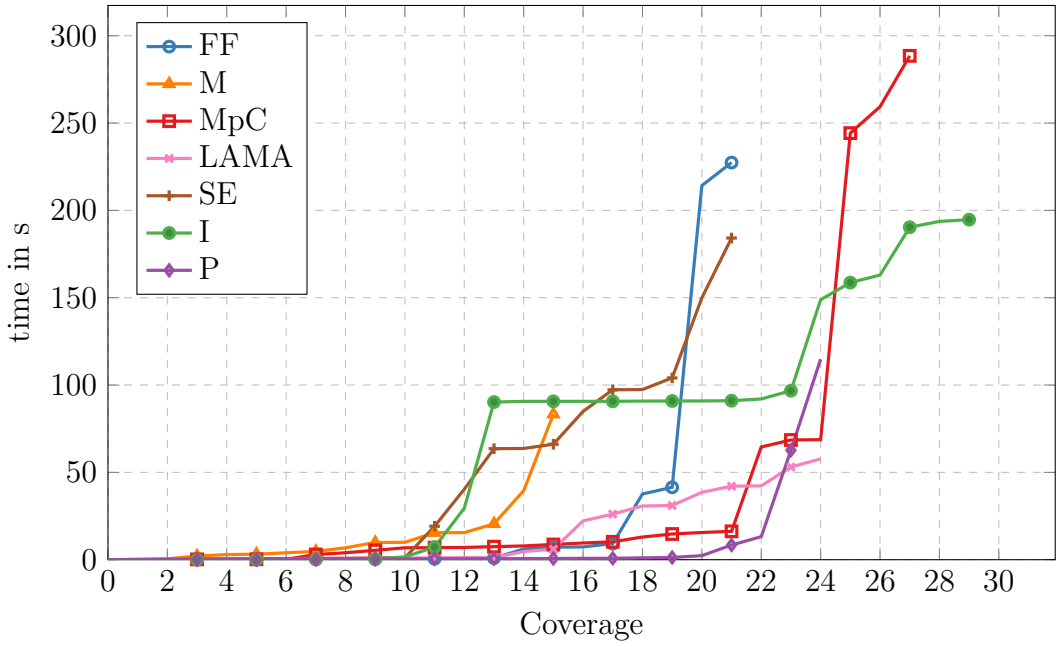
While our planners perform quite similar on most domains, the discrepancy among them is the highest on the domain "thoughtful". While both the parallel and the interruptive planner solve 5 tasks, the smallest encoding planner cannot solve a single task of that domain. The plans for tasks of that domain are comparably long. The smallest encoding planner chooses to encode the representation without grounding (see Table 5.9), but the higher step parallelism of the ground representation is needed to solve these tasks.

Domain	Planner						
	LAMA	FF	M	MpC	SE	I	P
barman (34)	**33**	6	4	18	4	2	3
childsnack (30)	15	3	27	15	**30**	29	29
data-network (40)	**31**	24	19	23	22	21	21
floortile (30)	9	1	**30**	**30**	22	22	15
ged (40)	**38**	24	20	34	20	20	20
hiking (40)	**37**	21	12	14	29	29	29
openstacks (40)	**40**	3	10	26	3	3	0
organic-synthesis (40)	**24**	0	16	19	19	14	19
snake (40)	17	14	2	**23**	6	5	6
termes (40)	**31**	7	1	1	1	0	0
tetris (37)	**34**	5	11	25	12	19	18
thoughtful (20)	**15**	12	5	5	0	5	5
transport (37)	**30**	19	14	19	8	8	9
visitall (40)	**40**	18	12	20	5	5	3
Sum (508)	**394**	157	183	272	181	182	177
agricola (10)	1	0	0	0	**2**	**2**	**2**
chairGame (10)	3	7	2	3	5	**10**	**10**
parking (10)	**10**	4	0	4	0	0	0
pipegrid (10)	0	0	**10**	**10**	**10**	**10**	**10**
pizza (10)	0	0	0	0	0	0	0
utc-distribution (10)	**10**	**10**	3	**10**	4	7	2
Sum (60)	24	21	15	27	21	**29**	24

Table 5.11.: Comparison of the smallest encoding planner (**SE**), the interruptive planner (**I**), the parallel planner (**P**) with 2 threads to the competition. The upper part is the domain set we used to tune our planners, the lower part are domains from the Sparkle Challenge. The number in parentheses is the number of tasks of that domain.



(a) Comparison of our planners to the competition on the IPC set.



(b) Comparison of our planners to the competition on the Sparkle set.

The picture is different on the tasks of the Sparkle set. Here, our interruptive planner solves the most tasks among all planners. The main contributor to the solved tasks compared to the other planners is the domain "chairgame". Tasks of that domain are very hard to ground. In fact, LAMA is only able to ground three out of ten tasks. No planner (including ours) can ground all tasks of this domain. However, all tasks of that domain are very easy to solve (the interruptive planner requires less than 2 s in total for each task) without grounding. The reason why the smallest encoding planner fails to solve all ten tasks is that the memory resources are exhausted during the search of the smallest encoding.

The domain "pipegrid" seems to be hard for both state-space searches but easy for all planners that utilize SAT solvers, as both Madagascar variants as well as all our planners solve all ten tasks. The domain "pizza" however is seemingly hard for all planners, as no one is able to solve a single of its tasks.

In general, the performance of our planners is complementary to the others on some domains and to LAMA specifically, which confirms the value of our approach. Most notably in that regard are "barman", "openstacks", "termes", "visitall", "chairgame" and "parking", where either LAMA or one of our planners performs significantly better than all other planners. Although our planners are not competitive with LAMA on the IPC set, they can be of value when used in a planner portfolio. Also, the presented techniques may be optimized further into more sophisticated planners to cover a wider range of domains.

6. Discussion

6.1. Conclusion

In this thesis we motivated the problem of automated planning and presented previous work on this field related to ours. We proposed novel approaches towards planning by incorporating SAT based planning into the grounding process. We therefore introduced the notion of *groundness* and heuristically determined advantageous representations to use for solving. We developed and thoroughly analyzed a hand-tailored grounding routine. After fine-tuning implementations of our planners, we compared them with their best configurations to state-of-the-art competition. While we cannot match the performance of the best state-space searches on the diverse IPC set, our planner complements the competition very well. Various planning tasks in our benchmark are only solvable by our planner. As our implementation is only a proof-of-concept, these results indicate that further investigations in our approaches are worthwhile. We believe that our approach can be especially valuable in planner portfolios.

Also, we briefly explored possibilities to parallelize our implementation with multiple threads and shared memory. Our experiments regarding parallelization have not been exhaustive but show that the grounding process can be sped up for most hard to ground tasks. Parallel SAT solving can also decrease the planning time, however the potential to parallelize our approach in the way we presented it is limited. Using two threads can generally lead to significant speedups, provided that enough memory is available.

6.2. Future work

This thesis covers many aspects of the pipeline from parsing and preprocessing the planning tasks to solving them with SAT solvers. Naturally, there is a lot of room for improvement on most of these parts. the most promising topics to work on in the future from the authors' perspective are given below.

PDDL We currently only support a subset of the PDDL specification. While this subset is sufficient to describe a variety of interesting domains, more advanced features such as conditional effects and axioms would enable our planners to compete on a broader spectrum of problems and improve their comparability.

Grounding The operator selection method we use is rather basic. Heuristics could be employed to select favourable operators first. Further, one could prune operators with a relaxed reachability analysis, similar to the grounder used by Fast Downward.

Encodings The exists encoding is the most successful among the ones we experimented with. However, much more sophisticated encodings with better performance have been proposed (see [RHN06]). It certainly is worthwhile to test these encodings in the context of partially instantiated representations.

Step scheduling The step scheduling measures we took to improve our planners' performance are not as sophisticated as the ones employed by Madagascar. Combining our approach with Madagascar's step scheduling mechanisms is very promising.

Parallelization The parallelization of our algorithm still has a lot of potential. We only parallelized on the dimension of groundness. As Rintanen[Rin14] showed, solving formulae for multiple steps at once can also be very efficient and generally scales better than the approach presented in this thesis. Finally, parallel SAT solvers have been proposed. We could also speed up the actual SAT solving by utilizing parallel SAT solvers.

A. Planning Tasks

We used a subset of the planning tasks of recent planning competitions for our experiments. The list of planning tasks by domain are given in the tables below sorted by the respective planning competition. The lists give the number of planning tasks for each domain. As some tasks occurred multiple times among the sources, we removed all duplicates. Also, we deemed domains unsuitable for our experiments if they make use of PDDL features we do not support or if they are already given in a ground representation.

A.1. IPC 2014

Detailed descriptions for the domains as well as download links can be found at `https://helios.hud.ac.uk/scommv/IPC-14/domains.html`

Domain	#Tasks
barman	34
childsnack	30
floortile	30
ged	40
hiking	40
openstacks	40
tetris	37
thoughtful	20
transport	37
visitall	40

Total number of tasks: 348

A.2. IPC 2018

The repository containing the planning tasks is located at `https://bitbucket.org/ipc2018-classical/domains/`

Domain	#Tasks
data-network	40
organic-synthesis	40
snake	40
termes	40

Total number of tasks: 160

A.3. Sparkle Planning Challenge 2019

The list of testing domains and download links are provided at `http://ada.liacs.nl/events/sparkle-planning-19/benchmarks.html`

Domain	#Tasks
agricola	10
chairgame	10
pipegrid	10
parking	10
utc-distribution	10
pizza	10

Total number of tasks: 60

Bibliography

[AS09] Gilles Audemard and Laurent Simon. "Glucose: a solver that predicts learnt clauses quality". In: *SAT Competition* (2009), pp. 7–8.

[AS14] Gilles Audemard and Laurent Simon. "Lazy clause exchange policy for parallel SAT solvers". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2014, pp. 197–205.

[Bal+16] Tomáš Balyo et al. "SAT race 2015". In: *Artificial Intelligence* 241 (2016), pp. 45–65.

[Bal13] Tomáš Balyo. "Relaxing the relaxed exist-step parallel planning semantics". In: *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*. IEEE. 2013, pp. 865–871.

[BF95] Avrim L Blum and Merrick L Furst. *Fast planning through planning graph analysis.* Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1995.

[Bie13] Armin Biere. "Lingeling, Plingeling and Treengeling entering the SAT competition 2013". In: *Proceedings of SAT competition* 2013 (2013), p. 1.

[Bre+05] John L Bresina et al. "Activity Planning for the Mars Exploration Rovers." In: *ICAPS*. 2005, pp. 40–49.

[BS] Tomáš Balyo and Dominik Schreiber. *Automated Planning and Scheduling*. URL: `https://baldur.iti.kit.edu/plan/` (visited on 03/31/2020).

[BSS15] Tomáš Balyo, Peter Sanders, and Carsten Sinz. "Hordesat: A massively parallel portfolio SAT solver". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2015, pp. 156–172.

[Byl91] Tom Bylander. "Complexity Results for Planning." In: *IJCAI*. Vol. 10. 1991, pp. 274–279.

[Coo71] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. URL: `https://doi.org/10.1145/800157.805047`.

[DNK97] Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. "Encoding planning problems in nonmonotonic logic programs". In: *European Conference on Planning*. Springer. 1997, pp. 169–181.

[EH99] Stefan Edelkamp and Malte Helmert. "Exhibiting knowledge in planning problems to minimize state encoding length". In: *European Conference on Planning*. Springer. 1999, pp. 135–147.

[ES03] Niklas Eén and Niklas Sörensson. "An extensible SAT-solver". In: *International conference on theory and applications of satisfiability testing*. Springer. 2003, pp. 502–518.

[FBS19] Nils Froleyks, Tomáš Balyo, and Dominik Schreiber. "PASAR Entering the Sparkle Planning Challenge 2019". In: (2019).

[FN71] Richard E Fikes and Nils J Nilsson. "STRIPS: A new approach to the application of theorem proving to problem solving". In: *Artificial intelligence* 2.3-4 (1971), pp. 189–208.

[Gna+19] Daniel Gnad et al. "Learning how to ground a plan–partial grounding in classical planning". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 7602–7609.

[GNT04] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. The Morgan Kaufmann Series in Artificial Intelligence. Elsevier Science, 2004. ISBN: 9780080490519.

[Hel09] Malte Helmert. "Concise finite-domain representations for PDDL planning tasks". In: *Artificial Intelligence* 173.5-6 (2009), pp. 503–535.

[HLF04] Richard Howey, Derek Long, and Maria Fox. "VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL". In: *16th IEEE International Conference on Tools with Artificial Intelligence*. IEEE. 2004, pp. 294–301.

[HN01] Jörg Hoffmann and Bernhard Nebel. "The FF planning system: Fast plan generation through heuristic search". In: *Journal of Artificial Intelligence Research* 14 (2001), pp. 253–302.

[HRK11] Malte Helmert, Gabriele Röger, and Erez Karpas. "Fast downward stone soup: A baseline for building planner portfolios". In: *ICAPS 2011 Workshop on Planning and Learning*. Citeseer. 2011, pp. 28–35.

[Kat+18] Michael Katz et al. "Delfi: Online planner selection for cost-optimal planning". In: *IPC-9 planner abstracts* (2018), pp. 57–64.

[KMS96] Henry Kautz, David McAllester, and Bart Selman. "Encoding plans in propositional logic". In: *KR* 96 (1996), pp. 374–384.

[KS+92] Henry A Kautz, Bart Selman, et al. "Planning as Satisfiability." In: *ECAI*. Vol. 92. Citeseer. 1992, pp. 359–363.

[KS98] Henry Kautz and Bart Selman. "BLACKBOX: A new approach to the application of theorem proving to problem solving". In: *AIPS98 Workshop on Planning as Combinatorial Search*. Vol. 58260. 1998, pp. 58–60.

[LK05] S. G. Loizou and K. J. Kyriakopoulos. "Automated Planning of Motion Tasks for Multi-Robot Systems". In: *Proceedings of the 44th IEEE Conference on Decision and Control*. Dec. 2005, pp. 78–83.

[MS96] J. P. Marques Silva and K. A. Sakallah. "GRASP-A new search algorithm for satisfiability". In: *Proceedings of International Conference on Computer Aided Design*. Nov. 1996, pp. 220–227. DOI: 10.1109/ICCAD.1996.569607.

[RF14] Bram Ridder and Maria Fox. "Heuristic evaluation based on lifted relaxed planning graphs". In: *Twenty-Fourth International Conference on Automated Planning and Scheduling*. 2014.

[RHN06] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. "Planning as satisfiability: parallel plans and algorithms for plan search". In: *Artificial Intelligence* 170.12 (2006), pp. 1031–1080. ISSN: 0004-3702. URL: http://www.sciencedirect.com/science/article/pii/S0004370206000774.

[Rina] Rintanen. *Madagascar*. URL: https://research.ics.aalto.fi/software/sat/madagascar/ (visited on 03/31/2020).

[Rinb] Rintanen. *Planning as Satisfiability: state of the art*. URL: https://users.aalto.fi/~rintanj1/satplan.html (visited on 03/24/2020).

[Rin11] Jussi Rintanen. "Planning with specialized SAT solvers". In: *Twenty-Fifth AAAI Conference on Artificial Intelligence*. 2011.

[Rin14] Jussi Rintanen. "Madagascar: Scalable planning with sat". In: *Proceedings of the 8th International Planning Competition (IPC-2014)* 21 (2014).

[Rob+09] Nathan Robinson et al. "SAT-based parallel planning using a split representation of actions". In: *Nineteenth International Conference on Automated Planning and Scheduling*. 2009.

[RWH11] Silvia Richter, Matthias Westphal, and Malte Helmert. "LAMA 2008 and 2011". In: *International Planning Competition*. 2011, pp. 117–124.

[Sav70] Walter J Savitch. "Relationships between nondeterministic and deterministic tape complexities". In: *Journal of computer and system sciences* 4.2 (1970), pp. 177–192.

[Sei+17] Jendrik Seipp et al. "Downward Lab". In: *Google Scholar Google Scholar Cross Ref Cross Ref* (2017).

[Tse83] Grigori S Tseitin. "On the complexity of derivation in propositional calculus". In: *Automation of reasoning*. Springer, 1983, pp. 466–483.

[WR07] Martin Wehrle and Jussi Rintanen. "Planning as Satisfiability with Relaxed Exists-Step Plans". In: *Australasian Joint Conference on Artificial Intelligence*. Springer. 2007, pp. 244–253.

[Xu+12] L. D. Xu et al. “AutoAssem: An Automated Assembly Planning System for Complex Products”. In: *IEEE Transactions on Industrial Informatics* 8.3 (Aug. 2012), pp. 669–678. ISSN: 1941-0050.