GENERALIZING LEARNED KNOWLEDGE IN ANSWER SET SOLVING

PATRICK LÜHNE

GENERALIZING LEARNED KNOWLEDGE IN ANSWER SET SOLVING

PATRICK LÜHNE

University of Potsdam

Knowledge Processing and
Information Systems Group

Hasso Plattner Institute

Operating Systems and
Middleware Group

Advisers

Prof. Dr. Torsten Schaub

Dipl.-Inf. Roland Kaminski

Javier Romero Davila, M. Sc.

Reviewer

Prof. Dr. Andreas Polze

Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

September 2015

CONTENTS

ABSTRACT

Constraint learning is one of the major advances of answer set programming (ASP) in the last decades. Unlike pure backtracking, modern answer set solvers analyze encountered conflicts to acquire new knowledge. However, learned knowledge is only valid for the currently solved problem instance. This thesis presents multiple ways to *reuse* learned knowledge. A study in the first part of this thesis demonstrates that solvers benefit from directly reusing previously learned knowledge when solving the same instance again. The second part of this thesis presents an unassisted algorithm that generalizes learned knowledge given an instance of a planning problem. To this end, the scope of learned properties is extended over the temporal domain. The derived properties are then validated with a proof by induction. As an alternative, this thesis introduces a novel, simplified proof method that renders knowledge generalization instance-independent. This proof method allows generalized knowledge to be transferred to future instances of similar structure. A pilot study shows that instances are indeed solved faster when reusing generalized knowledge of a previous instance in this way.

ZUSAMMENFASSUNG

Constraint Learning ist einer der wesentlichen Fortschritte der Antwortmengenprogrammierung (ASP) in den letzten Jahrzehnten. Im Gegensatz zu Backtracking analysieren moderne Solver auftretende Konflikte, um daraus neues Wissen zu gewinnen. Dieses gelernte Wissen ist allerdings nur in Bezug auf die gerade gelöste Probleminstanz gültig. Diese Arbeit stellt mehrere Wege vor, gelerntes Wissen *wiederzuverwenden*. Eine Studie im ersten Teil dieser Arbeit zeigt, dass Solver vom direkten Wiederverwenden gelernten Wissens profitieren, wenn die selbe Instanz erneut gelöst wird. Der zweite Teil dieser Arbeit stellt einen Algorithmus vor, der Wissen selbstständig verallgemeinert, das bei Instanzen von Planungsproblemen gelernt wurde. Zu diesem Zweck wird die Gültigkeit der gelernten Eigenschaften über die Zeitdomäne erweitert. Die abgeleiteten Eigenschaften werden dann mithilfe eines Induktionsbeweises validiert. Als Alternative stellt diese Arbeit eine neuartige, vereinfachte Beweismethode vor, die zur instanzunabhängigen Verallgemeinerung von Wissen führt. Diese Beweismethode ermöglicht es, verallgemeinertes Wissen auf zukünftige Instanzen ähnlicher Struktur zu übertragen. Eine Leitstudie zeigt, dass Instanzen tatsächlich schneller gelöst werden, wenn auf diese Weise verallgemeinertes Wissen wiederverwendet wird, das von einer früheren Instanz stammt.

1

INTRODUCTION

Answer set programming (ASP) is a paradigm for declarative program solving [2]. Instead of specifying the control flow that leads to a solution, the *problem's logic itself* is described and finding the solution is left to a dedicated *solver*. The idea is to express a problem in the syntax of a logic program in such a way that the solutions found by the solvers coincide with the solutions of the original problem. Commonly, logic programs are interpreted under the *stable model semantics* introduced by Gelfond and Lifschitz in 1988 [12].

The solving process employed by most answer set solvers is based on the classic *Davis–Putnam–Logemann–Loveland* algorithm (DPLL) [6, 5]. Many modern solvers implement an enhanced version of this algorithm called *conflict-driven clause learning* (CDCL) [3, 21]. This algorithm employs *learning*, one of the major advances of answer set solving in the past decades. Unlike the pure backtracking approach of DPLL, CDCL attempts to analyze encountered conflicts and acquire new knowledge in the process. Learned knowledge consists of *conflict constraints*, which are added to the problem

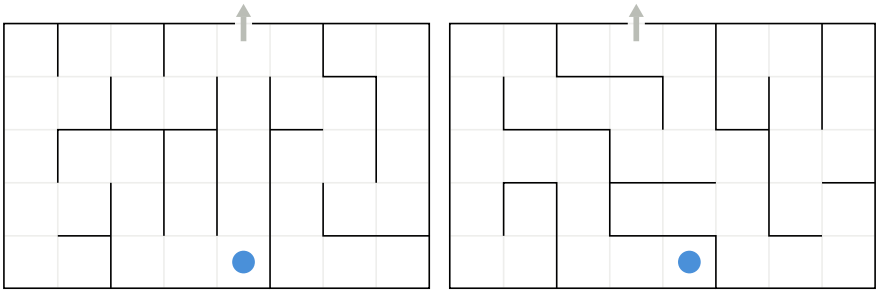


Figure 1.1: Two different instances of the maze problem.

specification to prune the remaining search space. This strategy often leads to considerably reduced solving times compared to simple backtracking.

However, knowledge learned in this way is only valid with respect to the currently solved logic program. Thus, it cannot be used again to solve future programs, even if they share many properties. For example, consider the two instances of the maze problem in Figure 1.1, which consists of finding the shortest way out of a labyrinth. When solving the left instance, the solver might learn that the blue token may not be moved west in the first step to find the shortest way. This assumption is not true for the instance on the right—there, the exit can *only* be reached by moving west first. Nevertheless, certain pieces of learned knowledge hold across all instances of a problem. For instance, shortest solutions to maze instances may never contain a move west in the first step and a subsequent move east—that is, simple loops.

This thesis explores ways to *reuse* learned knowledge, with the ultimate objective to overcome the issue that learned knowledge is bound to specific instances. The first part of this thesis investigates how modern answer set solvers benefit from reusing learned knowledge *directly*. For this purpose, learned knowledge is recorded while solving an instance, and then the instance is solved again, reusing some parts of the learned knowledge.

The second part of this thesis introduces multiple methods for generalizing learned knowledge. The discussed methods split the problem of gen-

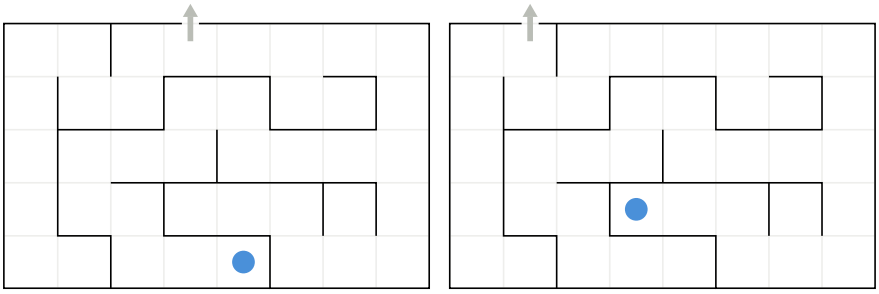


Figure 1.2: Two more instances of the maze problem that share the same domain, but differ in their initial states and goal conditions.

eralizing knowledge into two steps—generating candidates for generalized properties and validating them thereafter. With these techniques, an algorithm is designed that extracts and generalizes learned knowledge without assistance, called the *generalized knowledge feedback loop*. Finally, a configuration of this algorithm is derived that performs knowledge generalization in such a way that generalized properties may be transferred to *other instances* than they were learned from. To transfer knowledge, instances of the same problem may differ in their initial states and goal conditions, but they must share the same static environment (like the two instances in Figure 1.2).

STRUCTURE First, Chapter 2 recapitulates the theoretical background of this thesis. Chapter 3 then details the prerequisite for reusing knowledge—extracting it from the answer set solver. Building on this, Chapter 4 analyzes how solvers profit from reusing learned knowledge directly. In the second part of this thesis, Chapter 5 explores various methods for generalizing knowledge. These are then used in Chapter 6 to build and evaluate the generalized knowledge feedback loop. Chapter 7 concludes this thesis by reflecting on the contributions of this thesis and addresses related and future work.

2

BACKGROUND

This chapter covers the theoretic foundations of the research presented in this thesis. First, Section 2.1 describes the concepts of answer set programming and ASP's modeling language. Section 2.2 explains how answer set programs can be solved with the CDCL algorithm and learning. Planning problems, the *planning domain definition language* (PDDL), and *plasp* (a tool for converting PDDL to ASP facts) are discussed in Section 2.3. After the logic-related topics, Section 2.4 describes the geometric mean, which is used in statistical evaluations in this thesis.

2.1 ANSWER SET PROGRAMMING

This section revisits the basics of answer set programming. First, essential definitions are listed in Section 2.1.1. Section 2.1.2 describes the modeling language of ASP, which is used to specify logic programs. Section 2.1.3 explains how answer set programs are represented in order to be solved later.

Notations used in this thesis stem from the textbook *Answer Set Solving in Practice* [9] by Gebser et al.

2.1.1 TERMINOLOGY

LOGIC DEFINITIONS The formal language for answer set programming has a signature consisting of symbols for constants, variables, predicates, and functions. **Constants** $c, d, \ldots$ are named objects in the given universe, while **first-order variables** $X, Y, \ldots$ range over domains of such objects. **Predicates** $p, q, \ldots$ map tuples of objects (arguments) to Boolean values (`true` and `false`). In contrast, **functions** $f, g, \ldots$ map their arguments to objects. The number of parameters of predicates and functions is called **arity**. Predicates with arity 0 are also referred to as **propositions**.

Terms t are defined recursively:

- All constants and variables are terms.
- $f(t_1, \ldots, t_n)$ is a term if f is an n-ary function $t_1, \ldots, t_n$ are terms.

Building on this, **atoms** $a, b, \ldots$ are defined as follows:

- $t_1 = t_2$ is an atom if t_1 and t_2 are terms.
- $p(t_1, \ldots, t_n)$ is an atom if p is a predicate applied to terms $t_1, \ldots, t_n$.

Terms and atoms that contain no variables are called **ground terms** and **ground atoms**, respectively.

A **literal** l is either an atom a or its negation. Commonly, two types of negation are distinguished. With **default negation**, $\sim a$ refers to the *absence* of information that a holds. In contrast, **classical negation** requires the complement $\overline{a}$ to be provably known.

Clauses $\delta, \varepsilon, \ldots$ are disjunctions $(l_1 \vee \ldots \vee l_n)$ of literals $l_1, \ldots, l_n$. Sometimes, clauses are identified with the set $\{l_1, \ldots, l_n\}$ instead.

ANSWER SET SOLVING DEFINITIONS A **normal logic program** P is a finite set of **normal rules**. A normal rule r has the form

$$r: \ a_0 \leftarrow a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n$$

where a_0 is either `false` or a ground atom, $1 \leq m \leq n$, and $a_1, \ldots, a_n$ are ground atoms. a_0 is called the **head** of rule r, denoted by $head(r)$. The rule's **body** is $body(r) = \{a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n\}$. The body holds if all positive atoms $\{a_1, \ldots, a_m\}$ are provably `true` and the negative ones $\{a_{m+1}, \ldots, a_n\}$ are either `false` or unknown.

A rule r signifies that when its body holds, the head is derived to be `true`. Additionally, let $body(r)^+ = \{a_1, \ldots, a_m\}$ and $body(r)^- = \{a_{m+1}, \ldots, a_n\}$. A logic program P is **enriched** by adding new rules to P.

Facts are rules with an empty body, often written without $\leftarrow$. Facts represent properties that hold unconditionally. **Integrity constraints** are rules whose head is `false`, also written as:

$$r: \ \leftarrow a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n$$

Integrity constraints eliminate solutions of P that satisfy their body. Thus, additional integrity constraints can never introduce new solutions.

The **reduct** P^A of a program P with respect to a set of ground atoms A is defined in the following way:

$$P^A = \{head(r) \leftarrow body(r)^+ \mid r \in P, body(r)^- \cap A = \varnothing\}$$

The reduct P^A can be obtained by first removing rules entirely if they contain negative literals ${\sim}a$ with $a \in A$ in their body. Then, all negative body literals ${\sim}a$ in the remaining rules are eliminated.

A **model** of the program P is a set of ground atoms A such that $head(r) \in A$ for every rule r that satisfies $body(r)^+ \subseteq A$ and $body(r)^- \cap A = \varnothing$. **Stable models** (or *answer sets* or *solutions*) of P are minimal models of P^A with respect to the $\subseteq$ relation.

Answer set solving is the process of finding stable models of a logic program P. Logic programs are often specified in ASP's modeling language. Applications that automatically perform the search for stable models are called **answer set solvers**, or shortly *solvers*.

In the process of answer set solving, **propositional variables** (atoms and bodies) are **assigned** Boolean values. Formally, the domain of assignments of a program P is $domain(P) = atom(P) \cup body(P)$. For simplicity, the term *literal* is extended to refer to propositional variables (not only atoms) and their negations. Assigning a propositional variable v a truth value is denoted by its literals v (for `true`) or $\overline{v}$ (for `false`).

An **ordered assignment** A over $domain(P)$ is a sequence $(l_1, \ldots, l_n)$ of literals $l_i \in \{v, \overline{v} \mid v \in domain(P)\}$. The literals l_i stand for assignments of propositional variables, and l_n is the most recent one. An assignment is a **complete assignment** if all ground atoms are uniquely assigned `true` or `false`. All other assignments are called **partial assignments**. The propositional variables in A that are assigned `true` can be accessed with $A^T = \{v \in domain(P) \mid v \in A\}$ and the `false` ones with $A^F = \{v \in domain(P) \mid \overline{v} \in A\}$.

2.1.2 THE MODELING LANGUAGE OF ASP

ASP provides a declarative language to model logic problems under the stable model semantics. Solution of a program are later found with the help of an answer set solver. The modeling language stems from the input language that the grounding tool *Lparse* accepts [29].

In the modeling language of ASP, constants are numbers or identifiers starting with a lowercase letter, such as `10`, `p`, and `holds`. Variables start with a capital letter—for instance, `X` or `Robot`. Atoms are written like `identifier(arg1, arg2)`, where the identifier starts with a lowercase letter, and the arguments may be either constants or variables. Other examples are `color(C)`, `color(red)`, and `parent(john, jill)`. Default negation is identified with the keyword `not`.

Listing 2.1: ASP problem instance of Hanoi Tower.

```
1 #const moves=15.
2
3 peg(a; b; c).
4 disk(1..4).
5
6 init_on(1..4, a).
7 goal_on(1..4, c).
```

Programs in ASP syntax consist of rules of the form

```
sibling(X, Y) :- parent(Z, X), parent(Z, Y).
```

where `:-` is the inference symbol. The part left of `:-` is the rule's head, while its body is on the right. This rule, for instance, means that if two people X and Y share a parent Z, both of them are siblings. The symbol `:-` is omitted for rules without a body (facts).

To write integrity constraints, the head is left empty:

```
:- not paperFinished, deadlineOver.
```

This integrity constraint expresses that it may never be the case that a paper is unfinished when the deadline is over.

As a simple example, consider the Hanoi Tower problem description by Gebser et al. [9] in Listing 2.1. With the additional `#const` directive, a constant `moves` is defined along with its default value 15. The semicolon and `..` are syntactic sugar and expand to the following facts:

```
peg(a). peg(b). peg(c).
disk(1). disk(2). disk(3). disk(4).
```

and so on. This program specifies the existence of three pegs (`a` to `c`) and four disks (`1` to `4`), which initially reside on peg `a`. The program also defines a goal—all disks should be on peg `c` within 15 moves. Which moves are

Listing 2.2: ASP problem encoding of Hanoi Tower.

```
1  % Generating part
2  1 {move(D, P, T) : disk(D) : peg(P)} 1 :- T = 1..moves.
3
4  % Defining part
5  move(D, T) :- move(D, _, T).
6  on(D, P, 0) :- init_on(D, P).
7  on(D, P, T) :- move(D, P, T).
8  on(D, P, T + 1) :- on(D, P, T), not move(D, T + 1), T < moves.
9  blocked(D - 1, P, T + 1) :- on(D, P, T), T < moves.
10 blocked(D - 1, P, T) :- blocked(D, P, T), disk(D).
11
12 % Testing part
13 :- move(D, P, T), blocked(D - 1, P, T).
14 :- move(D, T), on(D, P, T - 1), blocked(D, P, T).
15 :- not 1 {on(D, P, T)} 1, disk(D), T = 1..moves.
16
17 :- goal_on(D, P), not on(D, P, moves).
18
19 % Displaying part
20 #hide.
21 #show move/3.
```

allowed and how they are performed is not yet defined. This program only specifies the environment of a Hanoi Tower problem.

Listing 2.2 contains another program that defines the missing rules to solve this Hanoi Tower problem. The program is divided into several parts. The *generating* part uses a *choice rule* (line 2) to express that for every time step `T` within the number of allowed moves (starting with 1), exactly one disk must be moved to a peg. The head of this choice rule is a *cardinality constraint*, which is satisfied when the number of elements it captures lies within a lower and upper bound (left and right of the braces)—in this ex-

ample, both bounds are 1, which forces exactly one element to be chosen. Which disk is moved to which peg is not specified, but instead guessed (by expanding the rule to all possible disks and pegs using the predicates after the colons and allowing exacly one choice at a time). At this point, illegal moves might be generated, but this is checked later in the *testing* part.

The *defining* part introduces auxiliary predicates. Atom `move(D, T)` is derived whenever any peg moves at a time step `T`. Atom `on(D, P, T)` states that a disk `D` is on peg `P` at time `T`. The rule in line 6 ensures that each disk receives its initial position at time step 0, and because of line 7, moved pegs assume their new position. Line 8 introduces *inertia*, causing pegs to stay at their location if they are not moved. Lines 9 and 10 define when disks are blocked on a peg. This happens if a smaller disk is on the same peg.

In the *testing* part, all invalid solution candidates are eliminated with the help of integrity constraints. First, disks `D` may never be moved if the smaller one (`D - 1`), wherever located, is blocked at that moment (line 13). Line 14 excludes solutions where disks are moved even though they are blocked by smaller disks. Line 15 ensures that disks always reside on exactly one peg through another cardinality constraint. Finally, line 17 requires the goal to be met—if one of the disks is not on the desired peg after 15 moves, the solution candidate is discarded.

The directives in the *displaying* part (lines 20 and 21) instruct the grounder to only remember and show the names of the chosen actions. These commands exist primarily for the user's convenience.

The program in Listing 2.2 contains general rules of the Hanoi Tower problem, whereas Listing 2.1 is a concrete instantiation of the problem—it only contains facts. Splitting answer set programs into a **problem encoding** (such as Listing 2.2) and corresponding **problem instances** (such as Listing 2.1) is a common approach in answer set programming. Also, the pattern of splitting a program into parts similar to the above (generating, defining, testing, displaying) is found in many encodings.

Taking together the two programs in Listings 2.1 and 2.2, the given Hanoi

Tower instance can now be solved. For this purpose, the programs are given to a grounder, for instance *gringo*, and the output is then passed to an answer set solver such as *clasp*.[1] Running these tools yields the following output:

```
move(4, b, 1)  move(3, c, 2)  move(4, c, 3)  move(2, b, 4)
move(4, a, 5)  move(3, b, 6)  move(4, b, 7)  move(1, c, 8)
move(4, c, 9)  move(3, a, 10) move(4, a, 11) move(2, c, 12)
move(4, b, 13) move(3, c, 14) move(4, c, 15)
```

Indeed, the answer set solver has found a solution to the given Hanoi Tower problem instance within 15 moves.

2.1.3 FROM DEFINING TO SOLVING LOGIC PROGRAMS

The modeling language of ASP provides support for first-order variables, default negation, and advanced rules, which are useful concepts for expressing logic programs. However, answer set solving algorithms commonly require the input program to be represented in a concise and uniform way free of default negation. To that end, the input—a first-order logic program—is converted into such a representation prior to the solving process itself. A common representation uses **nogoods**. A nogood δ is a set of literals $\{l_1, \ldots, l_n\}$ that is violated if all $l_i \in \delta$ are satisfied.

The conversion involves two steps. First, **grounding** translates the first-order logic program to a ground logic program. Second, the program's rules are converted to a set of nogoods, simultaneously replacing default negation with classical negation through **completion**.

GROUNDING Rules r of a first-order logic program P are **ground rules** if they they contain no variables. The **ground instances** $ground(r)$ of a rule r are obtained by systematically replacing the variables in r with all possible

[1] These tools are part of the *Potassco* answer set solving suite and can be found on the website `http://potassco.sourceforge.net/`

ground terms. The **ground instantiation** of a program P is:

$$ground(P) = \bigcup_{r \in P} ground(r)$$

While in some cases, the program's ground instantiation is sufficient to compute the stable models, $ground(P)$ can become an infinite set of rules[2] (of which many are unnecessary in practice). For this reason, the grounding procedure attempts to find a finite subset $P' \subseteq ground(P)$ that leads to the same stable models as $ground(P)$.

UNIFORM REPRESENTATION The **completion nogoods** Δ_P are derived from *Clark's completion* [4], which translates default negation to classical negation. They represent the program's rules as nogoods over literals,

$$\Delta_P = \bigcup_{B \in body(P), B = \{l_1, \ldots, l_n\}} \left\{ \begin{array}{c} \{\overline{B}, \boldsymbol{t}l_1, \ldots, \boldsymbol{t}l_n\}, \\ \{B, \boldsymbol{f}l_1\}, \ldots, \{B, \boldsymbol{f}l_n\} \end{array} \right\}$$
$$\cup \bigcup_{a \in atom(P), body_P(a) = \{B_1, \ldots, B_k\}} \left\{ \begin{array}{c} \{\overline{a}, B_1\}, \ldots, \{\overline{a}, B_k\}, \\ \{a, \overline{B_1}, \ldots, \overline{B_k}\} \end{array} \right\}$$

where $\boldsymbol{t}$ and $\boldsymbol{f}$ map default negation to classical negation on the literal level:

$$\boldsymbol{t}v = \begin{cases} v & \text{if } v \in domain(P) \\ \overline{u} & \text{if } \exists u \in domain(P) : v = {\sim}u \end{cases}$$
$$\boldsymbol{f}v = \begin{cases} \overline{v} & \text{if } v \in domain(P) \\ u & \text{if } \exists u \in domain(P) : v = {\sim}u \end{cases}$$

The nogoods $\{\overline{B}, \boldsymbol{t}l_1, \ldots, \boldsymbol{t}l_n\}$ express that the body B of a rule has to be `true` if all the literals l_i in B hold. In turn, the nogoods $\{B, \boldsymbol{f}l_i\}$ forbid the body B to be `true` if one of its literals l_i is `false`. Similarly, the $\{\overline{a}, B_i\}$ nogoods ensure that if a body B_i is `true`, its corresponding head atom a is derived to be `true` as well. Conversely, nogoods $\{a, \overline{B_1}, \ldots, \overline{B_k}\}$ prevent atoms a to be `true` if no bodies B_i entail them. In the definition of Δ_P, the set of all bodies that have a as the head atom is denoted by $body_P(a)$.

[2] This is the case if the logic program employs functions or arithmetics.

With certain programs P, called **tight programs**, the models of Δ_P are identical to the stable models of P. However, Δ_P might produce more models than that. In this case, the excess models contain atoms that cannot be derived from the original program P in a finite number of steps. These atoms are only part of the models because they derive each other in a cycle. They are thus referred to as **unfounded sets**.

In addition to the completion nogoods, **loop nogoods** Λ_P are defined to eliminate models with unfounded sets:

$$\Lambda_P = \bigcup_{U \subseteq atom(P), EB_P(U) = \{B_1, \ldots, B_k\}} \{\{a, \overline{B_1}, \ldots, \overline{B_k}\} \mid a \in U\}$$

$$EB_P(L) = body(ES_P(L))$$

$$ES_P(L) = \{r \in P \mid head(r) \in L, body(r)^+ \cap L = \varnothing\}$$

$ES_P(L)$ retrieves the rules that externally support a set of atoms $L \subseteq atom(P)$. The bodies of these rules are the *external bodies* $EB_P(L)$. The loop nogoods then disallow atoms to be `true` if there are no external bodies supporting them, that is, if they belong to an unfounded set U.

Together, Δ_P and Λ_P lead to the same stable models as P. With this representation of first-order logic programs P in the form of ground nogoods free of default negation, a search algorithm can now be applied.

2.2 ANSWER SET SOLVING

A classic algorithm for solving the satisfiability problem (SAT) is the *Davis–Putnam–Logemann–Loveland* algorithm (DPLL) [6, 5]. DPLL is a search algorithm that employs chronological backtracking.

Inspired by DPLL, **conflict-driven clause learning** (CDCL) is a search algorithm implemented by many modern SAT and answer set solvers. The CDCL algorithm was introduced by Bayardo and Schrag [3] and Marques-Silva and Sakallah [21]. In contrast to DPLL, which just reverts the last decision made by the algorithm whenever it encounters a conflict, CDCL at-

Algorithm 2.1: Outline of conflict-driven clause learning (CDCL).

```
1  loop
2      compute deterministic consequences
3      if no conflict then
4          if all variables assigned then
5              return variable assignment                    ▷ solution
6          else
7              decide                  ▷ nondeterministic variable assignment
8      else
9          if top-level conflict then
10             return unsatisfiable
11         else
12             analyze conflict                  ▷ learn conflict constraint
13             backjump                          ▷ return to source of conflict
```

tempts to **learn** from conflicts in a procedure called **conflict analysis**. Due to conflict analysis, the solver is able to backtrack multiple levels at the same time and to prune the search space yet to traverse. This pattern is called **backjumping** or *nonchronological backtracking* and constitutes an important improvement of CDCL over DPLL.

This section describes the workings of CDCL. An outline of the general algorithm is given in Section 2.2.1. Then, Section 2.2.2 describes how CDCL is applied to answer set solving. Section 2.2.3 details the conflict analysis step separately because it plays a vital role in this thesis. Finally, the CDCL answer set solver *clasp* is briefly presented in Section 2.2.4.

2.2.1 CONFLICT-DRIVEN CLAUSE LEARNING

An outline of CDCL is given in Algorithm 2.1. The procedure starts with an empty assignment and first computes deterministic consequences (line 2). In

the beginning, the deterministic consequences are the assignments of propositional variables that can be derived from the input directly.

Then, CDCL distinguishes multiple cases. A **conflict** occurs if the current assignment is inconsistent with the input program's rules. If there is no conflict and the assignment is already complete, the procedure has found a solution (line 5). Often however, the deterministic consequences alone do not lead to complete assignments. Then, the algorithm nondeterministically assigns a truth value to one of the unassigned propositional variables (line 7). This is called a **decision** and the resulting literal l_d is the **decision literal**.

Sometimes, the assignment becomes inconsistent in the process, which results in a conflict. In the event that the conflict is obtained from deterministic consequences alone, the program has no solutions (line 9). Otherwise, CDCL analyzes the conflict (line 12). As an outcome of this procedure, CDCL derives a **conflict constraint**, which is added to the input program in order to prune the future search space. The technique of memorizing information about conflicts is referred to as **learning**. After learning a conflict constraint, CDCL undoes some of the last decisions and backjumps (line 13).

These steps are repeated to enumerate the input program's solutions. If the procedure does not return any assignment after traversing the entire search space, the program is unsatisfiable.

2.2.2 ANSWER SET SOLVING WITH CDCL

DEFINITIONS Given a current assignment A, nogoods are **unit** if all literals except one are assigned and the remaining literal is unassigned. The remaining literal is called the **unit literal**. In order for the the unit nogood to be satisfied, the complement of the unit literal must be assigned. The process of iteratively identifying unit nogoods and assigning the complements of unit literals is called **unit propagation**. In other words, unit propagation computes the deterministic consequences until no unit nogoods remain.

If a literal l results from the unit propagation of a nogood ρ, ρ is called an

antecedent or **reason** of l, denoted by $reason(l)$. A literal might have multiple antecedent nogoods. To resolve the ambiguity, $reason(l)$ always returns one of them (chosen arbitrarily).

If the ordered assignment $A = (l_1, \ldots, l_n)$ is not yet complete and there are no more unit nogoods left, a nondeterministic decision needs to be made. Each literal $l \in A$ is associated with a **decision level**, a nonnegative integer stored in $dlevel(l)$. All the literals that are assigned prior to any decision receive the decision level 0. To make a decision, an assignment l_d on an unassigned propositional variable is made. Upon a decision, the decision level is incremented by 1. The current decision level is referred to as dl.

PROCEDURE The general CDCL procedure in 2.1 can be applied to answer set solving in order to enumerate the stable models of a logic program. In accordance with the uniform nogood representation, CDCL is changed into conflict-driven *nogood* learning. Algorithm 2.2 shows the concrete answer set solving procedure (CDNL-ASP).

Thanks to the translation in Section 2.1.3, CDNL-ASP can restrict itself to input programs P represented as sets of ground nogoods over $atom(P) \cup body(P)$. The algorithm starts with an empty assignment A, an empty set of learned conflict constraints ∇, and the initial decision level $dl = 0$ (lines 1–3). To remain consistent with the program representation, conflict constraints are also modeled as nogoods.

As in CDCL, the deterministic consequences are computed next via the function *propagate* (line 5). This method performs unit propagation on Δ_P and ∇ and additionally unfounded set checking with Λ_P to falsify atoms that belong to unfounded sets early.[3]

In the following case analysis, conflicts are detected if any nogood is violated with respect to the current assignment (line 6). Top-level conflicts occur when the violated nogood's literals belong to decision level 0—implying

[3]Note that in practice, Λ_P is not represented explicitly because the size of Λ_P is exponential in $atom(P) \times body(P)$ [10].

Algorithm 2.2: CDCL applied to answer set solving (CDNL-ASP).

Input: A program P, its completion nogoods Δ_P and loop nogoods Λ_P
Output: A stable model of P or `unsatisfiable`

```
1  A := ∅                                          ▷ assignment over atom(P) ∪ body(P)
2  ∇ := ∅                                          ▷ learned conflict constraints
3  dl = 0                                          ▷ decision level
4  loop
5      (A, ∇) := propagate(P, ∇, A)                ▷ deterministic consequences
6      if exists ε ∈ Δ_P ∪ ∇ with violated(ε, A) then   ▷ conflict
7          if max({dlevel(l) | l ∈ ε} ∪ {0}) = 0 then
8              return unsatisfiable
9          (δ, dl) := analyzeConflict(ε, P, ∇, A, dl)
10         ∇ := ∇ ∪ {δ}                            ▷ learn conflict constraint
11         A := A ⊖ {l ∈ A | dl < dlevel(l)}       ▷ backjumping
12     else if A^T ∪ A^F = atom(P) ∪ body(P) then
13         return A^T ∩ atom(P)                    ▷ stable model
14     else
15         l_d := decide(P, ∇, A)                  ▷ nondeterministic decision
16         dl := dl + 1
17         dlevel(l_d) := dl
18         A := A ⊕ l_d
```

that there are no stable models (line 8). All the other conflicts are analyzed, which results in a new learned nogood δ that is added to ∇ and a new decision level dl (lines 9–10). This decision level is the one the algorithm then backjumps to, undoing all the assignments made on higher decision levels (line 11). The operation $A \ominus L$ stands for the elimination of the literals $l \in L$ from the ordered assignment A.

Next, complete and conflict-free assignments are detected, in which case the corresponding stable model is returned (lines 12–13). If no conflict exists and unit propagation did not make the assignment complete, a decision is made and the decision level incremented (lines 15–18). To that end, a variable that is unassigned with respect to A is nondeterministically chosen and assigned a truth value. Complementary to $\ominus$, the operator $\oplus$ extends the assignment A by the decision literal l_d.

2.2.3 CONFLICT ANALYSIS

Conflict analysis is invoked whenever a conflict is detected (except for top-level conflicts). The cause of a conflict is a nogood $\varepsilon = \{l_1, \ldots, l_n\}$ with $l_i \in atom(P) \cup body(P)$ that is unsatisfied with respect to the current assignment A and the program P. The nogood ε is referred to as the **violated nogood**. The task of conflict analysis is to derive a new conflict constraint δ (added to ∇) and a backjump level k.

Conflict analysis must guarantee the returned decision level k to be **asserting**. That means that after backjumping, the learned conflict constraint δ becomes unit, leading to further deterministic consequences on that level. Then, the solving algorithm can continue traversing the search space but avoids running into the same conflict again thanks to the learned nogood δ.

IMPLICATION GRAPHS The process of deriving a conflict constraint δ can be illustrated with the help of **implication graphs** [31]. A typical implication graph is sketched in Figure 2.1. The implication graph's vertices consist of the

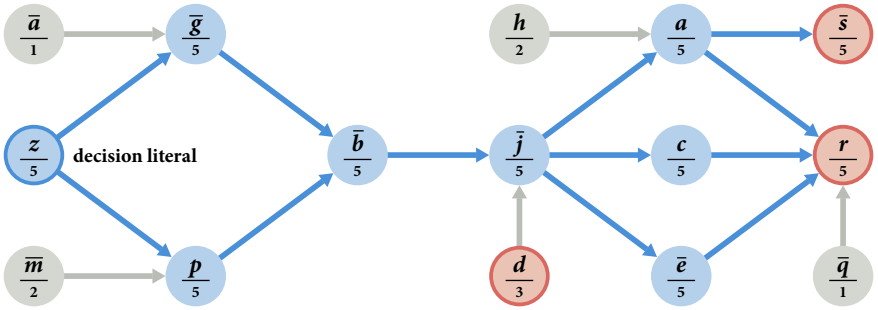


Figure 2.1: A part of an implication graph showing decision level 5. On this level, a conflict exists because the nogood $\varepsilon = \{d, r, \bar{s}\}$ (shown in red) is violated under the assignment. Nodes stand for literals, while edges illustrate the implication relationship. The lower part of a node l denotes $dlevel(l)$.

literals in the current assignment A (shown in the upper part of the nodes), along with their decision levels (in the lower part). An edge from a vertex l_1 to l_2 exists if l_1 has an antecedent nogood containing l_2.

For instance, $\bar{b}$ has two incident edges from $\bar{g}$ and p. This means that $\bar{b}$ is implied by $\bar{g} \wedge p$. This must be the case because the underlying program contains a nogood $\{\bar{g}, p, b\}$—the antecedent nogood of $\bar{b}$. This nogood became unit after assigning $\bar{g}$ and p, resulting in the assignment $\bar{b}$. Decision literals have no antecedents and no incoming edges in the graph because they are nondeterministically assigned and not deterministic consequences.

RESOLUTION In the example, nogood $\varepsilon = \{d, r, \bar{s}\}$ is violated. Conflict analysis essentially consists of **resolving** this violated nogood until a certain point.

In Figure 2.1, r is implied by $a \wedge c \wedge \bar{e} \wedge \bar{q}$. Thus, the antecedent ρ of r is:

$$\rho = reason(r) = \{a, c, \bar{e}, \bar{q}, \bar{r}\}$$

As r is contained in ε and $\bar{r}$ in ρ, the **resolution rule** may be applied:

$$\frac{\{a_1, \ldots, a_m, c\}, \quad \{b_1, \ldots, b_n, \bar{c}\}}{\{a_1, \ldots, a_m, b_1, \ldots, b_n\}}$$

This rule allows the two nogoods ε and ρ to be joined after removing both r and $\overline{r}$, producing a new nogood δ_1:

$$\delta_1 = \{d, \overline{s}, a, c, \overline{e}, \overline{q}\}$$

Under the current assignment, δ_1 is also a violated nogood because its deterministic consequences still lead to the conflict ε. However, δ_1 can be added as a learned nogood to avoid future conflicts under similar circumstances.

Before adding the nogood to ∇, the procedure may be continued by resolving further literals. Which and how many literals are resolved is specified by a **resolution scheme**. An advanced strategy found in modern solvers is the **first unique implication point** (first-UIP) resolution scheme [7, 22]. Empirical observations show that the first-UIP strategy performs better than other resolution schemes [31]. With this strategy, literals on the current decision level dl are resolved until the resulting nogood contains exactly one literal l_f on dl. Resolution proceeds in backward order of assignment, and literals on levels lower than dl are not resolved.

If a literal is the only one that belongs to a certain decision level (such as l_f), it is called a **unique implication point** (UIP) of that level. l_f is, more precisely, the *first* UIP reached by resolution.

To obtain the first UIP in the given example, δ_1 must be resolved further to eliminate a, c, and $\overline{e}$. This finally produces nogood δ_2:

$$\delta_2 = \{d, \overline{q}, h, \overline{j}\}$$

In this nogood, $\overline{j}$ is the only literal that belongs to decision level dl and is the first UIP. Other UIPs in Figure 2.1 are $\overline{b}$ and the decision literal z. In fact, all decision literals are UIP because the other literals on the same level can always be resolved to the decision literal.

The first-UIP resolution scheme is sketched in Algorithm 2.3. Iteratively, the procedure resolves the most recent literal l_f in δ with its antecedent ρ (lines 3 and 6). This is repeated until only one literal on level dl remains—the first UIP (line 4). Then, the backjump level k is computed as the maximum

Algorithm 2.3: Conflict analysis with the first-UIP resolution scheme.

Input: A violated nogood ε, a program P, a set of conflict nogoods ∇, an assignment $A = (l_1, \ldots, l_n)$, and the current decision level dl

Output: A derived conflict nogood δ and the backjump level k

```
1 δ := ε
2 loop
3     f := max{1 ≤ i ≤ n | l_i ∈ δ }          ▷ most recent assignment l_f ∈ δ
4     if {l ∈ δ | dlevel(l) = dl} ∖ {l_f} = ∅ then
5         break                                ▷ first UIP l_f found
6     δ := (δ ∖ {l_f}) ∪ (reason(l_f) ∖ {l̄_f})   ▷ resolution
7 k := max{dlevel(l) | l ∈ δ ∖ {l_f}}          ▷ backjump level k
8 return (δ, k)
```

decision level of the literals in δ except for the first UIP l_f (line 7). The derived conflict constraint δ and k are finally returned.

When the solver backjumps to the decision level k, all literals in δ but l_f stay assigned. Hence, unit propagation will result in the assignment of $\overline{l}_f$. For this reason, δ and k are asserting.

The first UIP can always be found because in the worst case, the first-UIP scheme resolves all literals on decision level dl to the decision literal l_d (z in the example). Thus, the condition in line 4 must be fulfilled at some point, at the latest when reaching l_d (l_d cannot be resolved further and is a UIP).

2.2.4 CLASP

Clasp is a high-performance answer set solver of the *Potassco* collection. Clasp is based on the CDCL algorithm and provides many configuration options controlling certain implementation details and optimizations, such as additional preprocessing steps and decision heuristics. Conflict analysis is

implemented with the first-UIP resolution scheme but uses clauses instead of nogoods to represent conflict constraints. However, both representations are equivalent because a clause $(l_1 \vee \ldots \vee l_n)$ can be translated to a nogood $\{\overline{l_1}, \ldots, \overline{l_n}\}$ with the same meaning and vice versa.

All implementations in this thesis solve answer set programs with clasp (in conjunction with the Potassco grounder **gringo**), and the evaluations use clasp's rich statistical output.

2.3 PLANNING

Planning is a decision-making problem that aims at achieving a specified goal from a given initial state [13]. For this purpose, sequences of actions are determined that, when applied, successively transform the initial state to subsequent states that accomplish the goal.

States describe the static and dynamic environment of the planning problem. Static properties (such as the size of a chess board) are called **domain properties**, while **fluents** represent the mutable characteristics of the environment (for instance, the position of a chess pawn). The **initial state** consists of facts specifying the fluents that shall hold in the beginning. Complementary, the **goal situation** describes properties that are requested to hold in the end. While the goal situation may define few properties and leave others open, the initial state must be completely described by its fluents (in other words, it may not be partial).

Actions manipulate some of the fluents (causing **effects**) and may require certain fluents to hold in order to be performed (**preconditions**). Applying an action is **legal** if the preconditions are met with respect to some state. Actions are applied one by one. The order in which actions are executed is commonly identified with consecutive **time steps**. The initial state corresponds to time step 0 and the first action is applied at time step 1. Then, the resulting successor state is established in time step 1, and so on.

Plans are sequences of actions. Given an initial state and a goal situation, a

plan is valid if all actions are legal with respect to the state sequence induced by the plan and the goal situation is satisfied. **Automated planning** is the process of finding valid plans computationally.

2.3.1 THE PLANNING DOMAIN DEFINITION LANGUAGE

The **planning domain definition language** (PDDL) was specified by McDermott et al. [14]. PDDL stems from STRIPS, a language developed by Fikes and Nilsson to describe planning problems [8]. STRIPS instances contain an initial state, a goal situation, and a set of actions along with the necessary preconditions and effects. PDDL provides a number of extensions to STRIPS, such as conditional effects and preferences.

Listing 2.3 illustrates the basic syntax and semantics of PDDL with a simple encoding of the Travelling Salesperson problem. The first two lines state the name of the domain `tsp` and declare that typing information is used by the encoding. Then, the type `vertex` is declared (line 3) and four predicates are introduced: `move`, `at`, `linked`, and `visited` (lines 4–7). There, identifiers preceded by `?` denote the names of variables, while identifiers following the character `-` specify the variable types. Finally, the encoding defines the action `move` by stating its parameters to be vertices (line 9) and listing the preconditions and effects (lines 10–14). Conjunctions with `and` are available to enumerate multiple preconditions and effects. The meaning of the precondition of `move` is that to travel from A to B, the salesperson has to be in A and there must be a link between A and B. After applying the action, the salesperson is at B, not at A anymore, and has visited the destination.

Like in ASP's modeling language, PDDL specifications are commonly split into a problem encoding (*domain description*) and a problem instance (*problem description*). Listing 2.4 shows the PDDL definition of a trivial instance of the traveling salesperson problem. In the first two lines, the instance is named `tsp-instance` and the domain `tsp`, which the instance belongs to, is specified. Three locations (called `Berlin`, `Hamburg`, and `Hannover`) are

Listing 2.3: PDDL encoding of the Travelling Salesperson problem.

```
(define (domain tsp)
    (:requirements :typing)
    (:types vertex)
    (:predicates (move ?v1 ?v2 - vertex)
        (at ?v - vertex)
        (linked ?v1 ?v2 - vertex)
        (visited ?v - vertex))
    (:action move
        :parameters (?v1 ?v2 - vertex)
        :precondition (and (at ?v1)
            (linked ?v1 ?v2))
        :effect (and (at ?v2)
            (not (at ?v1))
            (visited ?v2))))
```

Listing 2.4: PDDL instance of the Travelling Salesperson problem.

```
(define (problem tsp-instance)
    (:domain tsp)
    (:objects Berlin Hamburg Hannover - vertex)
    (:init (at Berlin)
        (linked Berlin Hamburg)
        (linked Hamburg Berlin)
        (linked Berlin Hannover)
        (linked Hannover Berlin)
        (linked Hamburg Hannover)
        (linked Hannover Hamburg))
    (:goal (and (at Berlin)
        (visited Berlin)
        (visited Hamburg)
        (visited Hannover))))
```

introduced in line 3. Then follow the initial state consisting of the start location `Berlin` and the connections between the cities (lines 4–10). The goal situation is stated at the end of the problem instance and requires the salesperson to return to `Berlin` after visiting all the three cities (lines 11–14).

To find plans for PDDL definitions, problem encodings and instances are solved with a planning system such as Hoffmann and Nebel's *FF* [16].

2.3.2 PLASP

In contrast to dedicated planning systems, **plasp** is a prototypical system that aims at solving planning problems with an answer set solver [11]. This requires the PDDL specification to be first translated to ASP. However, with this translation, plasp can rely on the already existing and mature infrastructure for answer set solving.

Listings 2.5 and 2.6 show how the PDDL encoding and instance in Listings 2.3 and 2.4 are translated to ASP by plasp. In Listing 2.5, the property `linked` was translated to a domain predicate denoted by `holds`, which enables the `move` action (line 3). The actions' preconditions are defined by the meta predicate `demands` (line 5). Effects of actions follow in lines 7–9 and are denoted by `adds` and `deletes`.

The instance in Listing 2.6 lists all available locations (lines 1–3) and the connections between them (lines 5–10). The initial state and goal situation are specified with plasp's `init` and `goal` meta predicates (lines 12–17).

To solve this instance with an answer set solver such as clasp, an additional encoding is required that defines how plasp's meta predicates should be understood. This **meta encoding** is shown in Listing 2.7. Essentially, the meta encoding establishes the initial state (line 5) and nondeterministically creates a plan (line 8), whose maximal length is set by an external constant `horizon`. The encoding prevents illegal actions (lines 11–12) and applies the effects of the chosen actions (lines 15–18). In the end, the encoding verifies that the plan meets the goal (lines 21–23).

Listing 2.5: Translation of Listing 2.3 to ASP with plasp.

```
1 object(X) :- typedobject(vertex(X)).
2
3 action(move(V1, V2)) :- holds(linked(V1, V2)).
4
5 demands(move(V1, V2), at(V1), true) :- holds(linked(V1, V2)).
6
7 adds(move(V1, V2), at(V2)) :- holds(linked(V1, V2)).
8 adds(move(V1, V2), visited(V2)) :- holds(linked(V1, V2)).
9 deletes(move(V1, V2), at(V1)) :- holds(linked(V1, V2)).
```

Listing 2.6: Translation of Listing 2.4 to ASP with plasp.

```
1 typedobject(vertex(hamburg)).
2 typedobject(vertex(berlin)).
3 typedobject(vertex(hannover)).
4
5 holds(linked(berlin, hamburg)).
6 holds(linked(hamburg, berlin)).
7 holds(linked(berlin, hannover)).
8 holds(linked(hannover, berlin)).
9 holds(linked(hamburg, hannover)).
10 holds(linked(hannover, hamburg)).
11
12 init(at(berlin)).
13
14 goal(at(berlin), true).
15 goal(visited(berlin), true).
16 goal(visited(hamburg), true).
17 goal(visited(hannover), true).
```

Listing 2.7: Meta encoding for solving plasp encodings and instances.

```
% Maximum plan length, must be specified with -c horizon=<n>
time(0..horizon).

% Establish initial state
holds(F, 0) :- init(F).

% Perform actions
1 {apply(A, T) : action(A)} 1 :- time(T), T > 0.

% Check preconditions
:- apply(A, T), demands(A, F, true), not holds(F, T - 1).
:- apply(A, T), demands(A, F, false), holds(F, T - 1).

% Apply effects
holds(F, T) :- apply(A, T), adds(A, F), action(A), time(T).
del(F, T) :- apply(A, T), deletes(A, F), action(A), time(T).
holds(F, T) :- holds(F, T - 1), not del(F, T),
               time(T), time(T - 1).

% Verify that goal is met
1 {terminal(T) : time(T) : T > 0} 1.
:- terminal(T), goal(F, true), not holds(F, T).
:- terminal(T), goal(F, false), holds(F, T).
```

When solving the three answer set programs together, clasp returns two stable models. The first sequence of actions is:

```
apply(move(hannover, hamburg), 2)
apply(move(hamburg, berlin), 3)
```

and the sequence of the second solution is:

```
apply(move(hamburg, hannover), 2)
apply(move(hannover, berlin), 3)
```

These indeed correspond to the solutions of the original PDDL problem.

2.4 STATISTICS: THE GEOMETRIC MEAN

On several occasions in this thesis, solving times of answer set solvers are measured to evaluate two competing implementations or configurations A and B. Given a set of test instances $\{t_1, \ldots, t_n\}$, a measurement is performed once with each instance and configuration. The results might look as follows:

instance	**configuration *A***	**configuration *B***	**factor**
1	1 s	100 s	1/100
2	100 s	1 s	100

To assess whether A has a benefit over B, the measurements for each instance are set in relation by computing the ratio of A to B (right column). In the end, all factors are averaged to make a general statement about A vs. B.

However, the *arithmetic mean* is a poor choice for computing the average of the solving time ratios. In the above example, the arithmetic mean of the factors is 50.005. It would be misleading to interpret this number as a slowdown of factor 50 from A to B. In fact, the same number is obtained when computing the ratio of B to A, which would mean that A was slower than B and B slower than A at the same time.

Ratios and percentages are better dealt with by using the **geometric mean**.

For numbers $\{x_1, \ldots, x_n\}$, it is defined as follows:

$$\mu_g = \sqrt[n]{x_1 * \ldots * x_n}$$

In the simple example, the geometric mean reflects the actual difference between the two configurations much better. $\sqrt[2]{1/100 * 100} = 1$, meaning that the results of both configurations are indeed comparable.

To calculate the mean μ_g, standard deviation σ_g, and a confidence interval $CI_g = (x_l, x_h)$ with the geometric instead of the arithmetic mean, the input measurements can be log-transformed [23]. For this purpose, the arithmetic mean μ_a, standard deviation σ_a, and confidence interval CI_a are computed as usual on the transformed data. Finally, μ_g, σ_g, and CI_g are obtained by undoing the logarithm on μ_a, σ_a, and CI_a with an exponential function.

While solving time ratios cannot safely be assumed to be normally distributed (which is the reason why the arithmetic mean is not applicable), their logarithm is normal, at least to a greater extent. A normal distribution of the logarithmized data is the condition for applying the geometric mean.

3

KNOWLEDGE EXTRACTION

Modern CDCL solvers such as clasp gather knowledge in the form of learned conflict constraints when solving ASP problem instances. This is an outcome of conflict analysis (see Section 2.2).

Accessing the learned conflict constraints is essential for the present research in two places. First, to analyze to which extent CDCL solvers benefit from reusing learned knowledge by directly feeding it back to them (Chapter 4). Second, the learned conflict constraints serve as a basis to generate candidates for generalized knowledge in the implementation of the *generalized knowledge feedback loop* (Chapters 5 and 6).

In this thesis, the answer set solver clasp is used to extract learned knowledge. Since clasp provides no native interface for this purpose, the solver has to be instrumented to record all the conflict constraints learned during conflict analysis. Furthermore, a modification of clasp's resolution scheme

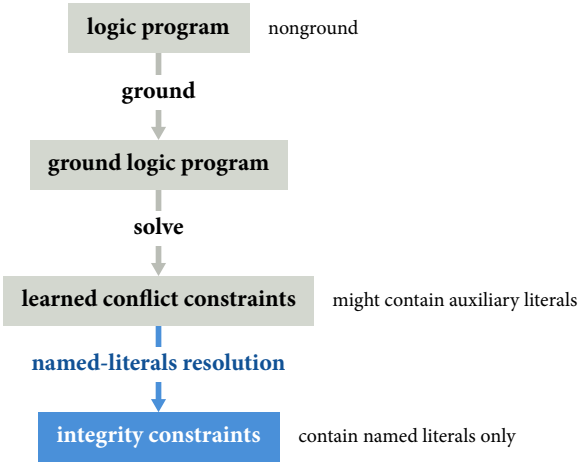


Figure 3.1: To extract knowledge, the CDCL algorithm is modified (blue): The custom named-literals resolution scheme ensures that logged conflict constraints only contain named literals. The learned constraints are output as integrity constraints in ASP's modeling language for further usage.

is necessary, as the default implementation might cause difficulties when looking up the names of literals.

The required changes to the solver are highlighted in Figure 3.1. The modified clasp variant is referred to as the Feedback Clasp variant and capable of recording learned knowledge thanks to the new resolution scheme and appropriate configuration options. In contrast, the original implementation with default configuration is called the Baseline Clasp variant.

STRUCTURE This chapter details the changes made to clasp. Section 3.1 explains how learned conflict constraints are recorded. Section 3.2 describes the modified resolution scheme and its implementation. The chapter concludes with a short overview of postprocessing steps (Section 3.3), which are used in later parts of this thesis.

3.1 RECORDING LEARNED CONFLICT CONSTRAINTS

In the present thesis, learned conflict constraints are mainly recorded to enrich instances, that is, to add new rules to their answer set program. In ASP's modeling language, integrity constraint rules are a convenient way of representing conflict constraints—a nogood $\{v_1, \ldots, v_m, \overline{u_1}, \ldots, \overline{u_n}\}$ directly corresponds to the following integrity constraint rule:

```
:- v_1, ..., v_m, not u_1, ..., not u_n.
```

To record conflict constraints in this format, clasp was extended to invoke a custom method every time a nogood is learned. This method essentially retrieves the signs and names of all literals belonging to the conflict constraint. Then, the method returns an according integrity constraint in ASP's modeling language. Signs are part of the literals' data structure, while the literal names (the names of the corresponding propositional variables) are stored in clasp's symbol table. Clasp internally manages the symbol table to map external IDs to *symbols* (internal literals with respective names). A custom method was added to the symbol table to look up the names of literals.

In this way, conflict constraints are recorded immediately when learned by clasp. The output is directed to a pipe for other applications to read. With this design, extracted knowledge need not be stored in a file temporarily.

3.2 THE NAMED-LITERALS RESOLUTION SCHEME

Even though this procedure records learned conflict constraints correctly, looking up certain literal names might fail due to a complication in conjunction with clasp's implementation. The reason for this is that clasp internally introduces *auxiliary literals* on the one hand and implements the first-UIP resolution scheme (see Section 2.2.3) on the other hand. Hence, conflict constraints learned during conflict analysis might comprise such auxiliary literals. This is problematic because they cannot be associated with named lit-

Algorithm 3.1: The named-literals resolution scheme.

Input: A violated nogood ε, a program P, a set of conflict nogoods ∇, an assignment $A = (l_1, \ldots, l_n)$, and the current decision level dl

Output: A derived conflict nogood δ and the backjump level k

```
1  δ := ε
2  loop
3      f := max{1 ≤ i ≤ n | l_i ∈ δ }          ▷ most recent assignment l_f ∈ δ
4      if {l ∈ δ | dlevel(l) = dl} ∖ {l_f} = ∅ and name(l_f) ≠ ⊥ then
5          break                                ▷ first named UIP l_f found
6      δ := (δ ∖ {l_f}) ∪ (reason(l_f) ∖ {l̄_f})   ▷ resolution on level dl
7  while exists l ∈ δ with 1 ≤ dlevel(l) < dl and name(l) = ⊥ do
8      δ := (δ ∖ {l}) ∪ (reason(l) ∖ {l̄})     ▷ resolution on levels < dl
9  k := max{dlevel(l) | l ∈ δ ∖ {l_f}}          ▷ backjump level k
10 return (δ, k)
```

erals in the original answer set program, thus rendering entire conflict constraints useless for the purposes of this thesis.

To overcome this issue, this section presents a new resolution scheme that guarantees all logged conflict constraints to reference named literals only. In contrast to first-UIP resolution, which stops resolution after finding the first unique implication point, the *named-literals resolution scheme* additionally resolves all the literals whose name look-up fails.

3.2.1 IMPLEMENTATION

Algorithm 3.1 shows an outline of the new scheme. Like clasp's default implementation (Algorithm 2.3), it accepts the violated nogood ε and clasp's current assignment A as input and returns an asserting, derived conflict constraint δ as well as the backjump level k.

First, the algorithm finds the first *named* UIP on the current decision level *dl* (lines 2–6). This is similar to the search for the first UIP, except that a UIP is also resolved if the look-up of the corresponding propositional variable fails. After completing the loop, l_f is the only remaining literal on *dl* and it has an accessible name. Thus, l_f is the first named UIP.

In the worst case, the search for the first named UIP proceeds until it ends up with the decision literal l_d of level *dl*. As mentioned earlier, decision literals are UIPs. Additionally, l_d is guaranteed to have an accessible name thanks to configuration options explained in Section 3.2.3. This ensures that the procedure always terminates successfully.

The second loop resolves unnamed literals on decision levels lower than *dl* (lines 7–8). Resolution terminates if no unnamed literals remain.

In the end, the procedure has derived the nogood δ through resolution. As with first-UIP resolution, the backjump level k is determined as the highest decision level of a literal in δ except for the asserting literal (line 9).

3.2.2 EXAMPLE

Figure 3.2 shows an implication graph similar to the one in Figure 2.1. However, some of the literals (x_1 to x_4) are unnamed.

Table 3.1 contains the complete resolution sequence performed by the named-literals resolution scheme. Starting with the violated nogood $\varepsilon = \{d, r, \overline{s}\}$, the named-literals resolution scheme first attempts to find the first named UIP (rows 1–6). The classical first-UIP scheme would stop resolution after finding the first UIP $\overline{x_2}$ (row 5). However, $\overline{x_2}$ is an unnamed literal in this example. Because of this, the named-literals resolution scheme continues by resolving $\overline{x_2}$ and obtains $\{d, \overline{b}, x_4, \overline{q}\}$ (row 6). The literal $\overline{b}$ is also a UIP and—in contrast to $\overline{x_2}$—it has a name.

Having found the first named UIP $\overline{b}$, the resolution scheme resolves all unnamed literals on lower decision levels. In this case, this concerns only x_4 (row 7). The final nogood derived by the named-literals scheme is $\{d, \overline{b}, v, \overline{q}\}$.

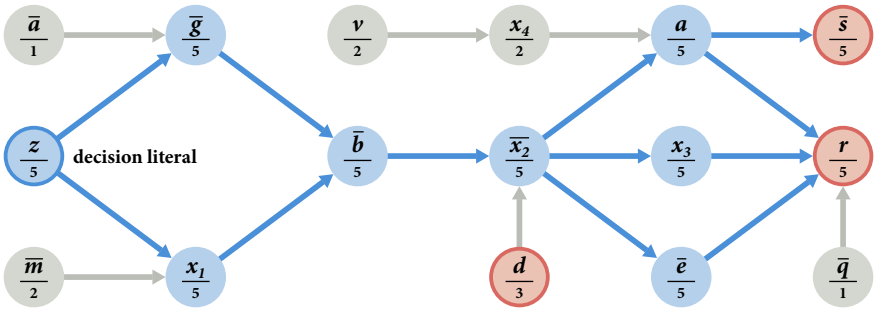


Figure 3.2: A part of an implication graph under the conflict $\varepsilon = \{d, r, \overline{s}\}$ (shown in red). Nodes stand for literals, while edges illustrate the implication relationship. The lower part of a node l denotes $dlevel(l)$. The nodes x_1 to x_4 are unnamed literals.

	resolved literal	**antecedent**	**resulting nogood**
ε			$\{d, r, \overline{s}\}$
1	$\overline{s}$	$\{a, s\}$	$\{d, r, a\}$
2	r	$\{a, x_3, \overline{e}, \overline{q}, \overline{r}\}$	$\{d, a, x_3, \overline{e}, \overline{q}\}$
3	a	$\{\overline{x_2}, x_4, \overline{a}\}$	$\{d, \overline{x_2}, x_4, x_3, \overline{e}, \overline{q}\}$
4	x_3	$\{\overline{x_2}, \overline{x_3}\}$	$\{d, \overline{x_2}, x_4, \overline{e}, \overline{q}\}$
5	$\overline{e}$	$\{\overline{x_2}, e\}$	$\{d, \overline{x_2}, x_4, \overline{q}\}$
6	$\overline{x_2}$	$\{\overline{b}, d, x_2\}$	$\{d, \overline{b}, x_4, \overline{q}\}$
7	x_4	$\{v, \overline{x_4}\}$	$\{d, \overline{b}, v, \overline{q}\}$

Table 3.1: Resolution sequence of the named-literals resolution scheme for the conflict in Figure 3.2.

3.2.3 CONFIGURATION OPTIONS

Throughout the entire thesis, the modified FEEDBACK CLASP variant is invoked with the following command-line arguments:

```
clasp --log-learnts --resolution-scheme=named
      --heuristic=Domain --dom-mod=1,16
      --loops=no --reverse-arcs=0 --otfs=0
```

The former two configuration options are custom additions made to clasp: The flag `--log-learnts` enables the conflict constraint logging procedure detailed in the beginning of this chapter. `--resolution-scheme` switches between the default (`first-uip`) and the modified scheme (`named`).

Then, the two domain-related configuration options ensure that recorded conflict constraints never contain any unnamed literals. The rationale is that given these settings, clasp favors selecting named atoms for decisions over other, unnamed propositional variables such as auxiliaries and bodies. Decision literals are UIPs inherently (see Section 2.2.3). This guarantees that the first named UIP on decision level *dl* exists and is found by the named-literals resolution scheme. What concerns lower decision levels, literals obtained by the resolution scheme are always named—in the worst case, the implementation terminates resolution at decision literals, which are known to be named.

The above statements are based on the assumption that assignments containing literals for all named atoms always lead to *complete* assignments. This is not necessarily the case. For example, an answer set program might specify not to export any names by using the `#hide` command in gringo 3. Then, clasp needs to make decisions on unnamed propositional variables, as there are no named ones available. To avoid such issues, the programs used in this thesis export all the names.

The latter three arguments disable multiple solver strategies that the current implementation does not support.

3.2.4 IMPLEMENTATION CONSIDERATIONS

The named-literals resolution scheme searches for the first named UIP. This behavior is the result of three considerations: First, a UIP must be found on the decision level *dl* in order to obtain an asserting literal. This literal is, in turn, necessary to ensure that unit propagation is possible after backjumping (see Section 2.2.2). Second, as stated earlier, the UIP needs to be named. Third, Zhang et al. empirically observed that first-UIP resolution leads to lower solving times than other common resolution schemes [31].

For this reason, the named-literals resolution scheme aims at terminating resolution on *dl* as early as possible, too. On the decision level *dl*, resolution is stopped at the *first* named UIP, and on levels lower than *dl*, principally at named literals. This approach (as depicted in Algorithm 3.1) is referred to the First Named UIP design.

In an earlier iteration of the resolution scheme (referred to as the Decision design), the literals on decision level *dl* were resolved until the respective decision literal was obtained. This is also a valid implementation, since the decision literal on *dl* is a UIP as well as a named UIP.

The effect of both designs on solving performance was briefly analyzed by conducting the study in Chapter 4 once with each of the two implementations. For each call of the Feedback Clasp variant, the solving time ratio between both implementations was computed. The results show that on average[1], the revised First Named UIP design performed about 41 % faster than the earlier Decision implementation. Because of this, the First Named UIP implementation is used in all following analyses.

Table 3.2 shows the times required to solve the instances from Chapter 4 with named-literals resolution (as well as the number of conflicts) in comparison to first-UIP resolution. Visibly, the requirement of having conflict constraints contain only named literals comes at the cost of reduced solving

[1] As this analysis deals with *ratios* of solving times, the *geometric* mean is used instead of the arithmetic mean here. See Section 2.4 for more details on this topic.

problem	solving time	conflicts
Ricochet Robots	120.22 %	75.18 %
Labyrinth	209.41 %	40.30 %
Hanoi Tower	214.27 %	106.21 %
Solitaire	814.23 %	92.04 %
Hamiltonian Cycle	368.93 %	45.10 %
Graph Coloring	400.52 %	45.22 %
Knight Cycle	0.15 %	0.17 %

Table 3.2: Performance of the named-literals resolution scheme, using Feedback Clasp and the options shown in Section 3.2.3 (except for `--log-learnts`). Solving times and numbers of conflicts are relative to the ones measured with first-UIP resolution (Baseline Clasp with default configuration).

performance.[2] This could be due to the fact that the named-literals resolution scheme needs to resolve unnamed entries on decision levels below *dl* (in contrast to first-UIP resolution). Furthermore, a name look-up for each propositional variable is necessary, which leads to a less cache-efficient implementation than the first UIP resolution scheme.

The named-literals resolution scheme was implemented in a forked version of clasp 3.1.1. For the Baseline Clasp and Feedback Clasp variants, two separate binaries were built, the former one stemming from clasp's unchanged code base and the latter one comprising the presented changeset.

The final implementation of the named-literals resolution scheme was validated against clasp's set of 370 acceptance tests. Additionally, it was verified that adding the learned integrity constraints to the test instances did not eliminate answer sets in any of the cases.

[2] The drastic decrease of solving times with Knight Cycle instances appears to be related to the way the instances were generated. This is discussed in Section 4.4.

3.3 POSTPROCESSING

After having recorded the learned conflict constraints, different types of postprocessing can be performed on the output. One example is *conflict constraint sorting*, that is, ordering the logged conflict constraints by different criteria in a second pass. This postprocessing step is important in Chapter 4, which analyzes the effects of feeding back subsets of the learned conflict constraints to the solver. There, the order in which conflict constraints are selected for feedback is varied (for instance, selecting n conflict constraints with the lowest number of literals).

Another postprocessing step used throughout this thesis is *subsumed conflict constraint removal*. During conflict analysis, clasp often learns conflict constraints that are subsets of previously learned ones (and thus more general and stronger than those). Similar to how clasp operates when enabling *on-the-fly subsumption*, conflict constraints may be removed from the output if they are *subsumed* in this way.

4

DIRECT KNOWLEDGE FEEDBACK

The objective of this thesis is to assess the potential benefits of *reusing* knowledge learned by modern CDCL solvers while solving problem instances. Reusing learned knowledge consists of enriching an instance with conflict constraints that the CDCL solver learned in a previous run (extracting the conflict constraints with the means described in Chapter 3). As this mechanism involves a feedback step (highlighted in blue in Figure 4.1), it is also referred to as *knowledge feedback*.

A sophisticated form of knowledge feedback strengthens learned knowledge by generalization before the feedback step (see Chapters 5 and 6). But before coming to that, this chapter investigates how CDCL solvers already profit from a simple approach of knowledge feedback: appending the learned conflict constraints to an instance directly—that is, in propositional form and without any further processing.

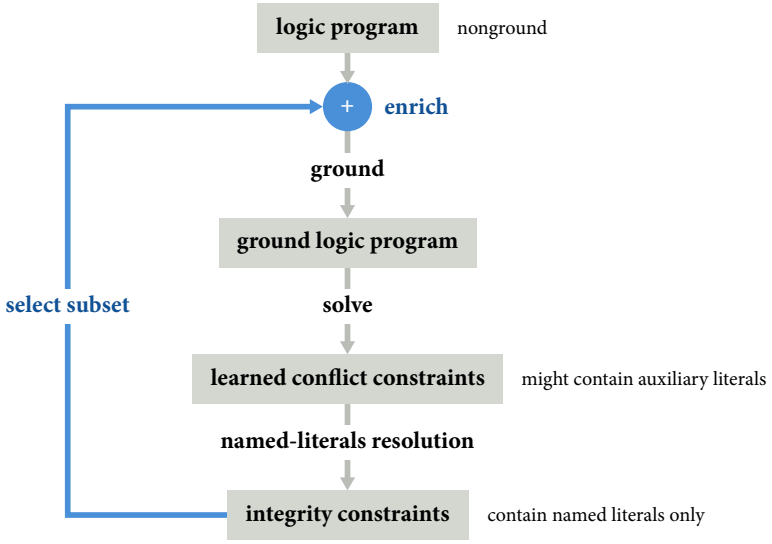


Figure 4.1: Direct knowledge feedback incorporates a feedback mechanism (blue): When solving a logic program for the first time, learned knowledge is extracted from the CDCL solver in the form of integrity constraints in ASP syntax. A subset of these integrity constraints is then used to enrich the program.

To this end, this thesis presents a study that covers randomly generated instances of four typical planning problems and three graph problems. While solving the instances, all learned conflict constraints are recorded. The instances are then solved for a second time, this time enriching their original programs with a subset of the logged conflict constraints. Then, the relation between the solving times without and with knowledge feedback indicates how knowledge feedback affects the solver's performance.

In this study, a particular point of interest is the way in which conflict constraints are designated for feedback. Instead of always feeding back the conflict constraints in their entirety, only a subset of those may be selected for feedback—for instance the conflict constraints recorded last or those containing the least number of literals. To analyze the effect of such selection

methods, the study tests various *quantities* of designated conflict constraints as well as multiple *selection orders*.

As the results show, modern solvers indeed profit from reused learned knowledge in many cases. Furthermore, certain selection orders appear to be more beneficial than others.

STRUCTURE This chapter provides a detailed analysis of the impact of reusing knowledge through direct feedback. After describing the examined problems (Section 4.1) and the experimental design (Section 4.2), this chapter lists the tested hypotheses (Section 4.3). Section 4.4 presents the results in detail, broken down by problem. Finally, Section 4.5 concludes this chapter with a discussion of the results.

4.1 ANALYZED PROBLEMS AND FACTORS

PROBLEMS This study covers seven problems: Ricochet Robots, Labyrinth, Hanoi Tower, Solitaire, Hamiltonian Cycle, Graph Coloring, and Knight Cycle (also known as *knight tour with holes*).

Except for the well-known Hamiltonian Cycle problem, they are all part of the official problem suite of the 2013 ASP Competition.[1] The former four encodings are typical planning problems, as they incorporate time steps and actions, while the remaining three problems are graph problems.

INSTANCES The benchmark set comprises 700 random, satisfiable instances with baseline solving times of 3–10 seconds.

Creating the instances involved manually finding an appropriate base configuration (for example, a suitable number of vertices and edge density in Hamiltonian Cycle). For the four planning problems, a fixed maximum plan length (horizon) was selected. Using the base configuration, a custom script created random initial states and goal states (where applicable). Fi-

[1] Problem descriptions can be found on the official website of the 2013 ASP Competition, `https://www.mat.unical.it/aspcomp2013/OfficialProblemSuite`

nally, the script measured the solving time of each candidate with Baseline Clasp, and rejected all the instances that took less than 3 or more than 10 seconds to solve. With Solitaire, nonsquare boards were also allowed to be generated to obtain enough distinct instances.

METRIC A custom benchmark procedure measures the solving time of each instance in two different runs with clasp: once with the original instance for reference (the baseline solving time) and once after enriching the instance through direct knowledge feedback. The ratio between both solving times indicates whether enabling feedback was beneficial or not.

The measured solving times are always the times required to find the first answer set, as reported by clasp. The benchmark procedure sets a solving time limit of 600 seconds and uses a *penalized average runtime* (PAR-10) [17] to handle timeouts. For reasons of comparability, all solving time measurements are performed with Baseline Clasp, and Feedback Clasp is only invoked to extract knowledge.

FACTORS Except for small instances, it is rarely sensible to feed all learned conflict constraints back to the solver. For this reason, the feedback mechanism may select a specific subset of the recorded conflict constraints instead. The *feedback selection method* specifies how this subset is obtained.

In addition to evaluating knowledge feedback in general, this study analyzes whether changing the feedback selection method has an effect on solving performance. In this study, feedback selection is controlled with two factors: the *number of selected conflict constraints* and the *order of selection*.

The number of selected conflict constraints varies from 8 to 16 384, with a factor of $\sqrt{2}$ between two consecutive steps—altogether 23 possible numbers of conflict constraints. The order of selection is one of the following six: First and Last select the n constraints that have been recorded first or last, respectively. Shortest and Longest stand for the n constraints containing the fewest or most literals. Lowest LBD and Highest LBD select n constraints having the lowest or highest *literals blocks distance* (LBD) [1].

EXPERIMENTAL CONDITIONS In the baseline condition, the solving time of an instance was recorded without any feedback, that is, with the original, unmodified instance. The feedback conditions are all combinations of numbers of selected conflict constraints and selection orders. Hence, there are $23 \times 6 = 138$ different feedback conditions.

Per instance and experimental condition, a single set of measurements was performed (no repetitions).

4.2 EXPERIMENTAL DESIGN

Given an instance I, the benchmark procedure first measures the reference solving time of I with Baseline Clasp (the baseline condition).

After that, the procedure extracts all the learned conflict constraints (∇) from solving the instance I. For this purpose, Feedback Clasp is called with the command-line arguments shown at the end of Section 3.2. Feedback Clasp is stopped after finding the first answer set (because conflict constraints learned after this point may exclude earlier answer sets) or after a timeout of 600 seconds. In this case, the benchmark continues with the already recorded conflict constraints only.

Next, the benchmark procedure repeats the following steps for each number of selected conflict constraints n and selection order o:

1. Sort ∇ according to o.
2. Select n (pairwise nonsubsuming) conflict constraints (∇_f).
3. Extend I with ∇_f (I_f).
4. Measure solving time of I_f with Baseline Clasp.

The first two items implement the respective feedback selection method, making use of the postprocessing steps explained in Section 3.3. To make the selection orders unambiguous, the conflict constraints in ∇ are sorted chronologically if they rank equally with respect to the chosen sort key.

Sometimes, there are less than n conflict constraints available (especially, after removing subsumed conflict constraints). In this event, the benchmark tool uses only the available conflict constraints and proceeds unchangedly.

After selecting a subset ∇_f of the conflict constraints ∇ for feedback, the benchmark procedure appends them to I and measures the solving time of the enriched problem instance I_f. The procedure then continues with the next experimental condition.

Having completed all measurements, an evaluation procedure computes the ratios of the measured solving times to the respective baseline measurements. It then groups the results by problem and experimental condition and aggregates the solving time ratios with the geometric mean (see Section 2.4). Since the limit of detection is 0.001 seconds, measurements of 0.000 s are replaced with 0.0005 s (half the precision) to avoid problems with the geometric mean caused by multiplications with zero.

BENCHMARK ENVIRONMENT The benchmarks ran on a PC running Arch Linux with an Intel Core i7-4790K (4.4 GHz) and 16 GB of RAM (1600 MHz, CL7 timings). Clasp 3.1.1 served as Baseline Clasp and as the foundation of Feedback Clasp, both in conjunction with the grounder gringo 3.0.5. The benchmark jobs used all CPUs exclusively (no parts of the benchmark ran in parallel) to avoid memory and cache interferences.

4.3 HYPOTHESES

This study tests three hypotheses concerning the number of selected conflict constraints, the nature of the problem, and the chosen selection order.

SOLVING TIME CURVE CDCL solvers benefit from enriching instances with conflict constraints they learned in previous runs. However, solvers have to consider the added conflict constraints during unit propagation. Because of this, direct knowledge feedback gradually slows down the solving process from a certain point on. Eventually, huge numbers of additional conflict

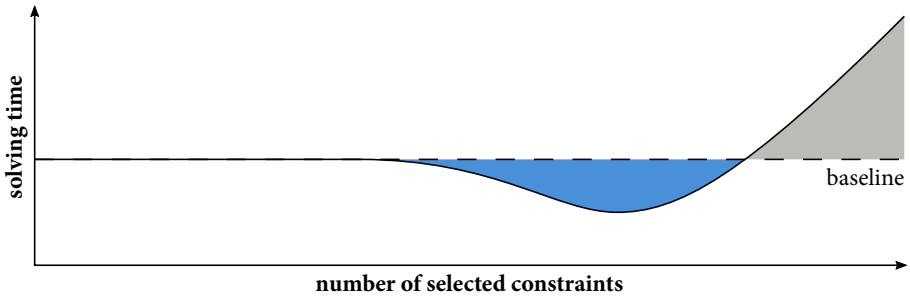


Figure 4.2: Hypothesized solving time curve: Providing a CDCL solver with a certain number of learned conflict constraints helps it solve instances faster (blue). However, the more the instances are enriched, the longer the solver's unit propagation takes. At some point, adding constraints will thus lead to higher solving times than without feedback (gray).

constraints lead to higher solving times than the original instance requires. Figure 4.2 sketches the expected curve.

PLANNING PROBLEMS VS. GRAPH PROBLEMS Planning problems benefit more from reusing learned knowledge than graph problems. The reasoning behind this assumption is that planning problems appear to have more structure that can be exploited by the solver than abstract problems. Thus, learned knowledge might be more useful in this case.

SELECTION ORDERS Last is more useful than First because conflict constraints recorded later potentially incorporate information contained in earlier ones. Between Shortest and Longest, the former performs better, as nogoods with few literals require less decisions to become unit. Lowest LBD is more beneficial than Highest LBD, since high-LBD conflict constraints are related to many decision levels, making it harder for them to become unit. The latter two hypotheses stem from empirical results obtained by Audemard and Simon [1], the authors of the *Glucose* SAT solver.

4.4 RESULTS

> For reasons of clarity, this section highlights and discusses only the most interesting results. More results are shown in Appendix A.

The results show that in many experimental conditions, clasp benefits from direct knowledge feedback. For each problem, Figure 4.3 shows the results of the selection order that led to the lowest average solving time. Throughout the entire benchmark, not a single timeout occurred.

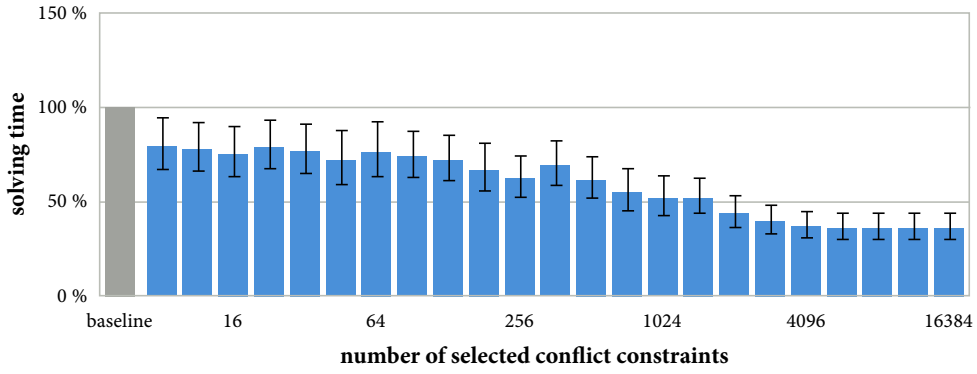
In the cases of Ricochet Robots, Labyrinth, and (to a lower extent) Hamiltonian Cycle, adding conflict constraints continuously reduced the solving time, before it stagnated at some point. This effect can also be observed with the selection orders not shown in Figure 4.3. In the best configurations, solving times were decreased by more than 50 %.

With the right number of selected conflict constraints, Graph Coloring instances were also solved faster thanks to direct knowledge feedback. However, enriching the instance with too many conflict constraints eventually decreased the performance gain.

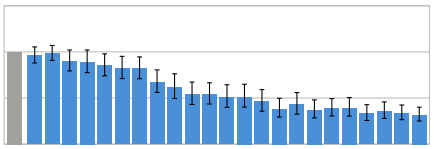
Solitaire and Hanoi Tower are the only two problems where direct knowledge feedback did not have a pronounced effect on the solving time.

The drastic decrease of solving time with Knight Cycle (visible in all experimental conditions) is most likely not related to direct knowledge feedback. As Table 3.2 shows, exchanging the resolution scheme leads to a considerable decrease of the number of conflicts with Knight Cycle. Thus, it should be assumed that the generated instances are special cases, where already small modifications of the solver's default search pattern lead to a solution quickly. For this reason, the Knight Cycle instances are not further detailed in the following results.

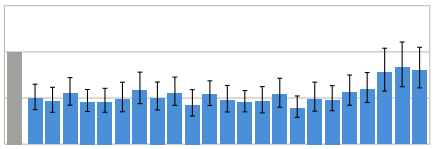
The issue concerning Knight Cycle reflects that answer set solvers are scarcely robust in terms of solving times. While some solver configurations might find a solution immediately, another search pattern might need to traverse the entire rest of the search space before finding a solution.



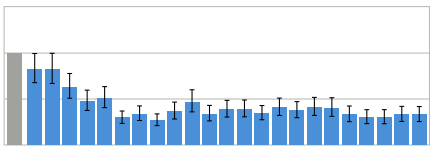
(a) Ricochet Robots, Shortest



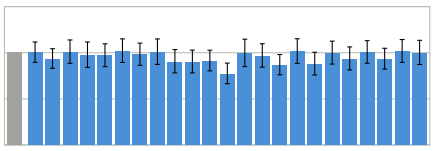
(b) Labyrinth, Highest LBD



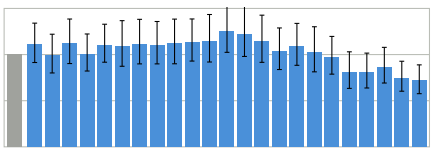
(c) Graph Coloring, Shortest



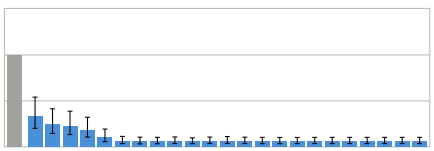
(d) Hamiltonian Cycle, Last



(e) Solitaire, Last



(f) Hanoi Tower, Shortest



(g) Knight Cycle, Lowest LBD

Figure 4.3: The best selection order for each problem (containing the experimental condition with the lowest average solving time). The x axis denotes the number of selected conflict constraints and the y axis shows the solving time relative to the baseline (gray bar). The error bars are 95 % confidence intervals. No timeouts occurred during the entire benchmark.

Having showcased the best configurations for each problem, Sections 4.4.1 to 4.4.3 analyze whether the results reflect the hypotheses from Section 4.3. After that, Section 4.4.4 explores alternative metrics. The results are eventually discussed in Section 4.5.

4.4.1 SOLVING TIME CURVE

In accordance with the hypothesized curve sketched in Figure 4.2, the measured solving time curves start decreasing at some point in almost all cases. Exceptions are Solitaire and Hanoi Tower, which never appear to have profited from direct knowledge feedback. With most of the other problems and configurations, already small numbers of selected conflict constraints had a positive effect on solving performance.

That is, however, not the case with Hamiltonian Cycle and the Longest or Highest LBD selection order. Here, the solving times first increase, then gradually decrease, and eventually remain at about 35 % of the baseline solving time. This opposes the assumption that small amounts of knowledge feedback are either meaningless or slightly beneficial for the solver.

Furthermore, the results do not confirm the second part of the hypothesis, that solving performance gets worse when selecting too many conflict constraints. Only with Graph Coloring and Hanoi Tower, this effect is barely observable, as the solving time curves rise toward the ending—at least, with some of the selection orders. Hence, adding too many conflict constraints to the instance appears to be of no grave concern in practice up to magnitudes of 10 000 selected conflict constraints.

Considering the diversity of the observed results, it is hardly possible to predict the shape of a solving time curve accurately prior to investigating a particular problem. It seems, however, reasonable to expect some improvements with respect to the solving performance when reusing learned knowledge through direct feedback.

4.4.2 PLANNING PROBLEMS VS. GRAPH PROBLEMS

While direct knowledge feedback was favorable with Ricochet Robots and Labyrinth, it had no mentionable effect on the other two planning problems, Hanoi Tower and Solitaire. Hence, it is misleading to assume that planning problems generally profit from reusing learned knowledge. Furthermore, direct knowledge feedback was more beneficial with the graph problems than hypothesized.

These observations invalidate the hypothesis that planning problems benefit more from direct knowledge feedback than graph problems. The results appear to strongly depend on the particular encoding and to be less related to the character or category of a problem.

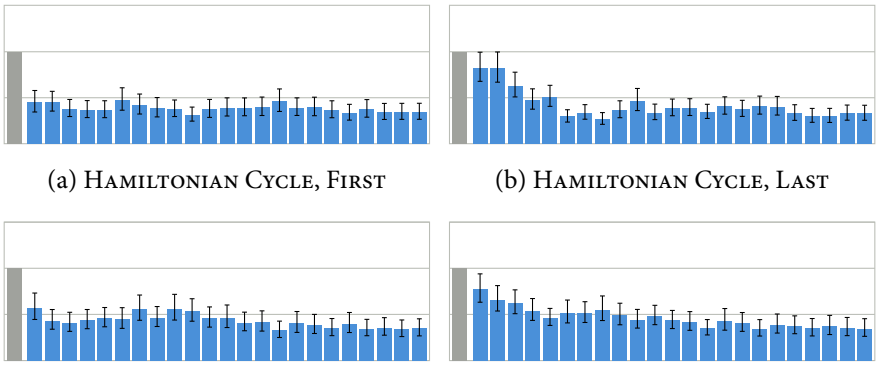
4.4.3 SELECTION ORDERS

FIRST VS. LAST In opposition to the hypothesis, the selection order First led to better solving times than Last whenever there was a visible difference. This effect is noticeable with Hamiltonian Cycle and Labyrinth (see Figure 4.4). In all other cases, both selection orders ranked equally.

SHORTEST VS. LONGEST When selecting conflict constraints in the Shortest order, the results were often similar to those obtained with Last. Though, in support of the hypothesis, the Shortest selection order led to huge improvements with certain problems. This concerns Hamiltonian Cycle and Graph Coloring (see Figure 4.5).

Preferring the Shortest selection order, hence, appears to be sensible, as it never had observable negative effects considering the remaining problems.

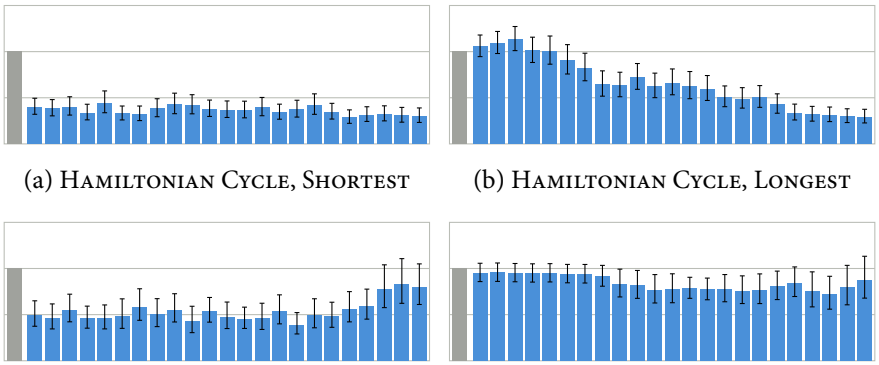
LOWEST LBD VS. HIGHEST LBD As hypothesized, Lowest LBD was generally more helpful than Highest LBD. Labyrinth and Graph Coloring provide empirical evidence (see Figure 4.6) as well as Hamiltonian Cycle. With the other problems, the two selection orders led to similar results.



(a) Hamiltonian Cycle, First (b) Hamiltonian Cycle, Last

(c) Labyrinth, First (d) Labyrinth, Last

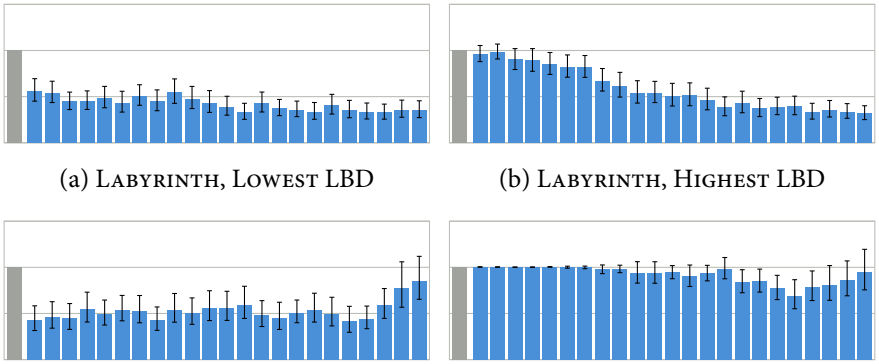
Figure 4.4: With Hamiltonian Cycle and Labyrinth, the selection order First performed better than Last.



(a) Hamiltonian Cycle, Shortest (b) Hamiltonian Cycle, Longest

(c) Graph Coloring, Shortest (d) Graph Coloring, Longest

Figure 4.5: Applying the Shortest selection order was more favorable than Longest. This is particularly visible with Hamiltonian Cycle and Graph Coloring.



(a) Labyrinth, Lowest LBD (b) Labyrinth, Highest LBD

(c) Graph Coloring, Lowest LBD (d) Graph Coloring, Highest LBD

Figure 4.6: With three problems, including Labyrinth and Graph Coloring, selecting conflict constraints by Lowest LBD helped more than selecting by Highest LBD.

4.4.4 OTHER METRICS

Up to now, the analysis only covered solving times relative to a certain baseline. When aggregating the measurements, this metric weighs all instances equally, independent of their size or solving time. Hence, it provides a good indication whether *single* instances benefit from direct knowledge feedback.

In practice, it might also be meaningful to consider the *total* solving time required for an entire test series to complete. For this purpose, the evaluation tool also plotted graphs based on total solving times. While maintaining the essential traits of the solving time curves, the benefit of direct knowledge feedback is less perceptible with this metric. In many cases, the total solving time barely falls below the baseline reference. A possible explanation for this observation is that harder instances—which have more impact on the total runtime—profit less from reusing learned knowledge.

In addition to solving times, the numbers of encountered conflicts and decisions were recorded. These alternative metrics were, however, highly correlated with the relative solving time and led to almost unchanged graphs.

4.5 DISCUSSION

It should be noted that in most cases, recording the learned conflict constraints takes longer than actually solving an instance. This is due to the lower performance of the named-literals resolution scheme, as mentioned in Section 3.2.4. Additionally, the presented feedback mechanism solves each instance twice—once to extract feedback and once with the extracted feedback. Hence, this preliminary study should be considered an assessment of the theoretically possible benefits of direct knowledge feedback rather than an attempt to improve solving performance in practice.

Aside from that, the results indicate that clasp often benefits from reusing learned knowledge when enriching the instance appropriately. Even though there were exceptions, direct knowledge feedback never impaired clasp's solving performance considerably.

Still, the analysis covered only seven different problems, and there might be other, unexplored practical problems that lead to contrary results. Aside from that, further selection methods could possibly yield better results with problems such as Solitaire that did not profit from direct knowledge feedback in the experimental conditions tested so far.

Altogether, the results give good reason to investigate how to further profit from reusing learned knowledge.

5

KNOWLEDGE GENERALIZATION

The results of Chapter 4 indicate that CDCL solvers profit from reusing learned knowledge directly and without further processing. Following that line of thought, this chapter explores unassisted methods for *generalizing* knowledge to make it more useful for solvers and to extend its applicability. With these methods, an algorithm can be developed that extracts learned conflict constraints and generalizes them given a problem instance. The abstracted knowledge gained in the process continuously enriches the instance. Such an algorithm, the *knowledge feedback loop*, is detailed in Chapter 6.

To generalize knowledge, this thesis pursues the two-step approach highlighted in Figure 5.1: The first step produces *candidates* (or hypotheses) for generalized conflict constraints, deriving them from already learned knowledge (after extracting it as in Chapter 3). But these candidate properties need not necessarily remain true anymore and have to be checked for validity in

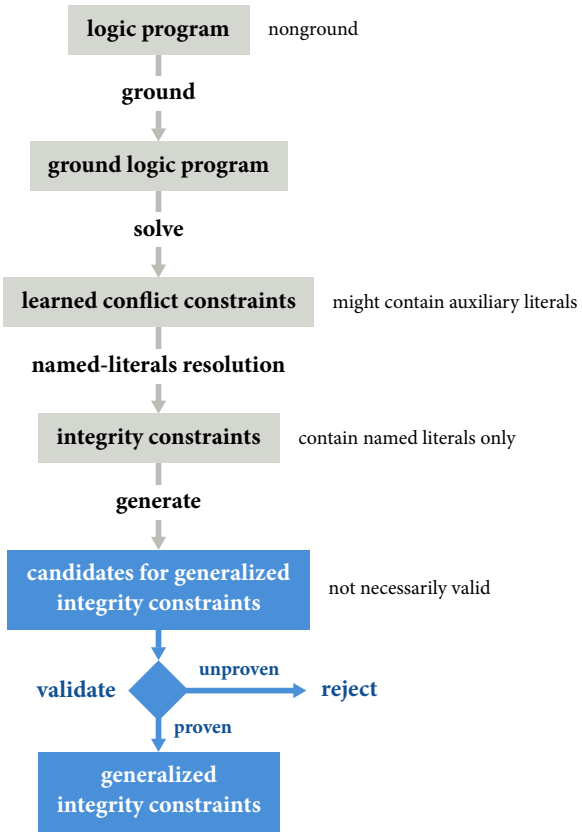


Figure 5.1: Knowledge generalization consists of two steps (blue) in this thesis. Based on previously extracted knowledge, candidates for generalized conflict constraints are generated. A proof method then validates the candidates and rejects all the invalid ones.

the second step. Candidates that cannot be proven valid are rejected.

One method for generating candidate properties is to replace constants in learned conflict constraints with variables. This may mean extending the validity of a conflict constraint from a specific object to all objects of the same domain or from one time step to all points in time. Another possibility is *minimizing* conflict constraints by eliminating as many literals as possible, while ensuring that the conflict constraints stay valid.

Two proof methods for validating the derived hypotheses are presented in this chapter. Both methods build on the concept of searching the space of an instance's solutions for counterexamples to a hypothesis. The first proof method employs of a proof by induction, and the second one is a simplification thereof. However, both methods require knowledge about the meaning of specific predicates in the encoding. For this reason, the discussed implementations only accept ASP facts obtained from encodings in PDDL with the help of plasp (see Section 2.3 for explanations on PDDL and plasp).

STRUCTURE This chapter examines how learned knowledge can be generalized without human support. First, Section 5.1 explains different ways of finding candidates for generalized conflict constraints, taking as a basis learned knowledge extracted from the answer set solver. Then, Section 5.2 explains the basic search for counterexamples and presents two proof methods for testing the candidate properties.

5.1 GENERATING CANDIDATES FOR GENERALIZED KNOWLEDGE

This section explores two ways to generate candidates for generalized conflict constraints from extracted learned knowledge: replacing constants with variables (Section 5.1.1) and minimizing conflict constraints through literal elimination (Section 5.1.2).

5.1.1 REPLACING CONSTANTS WITH VARIABLES

Conflict constraints learned by answer set solvers are ground (variable-free). One method for deriving candidate properties is to replace constants in the conflict constraints with *variables* covering the entire respective domains.

For instance, consider the following (ground) conflict constraint that the solver might have learned with a Ricochet Robots instance:

```
:- not go(red, up, 3), go(red, up, 4), not go(red, left, 5).
```

While this conflict constraint is specific to a particular robot (`red`), it might also be valid for all the other available robots:

```
:- robot(R),
   not go(R, up, 3), go(R, up, 4), not go(R, left, 5).
```

This is a candidate for a generalized conflict constraint and does not necessarily hold anymore. Whether it is still valid needs to be checked separately (Section 5.2 explains suitable methods). Another candidate may be obtained by extending the validity of a direction constant to any possible direction:

```
:- direction(D),
   not go(red, D, 3), go(red, up, 4), not go(red, left, 5).
```

These two examples extend the scope of constraints to the entire domains of *objects*. Clearly, there are many other possibilities of generating candidate properties in a similar fashion.

Aside from objects, many encodings such as Ricochet Robots specify a *time domain* to express the chronology of actions. In the above case, the third parameter of the go predicate determines the time step at which the robot moves. Hypothetically, the conflict constraint might be valid for any sequence of points in time:

```
:- time(T), time(T + 1), time(T + 2),
   not go(red, up, T), go(red, up, T + 1),
   not go(red, left, T + 2).
```

The time domain is very convenient when it comes to checking candidate properties, since it enables a validation method employing a proof by induction (Section 5.2.2). For this reason, the present thesis focuses on generalizing conflict constraints over the time domain rather than object domains.

5.1.2 MINIMIZING CONFLICT CONSTRAINTS

A second method for making knowledge more general attempts to find a minimum subset of a learned conflict constraint that is still valid. With integrity constraints, this means eliminating as many literals as possible. The motivation for minimizing conflict constraints stems from the observation that in manual experiments, many of the literals (often all but two or three) could be eliminated without making the conflict constraints invalid.

Taking again the learned conflict constraint from the last section,

```
:- not go(red, up, 3), go(red, up, 4), not go(red, left, 5).
```

a thinkable candidate is that the conflict constraint still holds after eliminating, for example, the last literal:

```
:- not go(red, up, 3), go(red, up, 4).
```

Assuming that some validation method found this *minimized* conflict constraint to be still valid, a further candidate property would be:

```
:- not go(red, up, 3).
```

A minimization procedure can perform this task automatically by eliminating literal by literal. Whenever the conflict constraint becomes invalid, the procedure reverts the last elimination and continues with the next literal. The minimization procedure used in this thesis always proceeds from left to right. However, when using another strategy, the procedure might find a different minimum, since the minimum subset is not necessarily unique [24].

5.2 VALIDATING CANDIDATE PROPERTIES

Having generated candidates for generalized conflict constraints, these need to be checked for validity before they can be used to enrich instances. This section presents two proof methods that are able to validate candidate properties through automated proofs. The two methods employ a simple search for counterexamples, which is detailed in Section 5.2.1. Based on that, Section 5.2.2 shows a proof method that involves a proof by induction. Then, Section 5.2.3 describes a simplification of this method that reduces the induction to a modified version of the proof by counterexample. Finally, Section 5.2.4 discusses the benefits and limitations of the two proof methods.

5.2.1 SEARCH FOR COUNTEREXAMPLES

In order to validate candidate properties, proof methods need to recognize whether the properties are *invariant* for all the stable models of a problem instance. Instances should only be enriched with invariants, since actual stable models might be eliminated otherwise.

To prove a property to be invariant, a classic indirect technique is to prove that there are no *counterexamples*. A counterexample is a stable model that violates the property. If there are *provably* no counterexamples, the property must be an invariant of the tested instance.

The process of searching for counterexamples can be automated with the help of an ASP solver. The essential idea is to add the negation of the hypothesized property to the answer set program of the instance. This change eliminates all the stable models that satisfy the hypothesis. The remaining solutions of this proof returned by the solver are only the stable models that violate the hypothesis—that is, all the counterexamples.

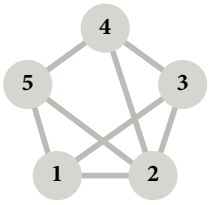
Hence, checking the validity of candidate properties boils down to solving a slightly extended version of an instance's answer set program. A candidate property is valid if the solver does not find any stable models (counter-

Listing 5.1: ASP problem encoding of Graph Coloring.

```
1  color(red; green; blue).
2
3  % Choose exactly one color per vertex
4  1 {chosenColor(V, C) : color(C)} 1 :- vertex(V).
5
6  % Make graph undirected
7  edge(V2, V1) :- edge(V1, V2).
8
9  % Adjacent edges must not have the same color
10 :- edge(V1, V2), chosenColor(V1, C), chosenColor(V2, C).
```

Listing 5.2: Problem instance of Graph Coloring.

```
1  vertex(1..5).
2
3  edge(1, 2).    edge(1, 3).    edge(1, 5).
4  edge(2, 3).    edge(2, 4).    edge(2, 5).
5  edge(3, 4).    edge(4, 5).
```



examples). Search can be terminated directly after finding the first stable model, in which case the hypothesis is invalid. However, to prove that there are no counterexamples, the solver must have searched the whole search space to not miss any stable models.

EXAMPLE To illustrate the procedure, consider the encoding of Graph Coloring in Listing 5.1 and the example instance in Listing 5.2. Solving this instance with clasp yields six stable models.

An invariant of Graph Coloring is: »From three pairwise connected vertices, one is red, one green, and one blue.« To test this property, it is first translated to ASP syntax (lines 1–7 in Listing 5.3). Then, an integrity constraint is added that forms the negation of the tested hypothesis (line 10). In this way, the solver only returns counterexamples violating the hypothesis.

Listing 5.3: Rules added to Listing 5.1 to test the hypothesis: »From three pairwise connected vertices, one is red, one green, and one blue.«

```
% Auxiliary predicate
colorMissing :- edge(V1, V2), edge(V2, V3), edge(V3, V1),
                color(C), not chosenColor(V1, C),
                not chosenColor(V2, C),
                not chosenColor(V3, C).

% Hypothesis
hypothesis :- not colorMissing.

% Find counterexamples only
:- hypothesis.
```

When solving the instance together with the program in Listing 5.3, there are no stable models anymore—thus, the property is indeed invariant (at least with respect to this instance).

As a negative example, consider the inverse hypothesis: »Each subgraph with exactly one red, one green, and one blue vertex is complete.« Applying the same procedure as above results in the rules depicted in Listing 5.4. Solving the original answer set program with this extension yields six counterexamples. Consequently, it is not an invariant and rejected.

LIMITATIONS The search for counterexamples is universal in that it can test candidate properties without specific knowledge about the problem. However, it has two major limitations.

First, its results are instance-specific. If the hypothesis from Listing 5.4 was tested against the instance shown in Listing 5.5, the proof method would not find any counterexamples. The property would, hence, be considered an invariant with respect to this instance (which is correct, in fact), even though it is rejected with the instance from Listing 5.2.

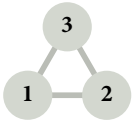
Second, to prove invariants to be indeed invariant, the solver needs to

Listing 5.4: Rules added to Listing 5.1 to test the hypothesis: »Subgraphs with exactly one red, one green, and one blue vertex are complete.«

```
% Auxiliary predicates
rgb(V1, V2, V3) :- chosenColor(V1, red),
                   chosenColor(V2, green),
                   chosenColor(V3, blue).

rgbUnconnected :- rgb(V1, V2, V3), not edge(V1, V2).
rgbUnconnected :- rgb(V1, V2, V3), not edge(V2, V3).
rgbUnconnected :- rgb(V1, V2, V3), not edge(V3, V1).

% Hypothesis
hypothesis :- not rgbUnconnected.

% Find counterexamples only
:- hypothesis.
```

Listing 5.5: Another problem instance of Graph Coloring.

```
vertex(1..3).

edge(1, 2).    edge(1, 3).    edge(2, 3).
```



search the entire space of possible stable models of the problem instance (to be certain that there are no counterexamples at all). With big instances, this is impracticable—especially when checking many candidate properties.

Since in general, planning problems are exponential in the number of fluents, search might also become infeasible. Additionally, plans must be limited to a specific maximum length (the horizon) when solving with solvers like clasp. Thus, the results are not universal anymore and the proof can become very time-consuming when testing with high maximum plan lengths.

5.2.2 PROOF BY INDUCTION

Schiffel and Thielscher presented another automated proof method that involves answer set programming [28]. Their approach is tailored to the *game description language* (GDL) [20] and proves game-specific properties related to single game states through a proof by induction. For instance, such a property related to chess is: »There is at most one pawn in each cell.«

This kind of induction is performed over the temporal domain. Therefore, the method is restricted to proving whether properties hold across all time steps. The proof by induction consists of the two typical steps: First, prove that the hypothesis holds for the game's *initial* state (induction base). Then, test that whenever the hypothesis holds for an *arbitrary* state, it also holds for any *subsequent* state (induction step). If both the induction base and step are proven, the property is an invariant of the game instance.

To test the induction base, the proof method translates the game rules to ASP's modeling language and establishes the instance's initial state. Then, it uses an answer set solver to find counterexamples, analogous to the method described in Section 5.2.1.

For the induction step, the proof method works slightly differently:

1. Nondeterministically build an initial state (instead of the instance's actual initial state). This concept is called a *state generator*.
2. Nondeterministically apply an action, producing a successor state.
3. Reject solutions if the initial state does not satisfy the hypothesis.
4. Reject solutions if the successor state satisfies the hypothesis.

What remains are all counterexamples to the induction step—answer sets in which the hypothesis is fulfilled for a certain state, but no more after applying a particular action. Again, if the solver finds no counterexamples, the induction step is eventually proven.

In their follow-up work, Thielscher et al. extended the proof to invariants referencing sequences of states [30, 15], for example: »If the black player moves a pawn this turn, then the white player moves a pawn next turn.« Supporting this additional type of invariants requires only a small change: Instead of choosing *one* successor state, a *sequence* of successor states is generated that spans as many states as the hypothesis references. Then, the induction base and step are performed as before.

With some adaptations, the extended proof method is capable of validating conflict constraints obtained from generalization over the time domain, as explained in Section 5.1.1. Despite this ability, the method has a drawback—to implement the generators for the induction step, the proof method needs to know which predicates denote mutable state properties (fluents) and actions. ASP's modeling language does not provide such information in general, in contrast to the GDL and also PDDL. For this reason, the proof method by Thielscher et al. cannot be simply applied to pure specifications in the modeling language of ASP.

Nonetheless, restricting the input to GDL or PDDL specifications and generalizing candidate properties over the temporal domain allow for implementing the generalized knowledge feedback loop. In the following, this section shows in detail how these properties are proven by induction.

The proof by induction is translated to ASP facts obtained from PDDL specifications with plasp (see Section 2.3.2). In this way, not just games can be adressed but planning problems in general.

BASIC DEFINITIONS AND RULES Suppose that the proof method needs to test the following candidate property:

```
:- time(T), time(T + 1), time(T + 2),
   not holds(f1, T), apply(a, T + 1), holds(f2, T + 2).
```

As with the simple search for counterexamples, it does this by constructing special programs (one for the induction base and one for the induction step) and feeding them to a CDCL solver, together with the problem instance.

Before the actual proof, both parts of the induction introduce the constant `degree` and the predicate `hypothesisConstraint` as follows:

```
#const degree=2.
hypothesisConstraint(T) :-
    time(T), time(T + 1), time(T + 2),
    not holds(f1, T), apply(a, T + 1), holds(f2, T + 2).
```

The *degree* is the range of time points spanned by the hypothesis (2 in the case of the example conflict constraint). With the predicate `hypothesisConstraint`, the answer set programs can easily check whether the hypothesis is satisfied for a state at a specific time step.

Apart from these definitions, the proofs for the induction base and step both need to respect the planning problem's rules. For this reason, they add another partial answer set program, shown in Listing 5.6. This program—essentially a part of the meta encoding in Listing 2.7—lets the solver choose actions, check their preconditions, and apply their effects. Along with a definition of an initial state, the meta encoding would generate all legal action sequences (up to a certain length) and the resulting states.

INDUCTION BASE Listing 5.7 contains the core of the induction base proof—establishing the initial state at time step 0 (line 2) and rejecting solutions if the candidate property is satisfied when applied to time step 0 (line 5).

By concatenating the three parts of the answer set program (the definitions, the meta encoding, and the code specific to the induction base), the proof method tests the induction base by searching for counterexamples. If the solver finds no solutions, the induction base is proven and the proof method proceeds with the induction step. Otherwise, the hypothesis could not be proven and is rejected.

INDUCTION STEP As mentioned earlier, a state generator is used to produce arbitrary initial states within the induction step. This is realized by Listing 5.8, which nondeterministically chooses the fluents that shall hold in the initial state (line 5). The induction step proof rejects solutions in two cases—if the

Listing 5.6: Meta-encoding part of the induction proof, shared by the induction base and the induction step.

```
% Degree of the hypothesis
time(0..degree).

% Perform actions
1 {apply(A, T) : action(A)} 1 :- time(T), T > 0.

% Check preconditions
:- apply(A, T), demands(A, F, true), not holds(F, T - 1),
   time(T), time(T - 1).
:- apply(A, T), demands(A, F, false), holds(F, T - 1),
   time(T), time(T - 1).

% Apply effects
holds(F, T) :- apply(A, T), adds(A, F), action(A), time(T).
del(F, T) :- apply(A, T), deletes(A, F), action(A), time(T).
holds(F, T) :- holds(F, T - 1), not del(F, T),
               time(T), time(T - 1).
```

Listing 5.7: Part of the answer set program to prove the induction base.

```
% Establish the initial state
holds(F, 0) :- init(F).

% Reject if the initial state satisfies the hypothesis
:- not hypothesisConstraint(0).
```

Listing 5.8: Part of the answer set program to prove the induction step.

```
1 % Base case has <degree> steps, induction step needs one more
2 time(degree + 1).
3
4 % Generate a state nondeterministically
5 {holds(F, 0)} :- fluent(F).
6
7 % Reject if the initial state does not satisfy the hypothesis
8 :- hypothesisConstraint(0).
9
10 % Reject if the successor state satisfies the hypothesis
11 :- not hypothesisConstraint(1).
```

initial state (time step 0) does not satisfy the hypothesis (line 8) or if the successor state (time step 1) satisfies the hypothesis (line 11).

To apply the candidate property to the successor step, it is shifted by 1 with `hypothesisConstraint(1)`. For this to work, the induction step requires one more time step than the induction base (see line 2).

The proof method performs the induction step by feeding the combined answer set program (definitions, meta encoding, and induction step program) to the solver, again to search for counterexamples. If both the induction base and step lead to no stable models, the candidate property is successfully proven by induction. The property can therefore be used to enrich the problem instance in the form of an integrity constraint in ASP syntax.

OPTIMIZATION Up to now, the state generator always considers all the fluents defined by the problem specification to construct initial states. As an optimization, the fluents used to generate states can be chosen from a subset instead, the *fluent closure* (see Listing 5.9). The output of this positive-logic program is a superset of all fluents reachable from the initial state.

To use the fluent closure, line 5 in Listing 5.8 is replaced by:

```
{holds(F, 0)} :- fluentClosure(F).
```

Listing 5.9: Encoding of the fluent closure, a positive-logic encoding building a subset of all states reachable from the initial state.

```
1 % Iteratively build the fluent closure using forward chaining
2 fluentClosure(F) :- init(F).
3 fluentClosure(F1) :- action(A), adds(A, F1),
4                      fluentClosure(F2) : demands(A, F2, true).
```

Listing 5.10: Part of the answer set program used with a simplification of the proof by induction.

```
1 % Generate a state nondeterministically
2 {holds(F, 0)} :- fluent(F).
3
4 % Reject if the initial state satisfies the hypothesis
5 :- not hypothesisConstraint(0).
```

5.2.3 SIMPLIFIED PROOF METHOD

The present thesis introduces a simplification of the proof by induction that still takes benefit of the state-generator mechanism. However, this proof method replaces the two induction steps with a single search for counterexamples, similar to the one explained in Section 5.2.1.

Listing 5.10 shows the essential part of the simplified proof method. As with the proof by induction, a nondeterministic initial state is generated (line 2). But instead of performing the induction, stable models of this program are eliminated if the generated initial state supports the hypothesis (line 5). In this way, counterexamples to the hypothesis are found directly, without the need of a second step.

The simplified proof method reuses the meta-encoding part shown in Listing 5.6 and the definitions for `hypothesisConstraint` and `degree` from Section 5.2.2. Additionally, the optimization of using the fluent closure is compatible with this proof method.

Listing 5.11: Manually crafted properties for Blocks World. With the proof by induction, none of these properties could be proven alone. However, the conjunction of all candidates is provable.

```
1  :- holds(holding(X), T), holds(ontable(X), T).
2  :- holds(holding(X), T), holds(on(Y, X), T).
3  :- holds(holding(X), T), holds(clear(X), T).
4  :- holds(holding(X), T), holds(on(X, Y), T).
5  :- holds(ontable(X), T), holds(on(X, Y), T).
6  :- time(T), typedobject(block(X)),
7     2 {holds(on(X, Y), T) : typedobject(block(Y))}.
8  :- holds(clear(X), T), holds(on(Y, X), T).
9  :- holds(holding(X), T), holds(handempty, T).
10 :- time(T), 2 {holds(holding(X), T) : typedobject(block(X))}.
11 :- time(T), typedobject(block(X)),
12    2 {holds(on(Y, X), T) : typedobject(block(Y))}.
```

5.2.4 DISCUSSION

As mentioned before, the implementations of both proof methods require problem specifications to be written in PDDL (and then translated to ASP facts). Another disadvantage of the proof methods is that they are weaker than the complete search for counterexamples from Section 5.2.1. The issue originates from the nondeterministic state generator, which produces many unreachable and even invalid states. Due to this, the proof methods may mistakenly find counterexamples and then reject potentially valid hypotheses.

In a preliminary study, candidate properties for the Blocks World problem were crafted manually (see Listing 5.11). Because of invalid states generated by the proof by induction, none of the candidate properties could be proven individually. Despite of this observation, the conjunction of all ten candidate properties is provable with the proof by induction. In fact, combining only nine of the shown properties always leads to (false) counterexamples. For instance, when testing all properties but the one in line 10 simul-

taneously, clasp returns the following counterexample:

```
holds(holding(c), 0)  holds(holding(d), 0)
apply(putdown(c), 1)  holds(handempty, 1)
holds(ontable(c), 1)  holds(clear(c), 1)
holds(holding(d), 1)
```

This is obviously no actual counterexample because two different blocks may never be held in the hand at the same time in Blocks World (and apart from that, there are multiple other issues).

To overcome this problem, the problem specification could be extended by constraints (such as the ones in Listing 5.11) ensuring that only valid states are generated. PDDL provides no such means, however.

Aside from these restrictions, the two proof methods reduce the validation of candidate properties to comparably inexpensive proofs (for the induction base and step, or the single simplified proof). The reason for this lies in line 2 of Listing 5.6—the search space is limited by the degree of the hypothesis because states after that do not affect the proof result.

In practice, this is often more feasible than performing the pure search for counterexample. Additionally, the two proof methods are independent of a planning horizon, as the number of simulated time steps depends on the degree of a candidate property. Importantly, the induction method can be applied to almost all candidate properties that stem from extracted learned conflict constraints by generalization over the temporal domain.[1] These features render the proof by induction and the simplification suitable for automated knowledge generalization.

However, the proof by induction validates properties against a specific problem instance (because the induction base depends on an initial state). Therefore, knowledge proven by induction with respect to one instance *cannot* be transferred to another instance of the same problem.

[1] Exceptions are hypotheses with plasp's `terminal(T)` predicate, which cannot be simply generalized over the time domain (because the proof methods do not define a goal).

In contrast, when the fluent-closure optimization is disabled, the simplified proof method becomes *independent* of specific problem instances. Then, candidate properties are only proven against the problem encoding along with the domain predicates. Hence, knowledge generalized like this is valid for other instances that have a different initial state or goal situation (but share the same encoding and domain).

Additionally, the simplified proof consists of a single step only. Thanks to this, testing a candidate property is less complex and may require less solver time. Despite the simplification, this proof method is still able to validate the same types of candidate properties as the proof by induction.

On the downside, the simplification might lead to a weaker proof, since hypotheses are tested only against a single state sequence and not two consecutive ones. For example, the simplified proof method is not able to prove the ten properties in Listing 5.11 simultaneously. For this reason, Chapter 6 considers two configurations of the knowledge feedback loop—one employing the inductive proof method and one using the discussed simplification. In the evaluation of the knowledge feedback loop (Section 6.2), the two proof methods are compared empirically.

6

GENERALIZED KNOWLEDGE FEEDBACK

This chapter introduces the *generalized knowledge feedback loop* (abbreviated as the *feedback loop*), an algorithm that autonomously gathers and generalizes learned knowledge. The feedback loop takes the classic learning technique of modern CDCL solvers one step further. Instead of using learned knowledge directly (as in Chapter 4), learned knowledge is strengthened by generalization in an additional step (with the methods from Chapter 5).

Figure 6.1 shows an outline of the feedback loop's workings. The algorithm accepts a problem instance as input, along with its problem encoding. These need to be provided in the form of ASP facts stemming from a PDDL specification by translation with plasp (see Section 5.2.2). With just this input, the procedure begins to extract learned knowledge by solving the problem with a modified solver (as explained in Chapter 3). Then, the algorithm repeats the following steps continuously:

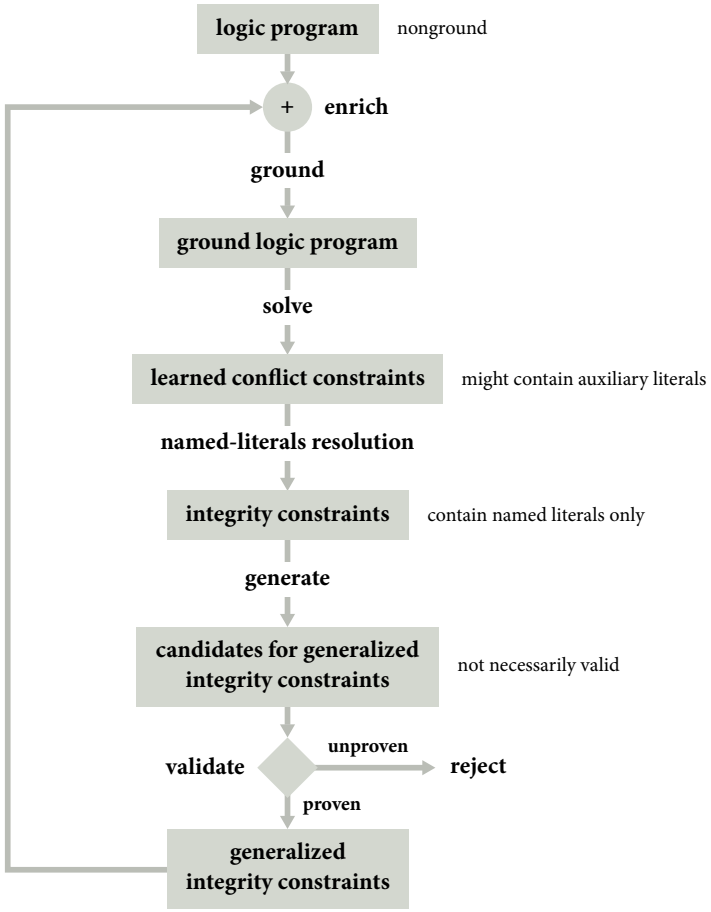


Figure 6.1: The generalized knowledge feedback loop combines all the techniques presented in this thesis: Autonomously and iteratively, it extracts knowledge from the solver, generalizes it, and feeds the generalized knowledge back to the solver.

1. Generate a candidate property by generalizing a learned conflict constraint (Section 5.1).

2. Validate the candidate property with a proof method (Section 5.2).

3. If the property holds, output the generalized conflict constraint and add it to the instance.

An important aspect of this procedure is that the instance is continuously *enriched* with the proven generalized conflict constraints. In this way, they strengthen the algorithm's ability to prove future candidate properties.

Many details of the feedback loop can be implemented in multiple ways. For instance, candidate properties may be validated with various proof methods and additional techniques such as minimization (Section 5.1.2) may or may not be applied. However, when deciding on such aspects of the feedback loop, there are currently no experience values to rely upon.

For this reason, this chapter presents a pilot study that evaluates several configurations of the feedback loop. In the study, a certain number of conflict constraints are generalized first. Then, the constraints are fed back to the solver through *generalized knowledge feedback*. This feedback mechanism corresponds to the one in Chapter 4—instances are solved once without and once with feedback to measure whether generalized knowledge feedback is advantageous for the solver's performance.

The results indicate that many candidate properties can be validated autonomously and that answer set solvers profit from generalized knowledge feedback. While the two tested proof methods produce similar results, only minimization leads to considerable benefits on the solving performance.

In a particularly interesting configuration, candidate properties are validated with the simplified proof but without the fluent-closure optimization. As mentioned in Section 5.2.4, this makes the proof instance-independent. Hence, knowledge learned and generalized in this way can be safely *transferred* to other instances of the same problem encoding and domain. This

might be the most valuable outcome of the present thesis because it enables applications where knowledge is learned and generalized in an offline procedure and reused to solve future instances.

STRUCTURE This chapter presents an implementation of the generalized knowledge feedback loop and is divided in three parts. Section 6.1 describes the design of the feedback loop and possible configurations. Section 6.2 evaluates the implementation. In the study, two proof methods are compared and the effects of minimization and the fluent-closure optimization are analyzed. The chapter concludes by investigating possible applications of the feedback loop in Section 6.3.

6.1 DESIGN AND IMPLEMENTATION

This section first explains the implementation of a simple version of the feedback loop (Section 6.1.1). After that, Section 6.1.2 discusses more details, alternatives to certain aspects, and configuration options of the feedback loop.

6.1.1 BASIC IMPLEMENTATION

Given a problem instance and its encoding in plasp-formatted ASP facts, the feedback loop begins by extracting learned knowledge from that instance. For this purpose, it invokes the modified solver presented in Chapter 3 with the instance facts and plasp's meta encoding (Listing 2.7). The feedback loop always extracts a fixed but configurable number of conflict constraints. If the solver terminates earlier, the feedback loop continues with the conflict constraints obtained up to that point.

The learned conflict constraints are then generalized over the time domain, resulting in candidate properties. These are internally represented by normalizing the time steps, that is, shifting the time points such that the lowest referenced time step becomes 0.

Next, the candidate properties are sorted by degree (in ascending order).

This is sensible because the higher a candidate's degree is, the bigger the space searched by the proof method becomes. Candidates of the same degree are sorted by their number of literals, since minimizing conflict constraints with many literals requires many tests.

The candidates are then iteratively validated with the proof by induction (Section 5.2.2) or the simplified proof (Section 5.2.3). Proven candidates are immediately added to the instance to potentially strengthen future proofs.

Optionally, proven candidates may be minimized (as explained in Section 5.1.2). In this case, literals are removed iteratively from left to right. After eliminating each literal, the shortened candidate property is validated again, using the same proof method as above. If the candidate becomes invalid by removing a literal, the change is undone and the next literal is tested. After testing all literals, the minimized conflict constraint is added to the instance instead of the original one.

The feedback loop terminates after successfully proving a user-defined number of conflict constraints, which are finally returned.

6.1.2 IMPLEMENTATION DETAILS

CONFIGURATION OPTIONS Certain details of the feedback loop may be implemented in multiple ways or require specific constants to be defined. To that end, the present implementation provides several configuration options, which are listed in Table 6.1.

One of the most relevant options is the choice of the proof method used to validate candidate properties (`--proof-method`). The implementation features the proof by induction and the simplification introduced in this thesis. Another important option is whether or not to minimize conflict constraints after successfully proving them (`--minimization-strategy`).

Two already mentioned constants are the number of conflict constraints to extract (`--constraints-to-extract`) and after how many successfully generalized properties the procedure should terminate (`--constraints-`

configuration option	effect
`--proof-method=p`	use proof by induction (`p` = `InductionProof`) or simplified proof (`p` = `SimpleProof`)
`--minimization-strategy=m`	disable minimization (`m` = `NoMinimization`) or enable it (`m` = `LinearMinimization`)
`--fluent-closure-usage=f`	enable fluent closure (`f` = `UseFluentClosure`) or disable it (`f` = `NoFluentClosure`)
`--testing-policy=t`	restart extraction after testing all hypotheses (`t` = `TestAll`) or after successful proof (`t` = `FindFirst`)
`--constraints-to-extract=n1`	extract `n1` $\in \mathbb{N}$ conflict constraints initially
`--constraints-to-prove=n2`	stop after proving `n2` $\in \mathbb{N}$ properties
`--max-degree=n3`	skip hypotheses if degree greater than `n3` $\in \mathbb{N}$
`--max-number-of-literals=n4`	skip hypotheses with more than `n4` $\in \mathbb{N}$ literals
`--extraction-timeout=n5`	set grounding time limit of knowledge extraction to `n5` $\in \mathbb{N}$ seconds
`--hypothesis-testing-timeout=n6`	set grounding/solving time limit to `n6` $\in \mathbb{N}$ seconds when proving properties
`--horizon=n7`	apply a fixed maximum plan length of `n7` $\in \mathbb{N}$ in knowledge extraction

Table 6.1: Configuration options of the presented feedback loop implementation.

`to-prove`). The other configuration options are explained below, where more details about the implementation are provided.

KNOWLEDGE EXTRACTION To extract knowledge, a planning horizon must be specified (`--horizon`). This is required by plasp's meta encoding, which requires an upper bound of the plan length.

After extracting the specified number of conflict constraints, the solver is paused through a POSIX signal (if it did not terminate already). If not enough properties could be proven given the extracted conflict constraints, the solver is restarted to produce more conflict constraints. In the rare case that no proof could be successfully performed, restarting would not help (because the instance is untouched), and the solver is *resumed* instead.

Should restarting the solver with the enriched program produce no more conflict constraints (because a solution is found without encountering conflicts), the feedback loop terminates early. This also happens if grounding takes longer than a specified time limit (`--extraction-timeout`).

KNOWLEDGE GENERALIZATION Hypotheses having a high degree or containing too many literals are skipped because proving and minimizing them takes too long. These thresholds are defined by the user (`--max-degree` and `--max-number-of-literals`).

The current implementation eliminates all `terminal(T)` literals from candidate properties. This is because the proof methods do not define goal conditions, and the `terminal` predicate would be undefined. Hence, there is no simple, meaningful way to generalize these literals. A brief experiment showed that some properties shortened by `terminal` literals are still provable, even though the success rate is lower than with the other candidates.

Additionally, hypotheses are skipped if they are already subsumed by a previously proven generalized conflict constraint. Similarly, the set of successfully generalized candidates is continuously cleaned up by removing all the ones that are subsumed by new proven conflict constraints.

Proofs that take too long are aborted, in which case the tested hypothe-

ses are assumed to be unproven. For this purpose, a configurable time limit (`--hypothesis-testing-timeout`) is imposed on every invokation of gringo and clasp, respectively.

Section 5.2.2 mentioned the fluent closure, an optimization that reduces the set of fluents considered by the state generator. An according option controls whether this optimization is enabled (`--fluent-closure-usage`).

In one scenario, all available candidates are tested before extracting new learned conflict constraints (called the Test All policy). Instead, knowledge extraction may be restarted after each successfully proven candidate (the Find First testing policy). The latter approach contributes generalized conflict constraints to the feedback extraction step immediately—hence, future candidate properties might contain more useful information. The implementation supports both testing policies (with `--testing-policy`).

MINIMIZATION Observations show that by minimization, many proven conflict constraints can be reduced to few literals—often just two or three. For this reason, the minimization procedure of the feedback loop is slightly optimized. After successfully eliminating a literal, the procedure tries to remove the next *two* literals simultaneously. The size of this *elimination window* increases by one again after every successful minimization step. Whenever a minimization step is unsuccessful, the window size resets to 1. Especially with conflict constraints containing many literals, this optimization can save many minimization steps.

6.2 EVALUATION

Section 6.1.2 showed that the presented implementation of the feedback loop can be configured in many different ways. However, there are currently no experience values to rely upon when deciding on such aspects of the feedback loop. This section presents a pilot study to get a first impression of the utility of the feedback loop and to find reasonable configurations.

Apart from gaining general insights, this study analyzes three configuration options in particular. The first one is the choice of the proof method that validates candidate properties—the proof by induction or the simplified proof. Second, this study assesses the effect of minimizing conflict constraints after proving them. As a third configuration option, this study investigates whether disabling the fluent-closure optimization has a negative effect on the feedback loop's utility.

The evaluation procedure is detailed in Section 6.2.1. Then, Section 6.2.2 describes the problem instances and configurations analyzed in this study. Section 6.2.3 finally presents the results and analyzes the impact of the three mentioned configuration options: the proof method, the use of minimization, and the fluent-closure optimization.

6.2.1 EXPERIMENTAL DESIGN

The benchmark procedure is divided in two parts. The first part executes the feedback loop using the implementation from Section 6.1 and writes all the successfully generalized conflict constraints to a separate file.

In all configurations, the following options of the feedback loop are fixed:

```
--testing-policy=TestAll --constraints-to-extract=16384
--constraints-to-prove=1024 --max-degree=10
--max-number-of-literals=50 --extraction-timeout=600
--hypothesis-testing-timeout=10
```

The testing policy is set to Test All because initial experiments proved Find First to be infeasible due to how frequently knowledge was extracted. Additionally, `--horizon` is fixed to the instances' respective minimum plan lengths, which were determined before running the benchmark.

The second part of the benchmark procedure assesses the benefit of generalized knowledge feedback in an offline procedure. For this purpose, the

instance is enriched with the first n generalized conflict constraints recorded in the first part. Then, the solving time of the enriched instance is measured and set in relation to the baseline solving time (obtained with the unmodified instance). Starting with $n = 8$, the benchmark procedure increases n to 1024 in exponential steps, similar to the experimental design in Section 4.2. The planning horizon is set to the same value as specified by `--horizon`.

BENCHMARK ENVIRONMENT The benchmark environment was identical to the one listed in Section 4.2. Again, the benchmark jobs used all CPUs exclusively. The feedback loop was implemented in C++14 with POSIX pipes to communicate with clasp and gringo.

6.2.2 ANALYZED PROBLEMS AND FACTORS

PROBLEMS Five different problems are part of the study: Blocks World, Elevator, FreeCell, Logistics, and Depots. The PDDL problem encodings stem from the *hclasp* benchmark[1] and were transformed to ASP facts with the help of plasp 2.0.0.

The programs were manually extended by `fluent` predicates, which are required by the implemented proof methods. These annotations are not automatically generated by plasp currently, even though plasp provides a respective interface and an implementation would be uncomplicated.

INSTANCES The benchmark comprises 25 instances in total, 5 for each problem. The small size of the test set is a result of the fact that the feedback loop has a comparably long runtime. The instances were partly taken from the hclasp experiments if the baseline solving time was below 10 seconds. Because this yielded only 19 usable instances, another 6 instances were manually created for Logistics and Depots.

[1] The original PDDL and generated ASP encodings can be found on the hclasp website, `http://www.cs.uni-potsdam.de/hclasp/#experiments3`

METRIC One purpose of this study is to estimate the influence of generalized knowledge feedback on the solver's performance. To that end, the second part of the benchmark procedure measures the solving times after enriching the instance with the first n successfully generalized conflict constraints (with multiple tested numbers n, as mentioned earlier). These are then compared to a baseline measurement, which is obtained by solving the original instance. The solving time measurements are performed identically as described in Section 4.1, always using Baseline Clasp.

Additional statistics are recorded while executing the feedback loop (in the first part of the benchmark procedure). This includes the total runtime of the feedback loop, grounding and solving times required to validate candidate properties, the numbers of proven, unproven, and skipped candidates, information about the tested candidates, and the number of literals eliminated through minimization.

FACTORS This study compares the two *proof methods* (Induction Proof and Simplified Proof). Further, the effect of conflict constraint *minimization* is evaluated by enabling or disabling it (Minimization or No Minimization). Third, the study analyzes if disabling the *fluent-closure optimization* leads to a weaker proof (Fluent Closure or No Fluent Closure).

EXPERIMENTAL CONDITIONS To find reasonable configurations of the feedback loop, this study tests all four combinations of proof method and minimization usage. In these four experimental conditions, the Fluent Closure optimization is always enabled.

Selecting the Simplified Proof in conjunction with No Fluent Closure leads to an instance-independent proof (see Section 5.2.4). For this reason, an additional fifth condition is tested, with Minimization enabled.

The number of generalized constraints selected for feedback is varied between $n = 8$ and $n = 1024$ in exponential steps of $\sqrt[4]{2}$. Together with the baseline measurement, this results in a total of 30 solving time measurements per instance and per configuration.

6.2.3 RESULTS

> For reasons of clarity, the results discussed in this section are aggregated over all problems. More detailed results are shown in Appendix B.

With Minimization, generalized knowledge feedback helped reducing the solving time considerably, in some experimental conditions by more than 65 %. Figure 6.2 shows one of the configurations where the solver profited most from generalized knowledge feedback.

On average, validating candidate properties took 148 ms of grounding time and 48 ms of solving time (across the experimental conditions with the Fluent Closure optimization). 59.6 % of the proofs were successful—a high number considering the experiments with manually crafted hypotheses discussed in Section 5.2.4. The feedback loop terminated after an average of 26 minutes. In 7 of the 125 runs, the feedback loop stopped early because of grounding timeouts within the knowledge extraction step. These timeouts occurred in cases where all candidates were already tested and new learned conflict constraints needed to be extracted.

PROOF BY INDUCTION VS. SIMPLIFIED PROOF Both proof methods were well-suited for validating candidate properties. On average, the Proof by Induction was able to successfully prove 60.5 % of all candidate properties (not counting proofs required for minimizing conflict constraints). The Simplified Proof had a success rate of 58.8 %. As Figure 6.3 shows, both proof methods also performed similar with respect to generalized knowledge feedback. With Blocks World and Depots, the results were almost identical between the two proof methods.

However, the Simplified Proof required only 45 % (30 ms) of the solving time that the Proof by Induction needed (67 ms) to validate candidate properties. This observation can be traced back to the fact that the Proof by Induction performs two separate proofs for the induction base and step. Similarly, the grounding time was 93 ms with the Simplified Proof and

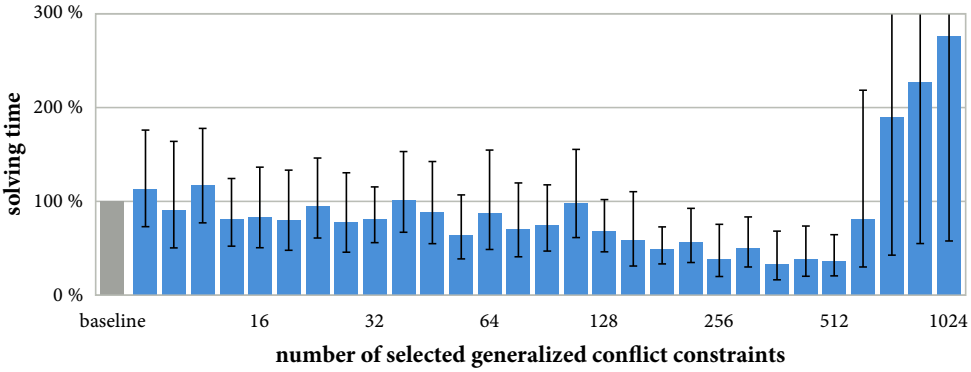
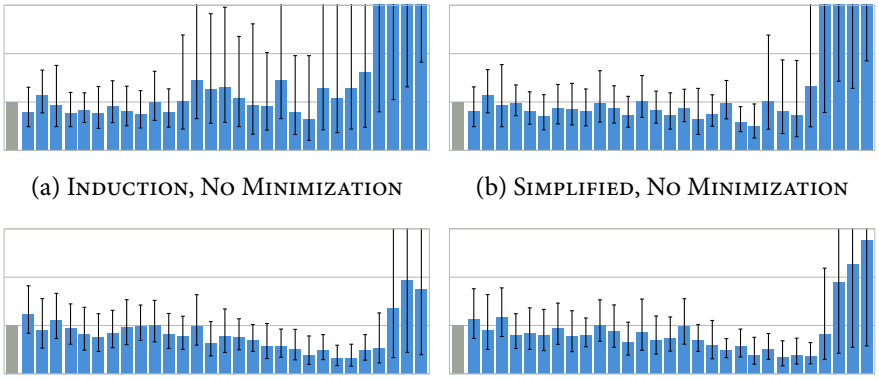


Figure 6.2: One of the best configurations (Simplified Proof, Minimization, and Fluent Closure). The x axis denotes the number of generalized properties added to the instance. The y axis shows the solving time relative to the baseline (gray bar), aggregated over all problems. The error bars are 95 % confidence intervals.



(a) Induction, No Minimization (b) Simplified, No Minimization

(c) Induction, Minimization (d) Simplified, Minimization

Figure 6.3: Comparison of the Proof by Induction (left) and the Simplified Proof (right). Both show similar results, whether No Minimization (top) is applied or Minimization (bottom). However, Minimization leads to considerably reduced solving times. All shown configurations use the Fluent Closure.

205 ms with the Proof by Induction. In turn, the feedback loop had a 48 % lower total runtime with the Simplified Proof.

Aside from these performance considerations, the other recorded statistics evinced no noteworthy differences between the two proof methods.

MINIMIZATION Figure 6.3 also shows that Minimization is generally a better choice than No Minimization. In fact, the conflict constraints generalized with No Minimization hardly reduced the solving times at all.

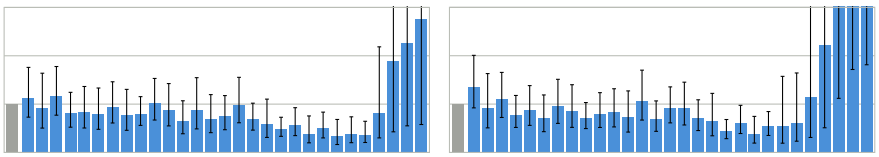
While the feedback loop performed 1546 proofs on average with No Minimization, 11 480 were required when Minimization was enabled (about 7.4 times as many). This difference is reflected by the total runtime of the feebdack loop—on average, it was 4.5 times higher with Minimization.

Through minimization, 57.6 % of all literals in minimized properties were successfully eliminated. The average tested hypothesis contained 8.6 literals before minimization. This coincides with prior observations that many conflict constraints can be reduced to few literals.

With No Minimization, 4.0 % of the candidates were subsumed by other ones. Enabling Minimization increased this amount to 10.5 %. Thus, minimization apparently leads to stronger conflict constraints. This seems to be an explanation of why the solver profited more from generalized knowledge feedback in experimental conditions with Minimization.

DISABLING THE FLUENT CLOSURE Figure 6.4 illustrates the results for the Simplified Proof with Minimization, once with Fluent Closure enabled and once with No Fluent Closure. As the results show, disabling the fluent closure has only little effect on the strength of the proof.

The feedback loop terminated early with all Logistics instances because of grounding timeouts while extracting learned knowledge (after running out of candidates). This is due to the fact that with No Fluent Closure, hypotheses of higher degree were eventually generalized, some of which caused timeouts while grounding the enriched instance. Due to this, the number of timeouts that occurred in the generalized knowledge feedback step also



(a) Fluent Closure (b) No Fluent Closure

Figure 6.4: Disabling the Fluent Closure has little effect in the tested configurations (Simplified Proof with Minimization). The increased solving times at the right end of the x axis are mostly due to timeouts while *grounding* the enriched instances. Affected measurements were penalized with 6000 seconds.

increased from 2.4 % to 5.2 % when disabling the fluent closure, affecting measurements with 362 or more selected generalized conflict constraints.

The amount of successfully proven candidate properties was lower without (30.6 %) than with the Fluent Closure optimization (54.0 %). Conversely, No Fluent Closure led to lower candidate validation times (17 ms for solving, 70 ms for grounding) than when using the Fluent Closure (27 ms for solving, 85 ms for grounding). Apparently, this compensated for the lower proof rate—on average, the feedback loop terminated after 1660 s with No Fluent Closure (instead of 1663 s with Fluent Closure).

Aside from that, generalized knowledge feedback produced comparable results with and without the Fluent Closure.

6.2.4 DISCUSSION

With minimization enabled, generalized knowledge feedback often reduced solving times. Still, some generalized conflict constraints that were proven late in the process seem to have caused timeouts when adding them to the instances. Such issues can be addressed by skipping properties that would result in high grounding times, for instance, with a heuristic or by measuring

the grounding time separately.

When disabling the fluent-closure optimization, the simplified proof is instance-independent (as explained in Section 5.2.4). In this setting, the Simplified Proof becomes interesting because it enables offline learning procedures where knowledge is transferred from one instance to other instances of the same problem encoding. Beyond that, the results show that with such a configuration, knowledge generalization is similarly beneficial for solvers than the other examined configurations. These considerations give rise to the question how knowledge generalization can be exploited in practice.

6.3 PRACTICAL APPLICATIONS

Seeing that enriching instances with generalized knowledge increases solving performance, it would be desirable for practical applications to make use of knowledge generalization. However, running the feedback loop typically requires more time than solving an instance directly. Still, the presented approaches may be employed in offline procedures where knowledge is generalized in a preparation step and preserved to reuse it at a later point.

One such application could reuse generalized knowledge obtained from an instance by *transferring* it to other instances of the same domain and problem encoding. This is already enabled by the implementation with the special configuration mentioned earlier (Simplified Proof, Minimization, No Fluent Closure). For instance, knowledge can be generalized by the feedback loop given a Ricochet Robots instance. With this knowledge, *future* Ricochet Robots instances can be solved faster if the robots' start and goal positions are changed (but not the wall placement or board size).

To analyze the utility of transferring generalized knowledge, an additional experiment was carried out. To this end, copies of the instances from Section 6.2 were created by altering their initial states or goal situations, but keeping the domains unchanged. Then, knowledge was extracted and generalized with the feedback loop, using the original set of instances from Sec-

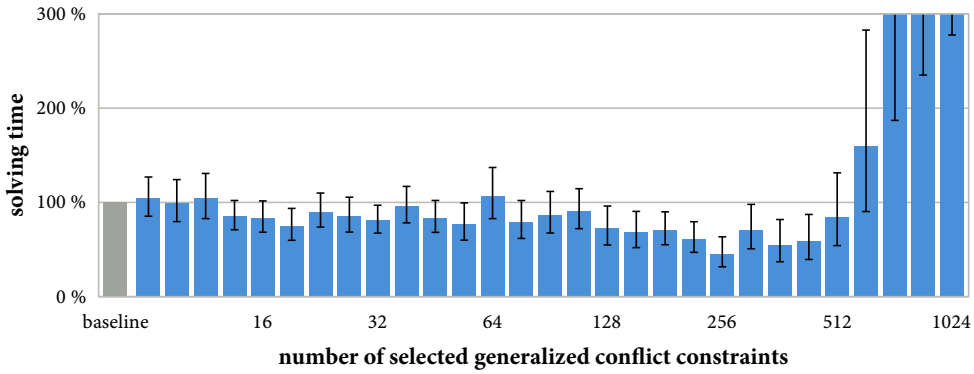


Figure 6.5: In this experiment, knowledge was learned and generalized with the instances from Section 6.2. The generalized conflict constraints were then *transferred* to copies of the instances with a changed initial state or goal situation. Apparently, generalized knowledge is also useful when transferring it to other instances of the same domain, since the results are comparable to Figure 6.2.

tion 6.2. Finally, the effect of transferring generalized knowledge was assessed by gradually enriching the *modified* instances with the generalized constraints obtained with the *original* ones and measuring the solving times.

Figure 6.5 shows the results of this experiment. Surprisingly, generalized conflict constraints appear to be almost as useful when transferring knowledge to other instances as they are when enriching the original instances (compare Section 6.2.3). Consequently, knowledge generalization through the presented feedback loop may indeed be valuable in practice.

This might be particularly interesting for use cases where large numbers of instances need to be solved. There, precomputing a set of generalized conflict constraints might be worthwile, even if the solving times are reduced by just a small extent. Furthermore, for systems with scarce resources, it can be sensible to generalize knowledge beforehand. Even if the offline generalization part takes long, the reduced cost on the live systems might pay off.

Another offline application could learn generalized knowledge with *small* instances in order to apply it to *large* instances later. There, the benefit of reusing knowledge could be even more noticeable, as conflict constraints are generalized faster with small instances, while the impact on large instances could be bigger. To implement this application, support for generalization over *object domains* is necessary, since temporally generalized conflict constraints are still bound to specific object identifiers.

7

CONCLUSIONS AND FUTURE WORK

This chapter summarizes the most important contributions of this thesis—on a high level as well as concerning the implementation—in Section 7.1. Then, related work is discussed in Section 7.2. Section 7.3 completes this thesis with an outline of possible future work.

7.1 CONTRIBUTIONS

The present thesis explored multiple opportunities to reuse learned knowledge and discussed their benefits. The first part analyzed how answer set solvers benefit from reusing learned conflict constraints *directly*. For this purpose, test instances were solved while recording the conflict constraints learned by the solver. Then, the solving time was measured with the original test instances and again after enriching them with selected conflict con-

straints. The results reflected that answer set solvers generally benefit from this type of knowledge feedback.

The second part of this thesis presented unassisted methods for *generalizing* learned knowledge in planning problems. Knowledge generalization was achieved by first producing candidates for generalized conflict constraints and validating these afterward. The candidates were obtained by extending the scope of the learned conflict constraints over the temporal domain. An evaluation of reusing generalized knowledge—similar to the first study with direct knowledge feedback—showed that solvers often benefit from enriching instances with generalized conflict constraints.

Ultimately, this thesis detailed how knowledge can be generalized in such a way that it becomes *independent* of a particular problem instance. This enables offline applications that first record and generalize knowledge and then transfer it to *future instances* of the same domain (but with different initial states or goal conditions) in order to solve them faster. A pilot study showed that instances are indeed solved faster when reusing generalized knowledge obtained from another instance.

On the implementation side, this thesis introduced multiple novel methods and algorithms that aid the process of generalizing knowledge. First, the solver clasp was extended to record learned conflict constraints as rules in ASP syntax in order to reuse them externally. For this purpose, the custom named-literals resolution scheme was developed. This resolution scheme guarantees the recorded conflict constraints to contain named literals only so that the rules are well-formed.

To validate candidates, an automated proof by induction was adopted. Additionally, a novel proof method was introduced—a simplification of the proof by induction that is faster by a factor of about 2, while leading to similar results. Most importantly, this proof method enables knowledge to be generalized in an instance-independent way.

Finally, an algorithm was implemented that automatically extracts learned knowledge and generalizes it using the concepts explained in this thesis.

7.2 RELATED WORK

Concerning the validation of candidate properties, Schiffel and Thielscher observed that many properties could not be proven by themselves, but only together with other properties. To cover such cases, the authors suggest extending their proof method to a *simultaneous induction* [28]. However, testing combinations of properties renders the proof much more complex.

To prove candidate properties to be invariants, Lin presented a method that tests the candidates against a small set of small problem instances by an exhaustive search covering all stable models [19]. Invariants returned by this procedure are not necessarily correct in the general case. Furthermore, the set of tested properties must be manually specified by a user. Similar to Schiffel and Thielscher's proposal [28], Lin suggests testing conjunctions of properties, as they might not be provable alone.

To avoid the manual specification of candidate properties that shall be tested, Li et al. describe a system that *enumerates* all possible 2-literal formulas [18]. The formulas are then automatically tested and, if invalid, iteratively strengthened until they become invariant. By first testing properties against a small set of small problem instances (inspired by Lin [19]), the authors improve the efficiency of falsifying incorrect properties. Still, the method is incomplete despite its soundness.

Invariants discovered like this could be used to strengthen the proof methods of the generalized knowledge feedback loop. Similar to the idea of proving multiple candidates simultaneously, additional invariants might allow certain properties to be proven that could not be proven otherwise.

Rintanen outlined another strategy for discovering invariants within planning problems [25]. It starts with a candidate invariant that contains all 2-literal clauses that are satisfied in the initial state of a problem. Then, clauses are removed progressively if they are violated by the effect of an action.

Rintanen presented a second method for synthesizing invariants [26]. The method starts with a set of candidate properties that are satisfied in the initial

state and iteratively checks that these are not violated by subsequent action. If this is the case, the affected candidate properties are removed from the set and replaced by weaker ones.

A further method for finding invariants in planning problems proposed by Rintanen relies on the *regression operator* [27]. This operator computes the weakest preconditions to a state to guarantee that some property holds in the subsequent state. Also starting with candidate properties that hold in the initial state of a problem instance, the algorithm iteratively eliminates invalid properties. These are detected by applying the regression operator to their negation. If the regression is satisfied, then there is a predecessor state that leads to a violation of the property after applying some action. Eliminated properties are once again replaced with weaker ones (in this case, disjunctions with other properties).

The three algorithms described by Rintanen share two limitations: The methods do not find all invariants and become inefficient when extending them from 2-literal to n-literal invariants, already with $n = 4$.

7.3 FUTURE WORK

Concerning knowledge extraction (Chapter 3), future research may evaluate alternatives to the presented named-literals resolution scheme. Further optimizations of the resolution scheme's implementation might also reduce the runtime of the generalized knowledge feedback loop, in which knowledge extraction is responsible for a considerable amount of time.

As mentioned in Section 5.2.4, the proof methods may be strengthened by manually specifying state constraints, which eliminate invalid states. In this way, the state generators used in the proof by induction and the simplified proof avoid generating such invalid states. Thus, less false counterexamples are potentially found and more candidates can be proven. Instead of manually specifying such state constraints, they may be obtained with one of the discovery methods mentioned in Section 7.2.

Futhermore, proving candidates simultaneously (as suggested by Schiffel and Thielscher [28]) could make the proof stronger in a similar fashion. Further studies can be conducted to find out whether state constraints and simultaneous proofs actually lead to improved validation methods or more useful generalized knowledge.

In this thesis, learned conflict constraints were generalized over the temporal domain. Additionally, generalizing knowledge over object domains may be implemented (see Section 5.1.1). This might make the generalized conflict constraints independent of some domain properties, such as specific object names or instance sizes. Ultimately, knowledge generalized with small instances of a problem might be transferred to bigger ones, possibly reducing the solving times of large instances.

Aside from generalizing conflict constraints over the temporal domain, proven candidates are minimized in a seperate procedure in the presented implementation of the generalized knowledge feedback loop. The minimization technique explained in Section 5.1.2 may be improved further to reduce the necessary number of steps to find the minimum subset of a conflict constraint, making minimization more feasible in practice.

Section 6.1.2 introduced many configuration options of the generalized knowledge feedback loop and evaluated several of the options to find reasonable configurations. In future work, more configurations may be analyzed with respect to different use cases—for instance, to make generalized knowledge more useful or to obtain generalized conflict constraints faster. Furthermore, more problems and instances should be studied to get a deeper understanding of generalized knowledge feedback.

In the evaluation of the feedback loop, several problem instances encountered grounding timeouts after certain generalized conflict constraints with a high degree were added (see Chatper 6.2.3). A future filtering step could detect and eliminate these infeasible properties (for instance, by adding them and testing them with a timeout or a heuristic). Additionally, the implementation of the feedback loop can be improved by incorporating incremental

grounding to avoid grounding similar sets of rules over and over.

For large planning problems, it could be possible to make the generalization step a part of the solving procedure. In other words, a solver could attempt to generalize recently learned conflict constraints in parallel to the actual solving procedure. If validating the candidate properties does not take too long, enriching the currently solved program with additional generalized properties might prune the search space considerably and possibly increase the solving performance.

REFERENCES

[1] Gilles Audemard and Laurent Simon. "Predicting Learnt Clauses Quality in Modern SAT Solvers". In: *Proceedings of the 21st International Joint Conference on Artificial Intelligence*. AAAI Press, 2009, pp. 399–404.

[2] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[3] Roberto J. Bayardo Jr. and Robert C. Schrag. "Using CSP Look-Back Techniques to Solve Real-World SAT Instances". In: *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Conference on Innovative Applications of Artificial Intelligence*. AAAI Press, 1997, pp. 203–208.

[4] Keith L. Clark. "Negation as Failure". In: *Proceedings of the Symposium on Logic and Data Bases*. Springer, 1978, pp. 293–322.

[5] Martin Davis, George Logemann, and Donald Loveland. "A Machine Program for Theorem-Proving". In: *Communications of the ACM* 5.7 (1962), pp. 394–397.

[6] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory". In: *Journal of the ACM* 7.3 (1960), pp. 201–215.

[7] Niklas Eén and Niklas Sörensson. "An Extensible SAT-Solver". In: *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2004, pp. 502–518.

[8] 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.

[9] Martin Gebser et al. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[10] Martin Gebser et al. "Conflict-Driven Answer Set Solving". In: *Proceedings of the 20th International Joint Conference on Artificial Intelligence*. AAAI Press, 2007, pp. 386–392.

[11] Martin Gebser et al. "plasp: A Prototype for PDDL-Based Planning in ASP". In: *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer, 2011, pp. 358–363.

[12] Michael Gelfond and Vladimir Lifschitz. "The Stable Model Semantics for Logic Programming". In: *Proceedings of the 5th International Conference and Symposium on Logic Programming*. MIT Press, 1988, pp. 1070–1080.

[13] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Elsevier, 2004.

[14] Malik Ghallab et al. *PDDL—The Planning Domain Definition Language*. Manual produced by the AIPS-98 Planning Competition Committee. 1998.

[15] Sebastian Haufe, Stephan Schiffel, and Michael Thielscher. "Automated Verification of State Sequence Invariants in General Game Playing". In: *Artificial Intelligence* 187 (2012), pp. 1–30.

[16] 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.

[17] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. "Tradeoffs in the Empirical Evaluation of Competing Algorithm Designs". In: *Annals of Mathematics and Artificial Intelligence* 60.1-2 (2010), pp. 65–89.

[18] Naiqi Li, Yi Fan, and Yongmei Liu. "Reasoning about State Constraints in the Situation Calculus". In: *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*. AAAI Press, 2013, pp. 997–1003.

[19] Fangzhen Lin. "Discovering State Invariants". In: *Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning*. AAAI Press, 2004, pp. 536–544.

[20] Nathaniel Love et al. *General Game Playing: Game Description Language Specification*. Technical report of the Stanford Logic Group at Stanford University (LG-2006-01). 2006.

[21] João P. Marques-Silva and Karem A. Sakallah. "GRASP: A Search Algorithm for Propositional Satisfiability". In: *IEEE Transactions on Computers* 48.5 (May 1999), pp. 506–521.

[22] David G. Mitchell. "A SAT Solver Primer". In: *Bulletin of the EATCS* 85 (2005), pp. 112–132.

[23] Nilan Norris. "The Standard Errors of the Geometric and Harmonic Means and Their Application to Index Numbers". In: *The Annals of Mathematical Statistics* 11.4 (1940), pp. 445–448.

[24] Max Ostrowski et al. "Boolean Network Identification from Multiplex Time Series Data". In: *Proceedings of the 13th International Conference on Computational Methods in Systems Biology*. Springer, 2015, pp. 170–181.

[25] Jussi Rintanen. "A Planning Algorithm Not Based on Directional Search". In: *Proceedings of the 6th International Conference on Principles of*

Knowledge Representation and Reasoning. Morgan Kaufmann, 1998, pp. 617–625.

[26] Jussi Rintanen. "An Iterative Algorithm for Synthesizing Invariants". In: *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence*. AAAI Press, 2000, pp. 806–811.

[27] Jussi Rintanen. "Regression for Classical and Nondeterministic Planning". In: *Proceedings of the 18th European Conference on Artificial Intelligence*. IOS Press, 2008, pp. 568–572.

[28] Stephan Schiffel and Michael Thielscher. "Automated Theorem Proving for General Game Playing". In: *Proceedings of the 21st International Joint Conference on Artificial Intelligence*. AAAI Press, 2009, pp. 911–916.

[29] Tommi Syrjänen. *Lparse 1.0 User's Manual*. 1999.

[30] Michael Thielscher and Sebastian Voigt. "A Temporal Proof System for General Game Playing". In: *Proceedings of the 24th AAAI Conference on Artificial Intelligence*. AAAI Press, 2010, pp. 1000–1005.

[31] Lintao Zhang et al. "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver". In: *Proceedings of the 2001 International Conference on Computer-Aided Design*. IEEE Computer Society, 2001, pp. 279–285.

A

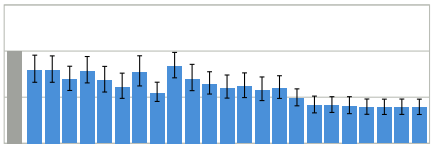
RESULTS: DIRECT KNOWLEDGE FEEDBACK

This appendix contains detailed results of the study in Chapter 4 for each of the seven analyzed problems.

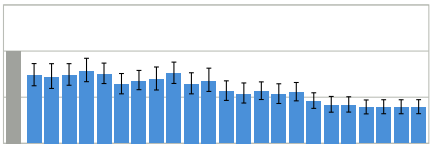
Each chart shows the results of one of the six analyzed selection orders in analogy to Figure 4.3 (a). The *x* axis denotes the number of selected conflict constraints and the *y* axis shows the solving time relative to the baseline (gray bar). The error bars are 95 % confidence intervals.

The additional statistics are aggregated over the respective instances.

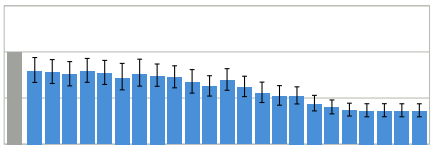
A.1 RICOCHET ROBOTS



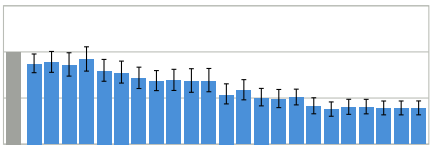
(a) First



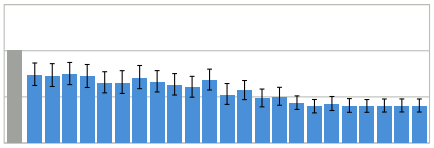
(b) Last



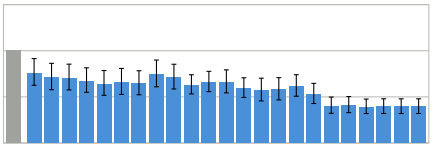
(c) Shortest



(d) Longest



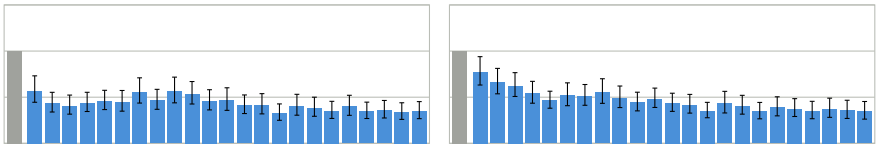
(e) Lowest LBD



(f) Highest LBD

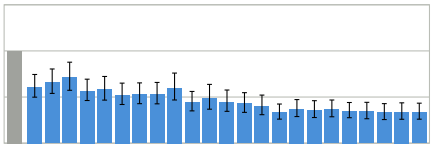
instances	100
baseline solving time	6.38 s
extracted conflict constraints	2570
literals per conflict constraint	97.0
timeouts (extraction)	0
timeouts (feedback)	0

A.2 LABYRINTH

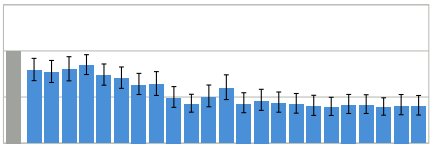


(a) First

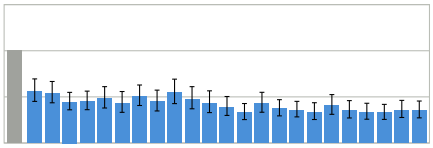
(b) Last



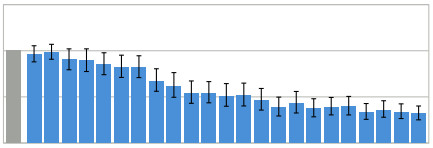
(c) Shortest



(d) Longest



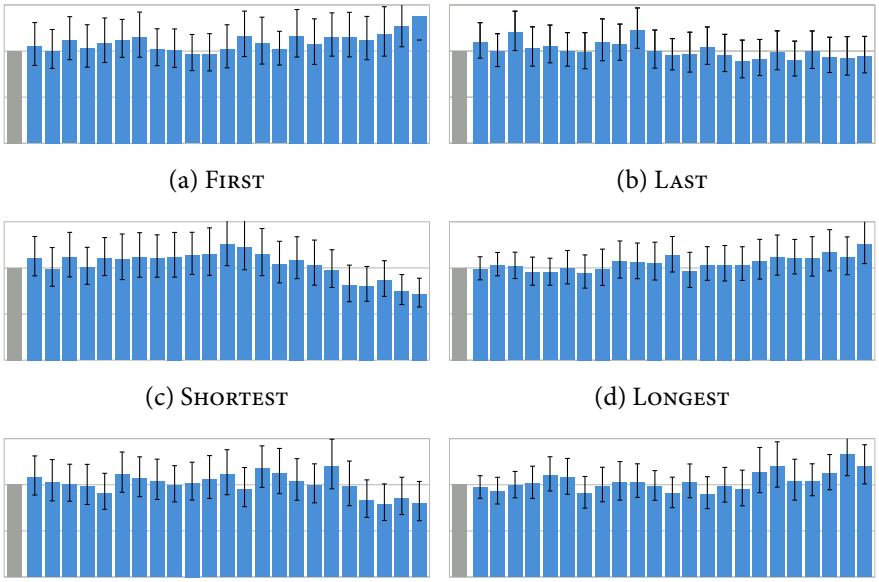
(e) Lowest LBD



(f) Highest LBD

instances	100
baseline solving time	5.07 s
extracted conflict constraints	5008
literals per conflict constraint	48.5
timeouts (extraction)	0
timeouts (feedback)	0

A.3 HANOI TOWER



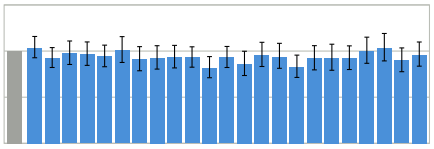
(a) First (b) Last

(c) Shortest (d) Longest

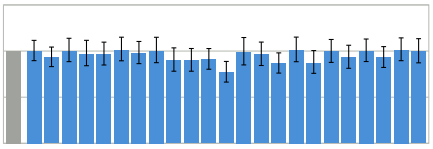
(e) Lowest LBD (f) Highest LBD

instances	100
baseline solving time	5.09 s
extracted conflict constraints	195 850
literals per conflict constraint	16.6
timeouts (extraction)	0
timeouts (feedback)	0

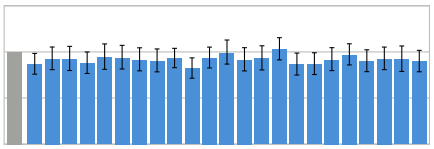
A.4 SOLITAIRE



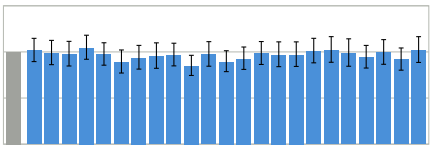
(a) First



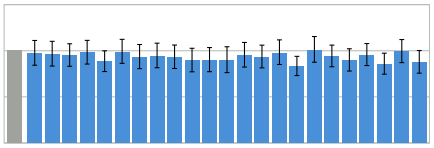
(b) Last



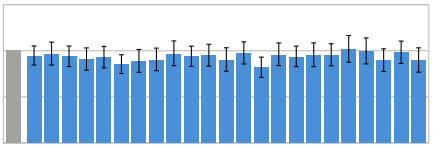
(c) Shortest



(d) Longest



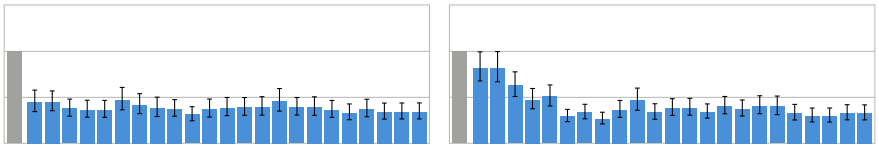
(e) Lowest LBD



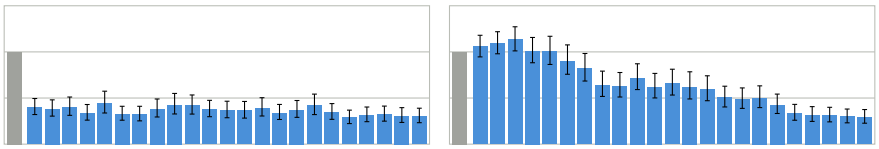
(f) Highest LBD

instances	100
baseline solving time	4.88 s
extracted conflict constraints	30 166
literals per conflict constraint	38.5
timeouts (extraction)	0
timeouts (feedback)	0

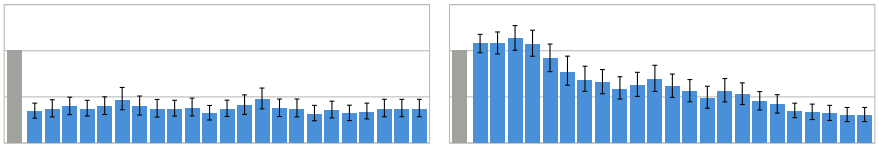
A.5 HAMILTONIAN CYCLE



(a) First (b) Last



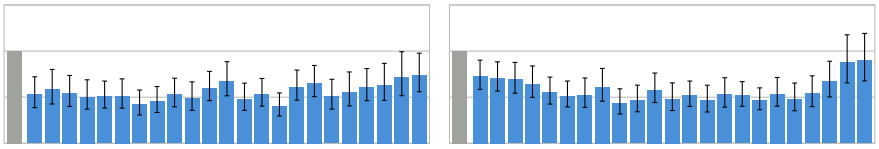
(c) Shortest (d) Longest



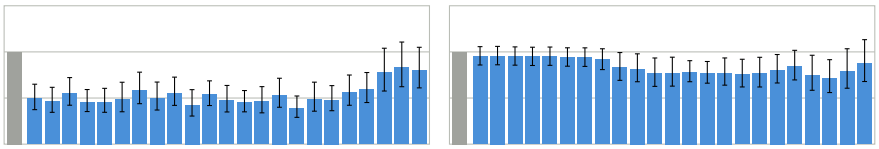
(e) Lowest LBD (f) Highest LBD

instances	100
baseline solving time	4.92 s
extracted conflict constraints	3244
literals per conflict constraint	171.5
timeouts (extraction)	0
timeouts (feedback)	0

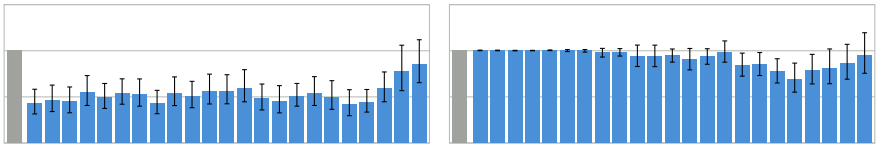
A.6 GRAPH COLORING



(a) First (b) Last



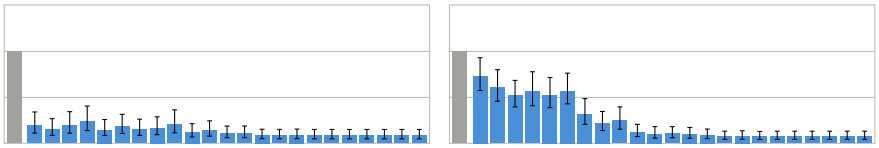
(c) Shortest (d) Longest



(e) Lowest LBD (f) Highest LBD

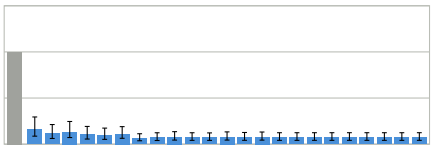
instances	100
baseline solving time	5.04 s
extracted conflict constraints	134 368
literals per conflict constraint	54.0
timeouts (extraction)	0
timeouts (feedback)	0

A.7 KNIGHT CYCLE

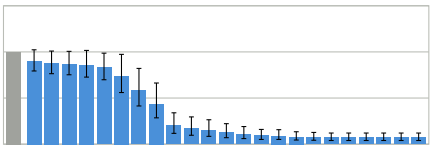


(a) First

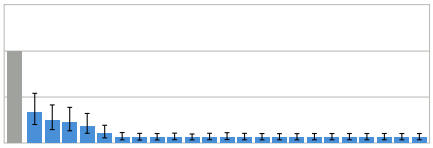
(b) Last



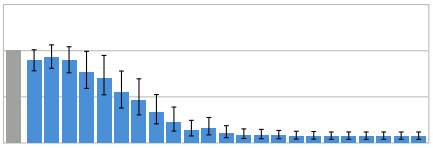
(c) Shortest



(d) Longest



(e) Lowest LBD



(f) Highest LBD

instances	100
baseline solving time	4.72 s
extracted conflict constraints	405
literals per conflict constraint	24.4
timeouts (extraction)	0
timeouts (feedback)	0

B

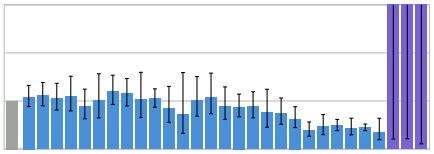
RESULTS: GENERALIZED KNOWLEDGE FEEDBACK

This appendix contains detailed results of the study in Section 6.2 for each of the five analyzed problems.

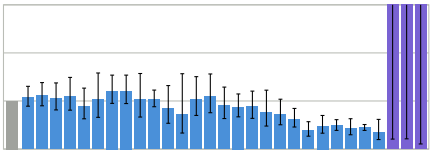
Each chart shows the results of one of the five analyzed configurations in analogy to Figure 6.2. The x axis denotes the number of generalized properties added to the instance. The y axis shows the solving time relative to the baseline (gray bar). Violet bars indicate that at least one contained measurement was penalized by PAR-10 due to a timeout. The error bars are 95 % confidence intervals.

The additional statistics are aggregated over the respective instances.

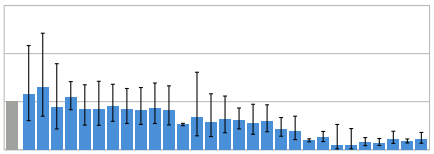
B.1 BLOCKS WORLD



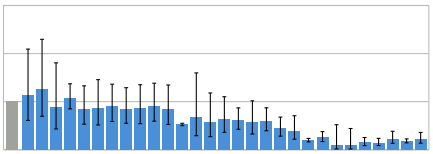
(a) IND., NO MIN., FL. CL.



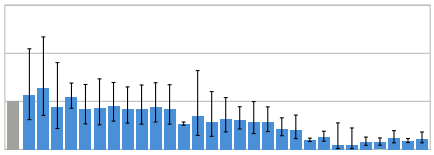
(b) SIMPL., NO MIN., FL. CL.



(c) IND., MIN., FL. CL.



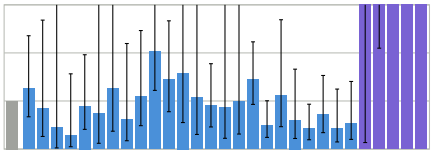
(d) SIMPL., MIN., FL. CL.



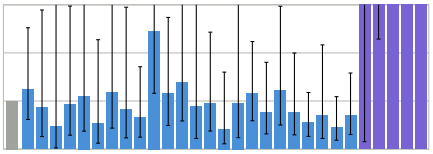
(e) SIMPL., MIN., NO FL. CL.

	(a)	(b)	(c)	(d)	(e)
instances	5	5	5	5	5
total runtime	261 s	188 s	1462 s	745 s	738 s
tested candidates	1400	1400	1978	1978	1978
successful proofs	74.0 %	74.0 %	51.8 %	51.8 %	51.8 %
proof grounding time	74 ms	36 ms	68 ms	33 ms	33 ms
proof solving time	7 ms	3 ms	8 ms	4 ms	4 ms
candidate degree	1.4	1.4	2.5	2.5	2.5
candidate literals	12.1	12.1	14.1	14.1	14.1
minimized literals	—	—	64.6 %	64.6 %	64.6 %
extraction time	12.1 s	11.7 s	11.5 s	11.5 s	11.9 s
extraced constraints	15 349	15 349	15 251	15 251	15 251
timeouts (feedback)	4.0 %	4.0 %	0.0 %	0.0 %	0.0 %

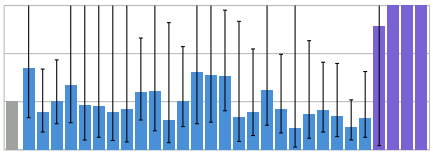
B.2 ELEVATOR



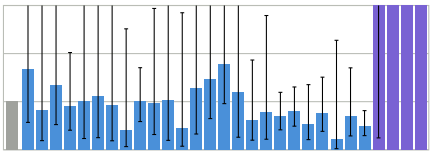
(a) Ind., No Min., Fl. Cl.



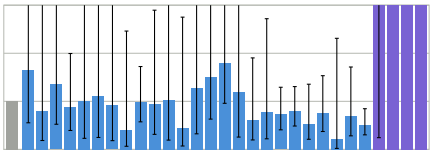
(b) Simpl., No Min., Fl. Cl.



(c) Ind., Min., Fl. Cl.



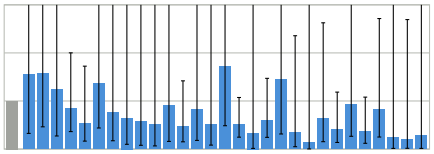
(d) Simpl., Min., Fl. Cl.



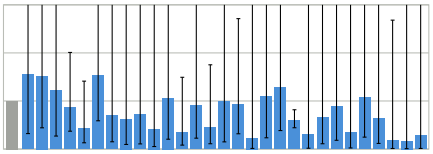
(e) Simpl., Min., No Fl. Cl.

	(a)	(b)	(c)	(d)	(e)
instances	5	5	5	5	5
total runtime	223 s	192 s	648 s	335 s	370 s
tested candidates	1455	1458	2063	2071	2071
successful proofs	70.4 %	70.2 %	49.6 %	49.4 %	49.4 %
proof grounding time	35 ms	17 ms	39 ms	17 ms	21 ms
proof solving time	3 ms	1 ms	3 ms	2 ms	2 ms
candidate degree	1.0	1.0	1.4	1.4	1.4
candidate literals	6.8	6.8	7.4	7.4	7.4
minimized literals	—	—	50.8 %	50.6 %	50.6 %
extraction time	4.1 s	4.0 s	4.0 s	4.1 s	4.1 s
extraced constraints	16 384	16 384	16 384	16 384	16 384
timeouts (feedback)	12.0 %	12.7 %	10.0 %	11.3 %	11.3 %

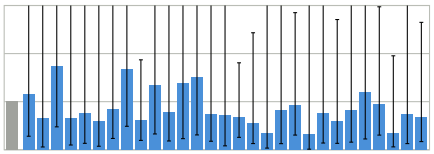
B.3 FREECELL



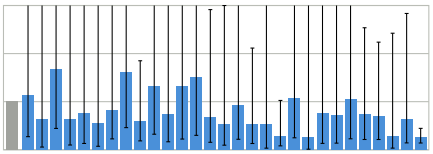
(a) Ind., No Min., Fl. Cl.



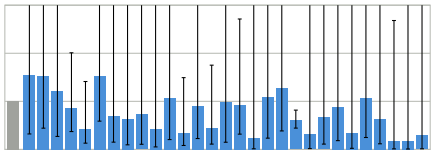
(b) Simpl., No Min., Fl. Cl.



(c) Ind., Min., Fl. Cl.



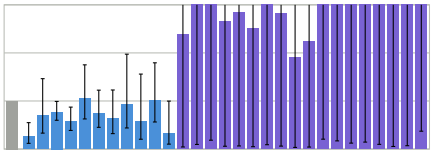
(d) Simpl., Min., Fl. Cl.



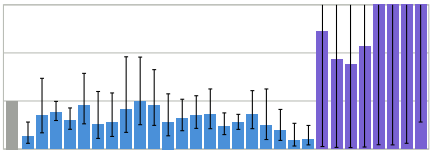
(e) Simpl., Min., No Fl. Cl.

	(a)	(b)	(c)	(d)	(e)
instances	5	5	5	5	5
total runtime	2356 s	1325 s	12 259 s	5988 s	5270 s
tested candidates	1675	1695	1687	1672	1694
successful proofs	62.3 %	61.3 %	60.7 %	61.3 %	60.5 %
proof grounding time	776 ms	354 ms	741 ms	335 ms	311 ms
proof solving time	300 ms	133 ms	298 ms	131 ms	94 ms
candidate degree	1.0	1.0	1.0	1.0	1.0
candidate literals	10.5	10.4	10.5	10.3	9.9
minimized literals	—	—	77.5 %	77.1 %	76.5 %
extraction time	366.8 s	374.4 s	292.9 s	302.8 s	322.5 s
extraced constraints	16 958	16 958	14 381	14 700	15 588
timeouts (feedback)	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %

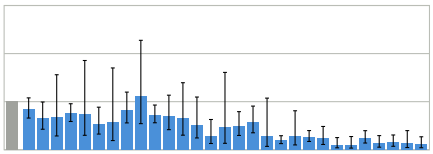
B.4 LOGISTICS



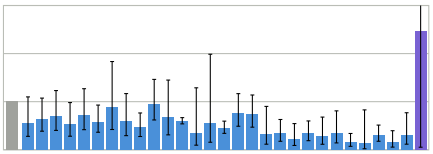
(a) IND., NO MIN., FL. CL.



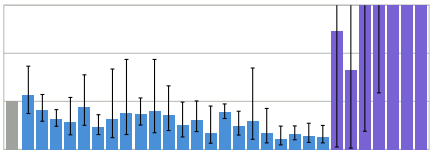
(b) SIMPL., NO MIN., FL. CL.



(c) IND., MIN., FL. CL.



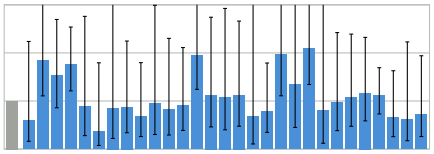
(d) SIMPL., MIN., FL. CL.



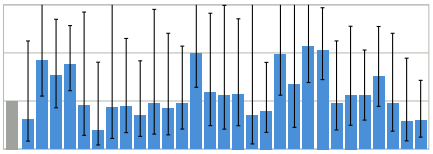
(e) SIMPL., MIN., NO FL. CL.

	(a)	(b)	(c)	(d)	(e)
instances	5	5	5	5	5
total runtime	340 s	282 s	1231 s	631 s	1320 s
tested candidates	1795	1877	1977	2348	7830
successful proofs	49.9 %	47.7 %	52.0 %	43.6 %	7.7 %
proof grounding time	65 ms	32 ms	68 ms	34 ms	45 ms
proof solving time	8 ms	5 ms	8 ms	5 ms	8 ms
candidate degree	2.8	2.8	2.9	3.0	5.7
candidate literals	7.2	7.3	7.6	7.6	13.7
minimized literals	—	—	28.2 %	26.1 %	41.4 %
extraction time	6.8 s	7.0 s	7.1 s	6.6 s	8.0 s
extraced constraints	10 271	10 271	13 673	11 688	13 547
timeouts (feedback)	18.0 %	8.7 %	0.0 %	0.7 %	14.7 %

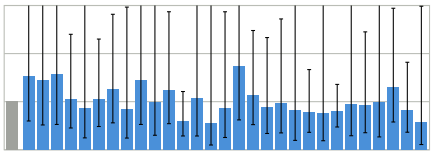
B.5 DEPOTS



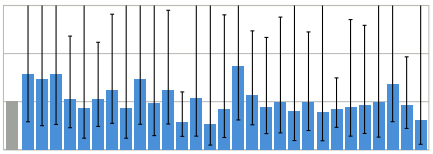
(a) Ind., No Min., Fl. Cl.



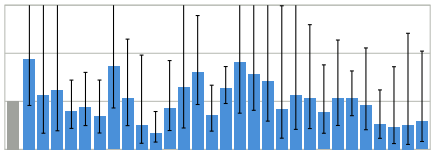
(b) Simpl., No Min., Fl. Cl.



(c) Ind., Min., Fl. Cl.



(d) Simpl., Min., Fl. Cl.



(e) Simpl., Min., No Fl. Cl.

	(a)	(b)	(c)	(d)	(e)
instances	5	5	5	5	5
total runtime	276 s	178 s	1230 s	617 s	605 s
tested candidates	1354	1355	1419	1420	1811
successful proofs	76.3 %	76.3 %	72.2 %	72.1 %	56.5 %
proof grounding time	104 ms	50 ms	95 ms	45 ms	48 ms
proof solving time	20 ms	10 ms	21 ms	10 ms	14 ms
candidate degree	1.2	1.2	1.3	1.3	1.6
candidate literals	6.6	6.6	7.5	7.5	9.3
minimized literals	—	—	55.4 %	55.3 %	59.7 %
extraction time	10.2 s	10.6 s	8.5 s	8.6 s	10.2 s
extraced constraints	14 084	14 084	12 061	12 061	14 075
timeouts (feedback)	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %

DECLARATION

I hereby declare that the present thesis is my original work and it has been written by me in its entirety. I have acknowledged all the sources of information that have been used in this thesis.

Hiermit erkläre ich, dass ich die gesamte vorliegende Arbeit selbstständig verfasst habe. Ich habe sämtliche in dieser Arbeit verwendeten Quellen als solche kenntlich gemacht.

Patrick Lühne
Potsdam, September 2015