Answer Set Solving with Generalized Learned Constraints*

Martin Gebser[1], Roland Kaminski[2], Benjamin Kaufmann[3], Patrick Lühne[4], Javier Romero[5], and Torsten Schaub[6]

1 **University of Potsdam, Potsdam, Germany**
2 **University of Potsdam, Potsdam, Germany**
3 **University of Potsdam, Potsdam, Germany**
4 **University of Potsdam, Potsdam, Germany**
5 **University of Potsdam, Potsdam, Germany**
6 **University of Potsdam, Potsdam, Germany; and INRIA, Rennes, France**

Abstract

Conflict learning plays a key role in modern Boolean constraint solving. Advanced in satisfiability testing, it has meanwhile become a base technology in many neighboring fields, among them *answer set programming* (ASP). However, learned constraints are only valid for a currently solved problem instance and do not carry over to similar instances. We address this issue in ASP and introduce a framework featuring an integrated feedback loop that allows for reusing conflict constraints. The idea is to extract (propositional) conflict constraints, generalize and validate them, and reuse them as integrity constraints. Although we explore our approach in the context of dynamic applications based on transition systems, it is driven by the ultimate objective of overcoming the issue that learned knowledge is bound to specific problem instances. We implemented this workflow in two systems, namely, a variant of the ASP solver *clasp* that extracts integrity constraints along with a downstream system for generalizing and validating them.

1998 ACM Subject Classification D.1.6 Logic Programming, I.2.3 Deduction and Theorem Proving

Keywords and phrases Answer Set Programming, Conflict Learning, Constraint Generalization, Generalized Constraint Feedback

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.9

1 Introduction

Modern solvers for *answer set programming* (ASP) such as *cmodels* [13], *clasp* [11], and *wasp* [1] owe their high effectiveness to advanced Boolean constraint processing techniques centered on *conflict-driven constraint learning* (CDCL; [2]). Unlike pure backtracking, CDCL analyzes encountered conflicts and acquires new constraints while solving, which are added to the problem specification to prune the remaining search space. This strategy often leads to considerably reduced solving times compared to simple backtracking. However, constraints learned in this way are propositional and only valid for the currently solved logic program. Learned constraints can thus only be reused as is for solving the very same problem; they cannot be transferred to solving similar problems, even if they share many properties. For illustration, consider a maze problem, which consists of finding the shortest way out of a

* This work was partially supported by DFG-SCHA-550/9.

Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016).
Editors: Manuel Carro, Andy King, Neda Saeedloei, and Marina De Vos; Article No. 9; pp. 9:1–9:15
Open Access Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

labyrinth. When solving an instance, the solver might learn that shortest solutions never contain a move west followed by a move east, that is, a simple loop. However, the solver only learns this for a specific step but cannot transfer the information to other steps, let alone for solving any other maze instance.

In what follows, we address this shortcoming and introduce a framework for *reusing* learned constraints, with the ultimate objective of overcoming the issue that learned knowledge is bound to specific instances. Reusing learned constraints consists of enriching a program with conflict constraints learned in a previous run. More precisely, our approach proceeds in four steps: (1) extracting constraints while solving, (2) generalizing them, which results in candidates, (3) validating the candidates, and (4) enriching the program with the valid ones. Since this mechanism involves a feedback step, we refer to it as *constraint feedback*. We implemented our framework as two systems, a variant of the ASP solver *clasp* 3 addressing step (1), referred to as *xclasp*, and a downstream system dealing with steps (2) and (3), called *ginkgo*. Notably, we use ASP for implementing different proof methods addressing step (3). The resulting integrity constraints can then be used to enrich the same or "similar" problem instances. To be more precise, we apply our approach in the context of automated planning, as an exemplar of a demanding and widespread application area representative of dynamic applications based on transition systems. As such, our approach readily applies to other related domains, such as action languages or model checking. Furthermore, automated planning is of particular interest because it involves invariants. Although such constraints are specific to a planning problem, they are often independent of the planning instance and thus transferable from one instance of a problem to another. Returning to the above maze example, this means that the constraint avoiding simple loops does not only generalize to all time steps, but is moreover independent of the particular start and exit position.

2 Background

A *logic program* is a set of rules of the form

$$a_0 \leftarrow a_1, \dots, a_m, {\sim}a_{m+1}, \dots, {\sim}a_n \tag{1}$$

where each a_i is a first-order atom for $0 \leq i \leq n$ and "$\sim$" stands for *default negation*. If $n = 0$, rule (1) is called a *fact*. If a_0 is omitted, rule (1) represents an integrity constraint. Further language constructs exist but are irrelevant to what follows (cf. [3]). Rules with variables are viewed as shorthands for the set of their ground instances. Whenever we deal with authentic source code, we switch to typewriter font and use "`:-`" and "`not`" instead of "$\leftarrow$" and "$\sim$"; otherwise, we adhere to the ASP language standard [4]. Semantically, a ground logic program induces a collection of *answer sets*, which are distinguished models of the program determined by answer set semantics; see [12] for details.

Accordingly, the computation of answer sets of logic programs is done in two steps. At first, an ASP grounder instantiates a given logic program. Then, an ASP solver computes the answer sets of the obtained ground logic program. In CDCL-based ASP solvers, the computation of answer sets relies on advanced Boolean constraint processing. To this end, a ground logic program P is transformed into a set Δ_P of *nogoods*, a common (negative) way to represent constraints [8]. A nogood can be understood as a set $\{\ell_1, \dots, \ell_n\}$ of literals representing an invalid partial truth assignment. Logically, this amounts to the formula $\neg(\ell_1 \wedge \dots \wedge \ell_n)$, which in turn can be interpreted as an integrity constraint of the form "$\leftarrow \ell_1, \dots, \ell_n$." By representing a total assignment as a set S of literals, one for each available atom, S is a *solution* for a set Δ of nogoods if $\delta \not\subseteq S$ for all $\delta \in \Delta$. Conversely, S is *conflicting* if $\delta \subseteq S$ for some $\delta \in \Delta$. Such a nogood is called a *conflict nogood* (and the starting point of conflict analysis in CDCL-based solvers). Finally, given a nogood δ and a set S representing a partial assignment, a literal $\ell \notin S$ is *unit-resulting* for δ with respect to S if $\delta \setminus S = \{\overline{\ell}\}$, where $\overline{\ell}$ is the complement of ℓ. Such a nogood δ is called a *reason* for ℓ. That is, if all but one literal of a nogood are contained in an assignment, the complement of the remaining literal must hold in any solution extending the current assignment. *Unit propagation* is the iterated process of extending assignments with unit-resulting literals until no further literal is unit-resulting for any nogood. For instance, consider the partial assignment $\{a \mapsto t, b \mapsto f\}$ represented by $\{a, {\sim}b\}$. Then, ${\sim}c$ is unit-resulting for $\{a, c\}$, leading to the extended assignment $\{a, {\sim}b, {\sim}c\}$. In other words, $\{a, c\}$ is a reason for ${\sim}c$ in $\{a, {\sim}b, {\sim}c\}$. In this way, nogoods provide reasons explaining why literals belong to a solution. Note that any individual assignment is obtained by either a choice operation or unit propagation. Accordingly, assignments are partitioned into *decision levels*. Level zero comprises all initially propagated literals; each higher decision level consists of one choice literal along with successively propagated literals. Further Boolean constraint processing techniques can be used to analyze and recombine inherent reasons for conflicts, as described in Section 3.1. We refer the reader to [11] for a detailed account of the aforementioned concepts.

3 Generalization of Learned Constraints

This section presents our approach by following its four salient steps. At first, we detail how conflict constraints are extracted while solving a logic program and turned into integrity constraints. Then, we describe how the obtained integrity constraints can be generalized by replacing specific terms by variables. Next, we present ASP-based proof methods for validating the generated candidate constraints. For clarity, these methods are developed in the light of our application area of automated planning. Finally, we close the loop and discuss the range of problem instances that can be enriched by the resulting integrity constraints.

While we implemented constraint extraction as an extension to *clasp*, referred to as *xclasp*, our actual constraint feedback framework involving constraint generalization and validation is comprised in the *ginkgo* system. The implementation of both systems is detailed in Section 4.

3.1 Extraction

Modern CDCL solvers gather knowledge in the form of conflict nogoods while solving. Accessing these learned nogoods is essential for our approach. To this end, we have to instrument a solver such as *clasp* to record conflict nogoods resulting from conflict analysis. This necessitates a modification of the solver's conflict resolution scheme, as the learned nogoods can otherwise contain auxiliary literals (standing for unnamed atoms, rule bodies, or aggregates) having no symbolic representation.

The needed modifications are twofold, since literals in conflict nogoods are either obtained by a choice operation or by unit propagation. On the one hand, enforcing named choice literals can be done by existing means, namely, the heuristic capacities of *clasp*. To this end, it is enough to instruct *clasp* to strictly prefer atoms in the symbol table (declared via `#show` statements) for nondeterministic choices.[1]

On the other hand, enforcing learned constraints with named literals only needs changes to *clasp*'s internal conflict resolution scheme. In fact, *clasp*, as many other ASP and SAT solvers,

[1] This is done by launching *clasp* with the options `--heuristic=domain --dom-mod=1,16`.

uses the *first unique implication point* (1UIP) scheme [20]. In this scheme, the original conflict nogood is transformed by successive resolution steps into another conflict nogood containing only a single literal from the decision level at which the conflict occurred. This is either the last choice literal or a literal obtained by subsequent propagation. Each resolution step takes a conflict nogood δ containing a literal ℓ and resolves it with a reason ϵ for ℓ, resulting in the conflict nogood $(\delta \setminus \{\ell\}) \cup (\epsilon \setminus \{\overline{\ell}\})$. We rely upon this mechanism for eliminating unnamed literals from conflict nogoods. To this end, we follow the 1UIP scheme but additionally resolve out all unnamed (propagated) literals. We first derive a conflict nogood with a single *named* literal from the conflicting decision level and then resolve out all unnamed literals from other levels. As with 1UIP, the strategy is to terminate resolution as early as possible. In the best case, all literals are named and we obtain the same conflict nogood as with 1UIP. In the worst case, all propagated literals are unnamed and thus resolved out. This yields a conflict nogood comprised of choice literals, whose naming is enforced as described above.[2] Hence, provided that the set of named atoms is sufficient to generate a complete assignment by propagation, our approach guarantees all conflict nogoods to be composed of named literals. Finally, each resulting conflict nogood $\{\ell_1, \ldots, \ell_n\}$ is output as an integrity constraint "$\leftarrow \ell_1, \ldots, \ell_n$."

Eliminating unnamed literals burdens conflict analysis with additional resolution steps that result in weaker conflict nogoods and heuristic scores. To quantify this, we conducted experiments contrasting solving times with *clasp*'s 1UIP scheme and our named variant, with and without the above heuristic modification (yet without logging conflict constraints). We ran the configurations up to 600 seconds on each of the 100 instances of track 1 of the 2015 ASP competition. Timeouts were accounted for as 600 seconds. *clasp*'s default configuration solved 70 instances in 28 014 seconds, while the two named variants solved 65 in 29 982 and 63 in 29 700 seconds, respectively. Additionally, we ran all configurations on the 42 instances of our experiments in Section 5. While *clasp* solved all instances in 5596 seconds, the two named variants solved 22 in 16 133 and 16 in 17 607 seconds, respectively. Given that these configurations are meant to be used offline, we consider this loss as tolerable.

3.2 Selection

In view of the vast amount of learnable constraints, it is indispensable to select a restricted subset for constraint feedback. To this end, we allow for selecting a given number of constraints satisfying certain properties. We consider the

1. length of constraints (longest vs. shortest),
2. number of decision levels associated with their literals[3] (highest vs. lowest), and
3. time of recording (first vs. last).

To facilitate the selection, *xclasp* initially records all learned conflict constraints (within a time limit), and the *ginkgo* system then picks the ones of interest downstream.

The simplest form of reusing learned constraints consists of enriching an instance with subsumption-free propositional integrity constraints extracted from a previous run on the same instance. We refer to this as *direct constraint feedback*. We empirically studied the impact of this feedback method along with the various selection options in [19] and for brevity only summarize our results here. Our experiments indicate that direct constraint feedback

---

[2] This worst-case scenario corresponds to the well-known *decision scheme*, using conflict clauses containing choice literals only (obtained by resolving out all propagated literals). Experiments with a broad benchmark set [19] showed that our named 1UIP-based scheme uses only 41 % of the time needed with the decision scheme.

[3] This is known as the *literal block distance* (LBD).

generally improves performance and leads to no substantial degradation. This applies to runtime but also to the number of conflicts and decisions. We observed that solving times decrease with the number of added constraints,[4] except for two benchmark classes[5] showing no pronounced effect. This provided us with the pragmatic insight that the addition of constraints up to a magnitude of 10 000 does not hamper solving. The analysis of the above criteria yielded that (1) preferring short constraints had no negative effect over long ones but sometimes led to significant improvements, (2) the number of decision levels had no significant impact, with a slight advantage for constraints with fewer ones, and (3) the moment of extraction ranks equally well, with a slight advantage for earlier extracted constraints. All in all, we observe that even this basic form of constraint feedback can have a significant impact on ASP solving, though its extent is hard to predict. This is not as obvious as it might seem, since the addition of constraints slows down propagation, and initially added constraints might not yet be of value at the beginning of solving.

3.3 Generalization

The last section indicated the prospect of improving solver performance through constraint feedback. Now, we take this idea one step further by generalizing the learned constraints before feeding them back. The goal of this is to extend the applicability of extracted information and make it more useful to the solver ultimately. To this end, we proceed in two steps. First, we produce candidates for generalized conflict constraints from learned constraints. But since the obtained candidates are not necessarily valid, they are subject to validation. Invalid candidates are rejected, valid ones are kept. We consider two ways of generalization, namely, minimization and abstraction. *Minimization* eliminates as many literals as possible from conflict constraints. The smaller a constraint, the more it prunes the search space. *Abstraction* consists of replacing designated constants in conflict constraints by variables. This allows for extending the validity of a conflict constraint from a specific object to all objects of the same domain. This section describes generalization by minimization and abstraction, while the validation of generalized constraints is detailed in Section 3.4.

3.3.1 Minimization

Minimization aims at finding a minimal subset of a conflict constraint that still constitutes a conflict. Given that we extract conflicts in the form of integrity constraints, this amounts to eliminating as many literals as possible. For example, when solving a Ricochet Robots puzzle encoded by a program P, our extended solver *xclasp* might extract the integrity constraint

$$\leftarrow {\sim}go(red, up, 3), go(red, up, 4), {\sim}go(red, left, 5) \tag{2}$$

This established conflict constraint tells us that $P \cup \{h \leftarrow C, \leftarrow {\sim}h\}$ is unsatisfiable for $C = \{{\sim}go(red, up, 3), go(red, up, 4), {\sim}go(red, left, 5)\}$. The minimization task then consists of determining some minimal subset C' of C such that $P \cup \{h \leftarrow C', \leftarrow {\sim}h\}$ remains unsatisfiable, which in turn means that no answer set of P entails all of the literals in C'.

To traverse (proper) subsets C' of C serving as candidates, our *ginkgo* system pursues a greedy approach that aims at eliminating literals one by one. For instance, given C as above, it may start with $C' = C \setminus \{{\sim}go(red, up, 3)\}$ and check whether $P \cup \{h \leftarrow C', \leftarrow {\sim}h\}$ is

[4] We varied the number of extracted constraints from 8 to 16 384 in steps of factor $\sqrt{2}$.
[5] These classes consist of Solitaire and Towers of Hanoi puzzles.

unsatisfiable. If so, "$\leftarrow C'$" is established as a valid integrity constraint; otherwise, the literal $\sim go(red, up, 3)$ cannot be eliminated. Hence, depending on the result, either $C' \setminus \{\ell\}$ or $C \setminus \{\ell\}$ is checked next, where ℓ is one of the remaining literals $go(red, up, 4)$ and $\sim go(red, left, 5)$. Then, (un)satisfiability is checked again for the selected literal ℓ, and ℓ is either eliminated or not before proceeding to the last remaining literal.

Clearly, the minimal subset C' determined by this greedy approach depends on the order in which literals are selected to check and possibly eliminate them. Moreover, checking whether $P \cup \{h \leftarrow C', \leftarrow \sim h\}$ is unsatisfiable can be hard, and in case P itself is unsatisfiable, eventually taking $C' = \emptyset$ amounts to solving the original problem. The proof methods of *ginkgo*, described in Section 3.4, refer to problem relaxations to deal with the latter issue.

3.3.2 Abstraction

Abstraction aims at deriving candidate conflict constraints by replacing constants in ground integrity constraints with variables covering their respective domains. For illustration, consider integrity constraint (2) again. While this constraint is specific to a particular robot (*red*), it might also be valid for all the other available robots:

$$\leftarrow robot(R), \sim go(R, up, 3), go(R, up, 4), \sim go(R, left, 5)$$

Here, the predicate *robot* delineates the domain of robot identifiers. Further candidates can be obtained by extending either direction *up* or *left* to any possible direction. In both cases, we extend the scope of constraints from objects to unstructured domains.

Unlike this, the third parameter of the *go* predicate determines the time step at which the robot moves and belongs to the ordered domain of nonnegative integers. Thus, the conflict constraint might be valid for any sequence of points in time, given by the predicate *time*:

$$\leftarrow time(T), time(T+1), time(T+2), \sim go(red, up, T), go(red, up, T+1), \sim go(red, left, T+2)$$

The time domain is of particular interest when it comes to checking candidates, since it allows for identifying invariants in transition systems (see Section 3.4). This is a reason why the current prototype of *ginkgo* focuses on abstracting temporal constants to variables. In fact, *ginkgo* extracts all time points $t_1, \dots, t_n$ in a constraint in increasing order and replaces them by $T, T + (t_2 - t_1), \dots, T + (t_n - t_1)$, where T is a variable and $t_i < t_{i+1}$ for $0 < i < n$. We refer to integrity constraints obtained by abstraction over a domain of time steps as *temporal constraints*, denote them by "$\leftarrow C[T]$," where T is the introduced temporal variable, and refer to the difference $t_n - t_1$ as the *degree*.

3.4 Validation

Validating an integrity constraint is about showing that it holds in all answer sets of a logic program. To this end, we use counterexample-oriented methods that can be realized in ASP. Although the respective approach at the beginning of Section 3.3.1 is universal, as it applies to any program, it has two drawbacks. First, it is instance-specific, and second, proof attempts face the hardness of the original problem. With hard instances, as encountered in planning, this is impracticable, especially when checking many candidates. Also, proofs neither apply to other instances of the same planning problem nor carry over to different horizons (plan lengths). To avoid these issues, we pursue a problem-specific approach by concentrating on invariants of transition systems (induced by planning problems). Accordingly, we restrict ourselves to temporal abstractions, as described in Section 3.3, and require problem-specific information, such as state and action variables.

In what follows, we develop two ASP-based proof methods for validating candidates in problems based on transition systems. We illustrate the proof methods below for sequential planning and detail their application in Section 4. We consider *planning problems* consisting of a set F of fluents and a set A of actions, along with *instances* containing an initial state I and a goal condition. Letting $A[t]$ and $F[t]$ stand for action and fluent variables at time step t, a set $I[0]$ of facts over $F[0]$ represents the initial state and a logic program $P[t]$ over $A[t]$ and $F[t-1] \cup F[t]$ describes the transitions induced by the actions of a planning problem (cf. [17]). That is, the two validation methods presented below and corresponding ASP encodings given in [9] do not rely on the goal.

3.4.1 Inductive Method

The idea of using ASP for conducting proofs by induction traces back to verifying properties in game descriptions [14]. To show that a temporal constraint "$\leftarrow C[T]$" of degree k is invariant to a planning problem represented by $I[0]$ and $P[t]$, two programs must be unsatisfiable:

$$I[0] \ \cup \ P[1] \cup \cdots \cup P[k] \ \cup \ \{h(0) \leftarrow C[0], \leftarrow {\sim} h(0)\} \quad (3)$$

$$S[0] \ \cup \ P[1] \cup \cdots \cup P[k+1] \ \cup \ \{h(0) \leftarrow C[0], \leftarrow h(0)\} \ \cup \ \{h(1) \leftarrow C[1], \leftarrow {\sim} h(1)\} \quad (4)$$

Program (3) captures the induction base and rejects a candidate if it is satisfied (starting) at time step 0. Note that when a constraint spans k different time points, all trajectories of length k starting from the initial state are examined.

The induction step is captured in program (4) by using a program $S[0]$ for producing all possible predecessor states (marked by "0"). To this end, $S[0]$ contains a choice rule "$\{f(0)\} \leftarrow$" for each fluent $f(0)$ in $F[0]$. Moreover, program (4) rejects a candidate if the predecessor state (starting at time step 0) violates the candidate or if the successor state (starting at 1) satisfies it. To apply the candidate to the successor step, it is shifted by 1 via $h(1)$. That is, the induction step requires one more time step than the base. If both programs (3) and (4) are unsatisfiable, the candidate is validated. Although the obtained integrity constraint depends on the initial state, it is independent of the goal and applies to varying horizons. Hence, the generalized constraint cannot only be used for enriching the planning instance at hand but also carries over to instances with different horizons and goals.

3.4.2 State-Wise Method

We also consider a simpler validation method that relies on exhaustive state generation. This approach replaces the two-fold induction method with a single search for counterexamples:

$$S[0] \ \cup \ P[1] \cup \cdots \cup P[k] \ \cup \ \{h(0) \leftarrow C[0], \leftarrow {\sim} h(0)\} \quad (5)$$

As in the induction step above, a state is nondeterministically generated via $S[0]$. But instead of performing the step, program (5) rejects a candidate if it is satisfied in the generated state. As before, the candidate is validated if program (5) is unsatisfiable. While this simple proof method is weaker than the inductive one, it is independent of the initial state, and validated generalized constraints thus carry over to all instances of a planning problem. We empirically contrast both approaches in Section 5.

3.5 Feedback

Combining all the previously described steps allows us to enrich logic programs with validated generalized integrity constraints. We call this process *generalized constraint feedback*.

The scope of our approach is delineated by the chosen proof methods. First, they deal with problems based on transition systems. Second, both methods are incomplete, since they might find infeasible counterexamples stemming from unreachable states. However, both methods rely on relatively inexpensive proofs, since candidates are bound by their degree rather than the full horizon. This also makes valid candidates independent of goal conditions and particular horizons; state-wise proven constraints are even independent of initial states.

4 Implementation

We implemented our knowledge generalization framework as two systems: *xclasp* is a variant of the ASP solver *clasp* 3 capable of extracting learned constraints while solving, and the extracted constraints are then automatically generalized and validated offline by *ginkgo*. In this way, *ginkgo* produces generalized constraints that can be reused through generalized constraint feedback. Both *xclasp* and *ginkgo* are available at the Potassco Labs website.[6]

4.1 *xclasp*

xclasp implements the instrumentation described in Section 3.1 as a standalone variant of *clasp* 3.1.4 extended by constraint extraction. The option `--log-learnts` outputs learned integrity constraints so that the output can be readily used by any downstream application. The option `--logged-learnt-limit=n` stops solving once `n` constraints were logged. Finally, the named-literals resolution scheme is invoked with `--resolution-scheme=named`.

4.2 *ginkgo*

ginkgo incorporates the different techniques developed in Section 3. After recording learned constraints, *ginkgo* offers postprocessing steps, one of which is *sorting* logged constraints by multiple criteria. This option is interesting for analyzing the effects of reusing different types of constraints. Another postprocessing step used throughout this paper is (propositional) *subsumption*, that is, removing subsumed constraints. In fact, *xclasp* often learns constraints that are subsets of previous ones (and thus more general and stronger). For example, when solving the 3-Queens puzzle, recorded integrity constraints might be subsumed by "$\leftarrow queen(2,2)$," as a single queen in the middle attacks entire columns and rows.

Figure 1 illustrates *ginkgo*'s generalization procedure. In our setting, the input to *ginkgo* consists of a planning problem, an instance, and a fixed horizon. First, *ginkgo* begins to extract a specified number of learned constraints by solving the instance with our modified solver *xclasp*. Then, *ginkgo* abstracts the learned constraints over the time domain, which results in a set of candidates (see Section 3.3.2). These candidates are validated and optionally minimized (see Section 3.3.1) one by one. For this purpose, *ginkgo* uses either of the two presented validation methods (see Section 3.4), where the candidates are validated in ascending order of degree. This is sensible because the higher the degree, the larger is the search space for counterexamples. Among candidates with the same degree, the ones with fewer literals are tested first, given that the optional minimization of constraints (using the same proof method as for validation) requires less steps for them. Moreover, proven candidates are immediately added to the input logic program in order to strengthen future proofs (while unproven ones are discarded). Finally, *ginkgo* terminates after successfully generalizing a user-specified

[6] `http://potassco.sourceforge.net/labs.html`

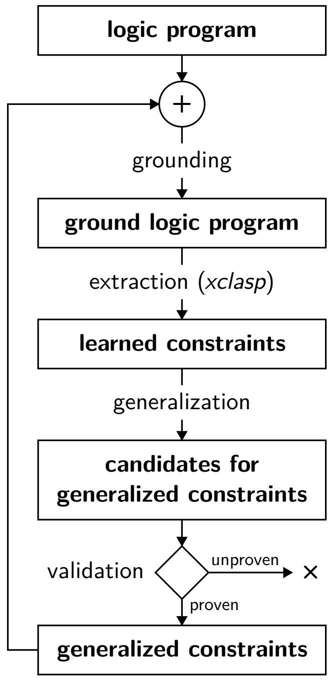


Figure 1 *ginkgo*'s procedure for automatically generalizing learned constraints.

number of constraints. The generalized constraints can then be used to enrich the same or a related logic program via generalized constraint feedback.

ginkgo offers multiple options to steer the constraint generalization procedure. `--horizon` specifies the planning horizon. The validation method is selected via `--proof-method`. `--minimization-strategy` defines whether constraint minimization is used. `--constraints-to-extract` decides how many constraints *ginkgo* extracts before starting to validate them, where the extraction step can also be limited with `--extraction-timeout`. By default, *ginkgo* tests all initially extracted constraints before extracting new ones.[7] Alternatively, new constraints may be extracted after each successful proof (controlled via `--testing-policy`). Candidates exceeding a specific degree (`--max-degree`) or number of literals (`--max-number-of-literals`) may be skipped. Additionally, candidates may be skipped if the proof takes too long (`--hypothesis-testing-timeout`). *ginkgo* terminates after generalizing a number of constraints specified by `--constraints-to-prove` (or if *xclasp*'s constraints are exhausted).

5 Evaluation

To evaluate our approach, we instruct *ginkgo* to learn and generalize constraints autonomously on a set of benchmark instances. These instances stem from the International Planning Competition (IPC) series and were translated to ASP with *plasp* [10].

First, we study how the solver's runtime is affected by generalized constraint feedback – that is, enriching instances with generalized constraints that were obtained beforehand with *ginkgo*. In a second experiment, generalized constraint feedback is performed after *varying*

[7] Note that extracting more constraints is only necessary if the initial chunk of learned constraints does not lead to the requested number of generalized constraints. In practice, this rarely happens when choosing a sufficient number of constraints to extract initially.

Table 1 Configurations of *ginkgo* for studying generalized constraint feedback.

	validation method	minimization	constraint feedback
(a)	state-wise	on	generalized
(b)	inductive	on	generalized
(c)	state-wise	off	generalized
(d)	state-wise	on	direct

the instances' horizons. Among other things, this allows us to study scenarios in which constraints are first generalized using *simplified* settings to speed up the solving process of the *actual* instances later on. The benchmark sets are available at *ginkgo*'s website.[6]

5.1 Generalized Constraint Feedback

In this experiment, we use *ginkgo* to generalize a specific number of learned constraints for each instance. Then, we enrich the instances via generalized constraint feedback and measure how the enriched instances relate to the original ones in terms of runtime. This setup allows us to assess whether reusing generalized constraints improves solving the individual instances.

The benchmark set consists of 42 instances from the 2000, 2002, and 2006 IPCs and covers nine planning domains: Blocks World (8), Driver Log (4), Elevator (11), FreeCell (4), Logistics (5), Rovers (1), Satellite (3), Storage (4), and Zeno Travel (2). We selected instances with solving times within 10 to 600 seconds on the benchmark system (using *clasp* with default settings). For 33 instances, we used minimal horizons. We chose higher horizons for the remaining nine instances because timeouts occurred with minimal horizons.

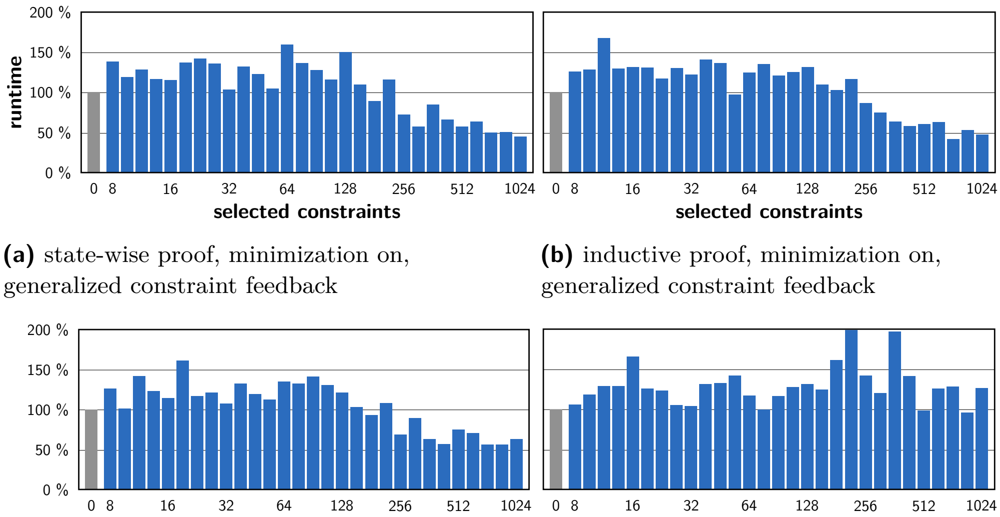
Given an instance and a fixed horizon, 1024 generalized constraints are first generated offline with *ginkgo*. Afterward, the solving time of the instance is measured multiple times. Each time, the instance is enriched with the first n generalized constraints, where n varies between 8 and 1024 in exponential steps. The original instance is solved once more without any feedback for reference. Afterward, the runtimes of the enriched instances are compared to the original ones. All runtimes are measured with *clasp*'s default configuration, not *xclasp*.

We perform this experiment with the four *ginkgo* configurations shown in Table 1.

First, we select the state-wise proof method and enable minimization (a). We chose this configuration as a reference because the state-wise proof method achieves instance independence (see Section 3.4.2) and because minimization showed to be useful in earlier experiments [19]. To compare the two validation methods presented in this paper, we repeat the experiment with the inductive proof method (b). In configuration (c), we disable constraint minimization to assess the benefit of this technique. Finally, configuration (d) replaces generalized with direct constraint feedback (that is, the instances are not enriched with the generalized constraints but the ground learned constraints they stem from). With configuration (d), we can evaluate whether generalization renders learned constraints more useful.

We fix *ginkgo*'s other options across all configurations. Generalization starts after *xclasp* extracted 16 384 constraints or after 600 seconds. Candidates with degrees greater than 10 or more than 50 literals are skipped, and proofs taking more than 10 seconds are aborted. After *ginkgo* terminates, the runtimes of the original and enriched instances are measured with a limit of 3600 seconds. Timeouts are penalized with PAR-10 (36 000 seconds). The benchmarks were run on a Linux machine with Intel Core i7-4790K at 4.4 GHz and 16 GB RAM.

As Figure 2a shows, generalized constraint feedback reduced the solver's runtime by up to 55 %. The runtime decreases the more generalized constraints are selected for feedback.



(a) state-wise proof, minimization on, generalized constraint feedback

(b) inductive proof, minimization on, generalized constraint feedback

(c) state-wise proof, minimization off, generalized constraint feedback

(d) state-wise proof, minimization on, direct constraint feedback

Figure 2 Runtimes after generalized constraint feedback with four different *ginkgo* configurations.

On average, validating a candidate constraint took 73 ms for grounding and 22 ms for solving in reference configuration (a). 38 % of all proofs were successful, and *ginkgo* terminated after 1169 seconds on average. The tested candidates had an average degree of 2.2 and contained 9.3 literals. Constraint minimization eliminated 63 % of all literals in generalized constraints.

While lacking the instance independence of the state-wise proof, the supposedly stronger inductive proof did not lead to visibly different results (see Figure 2b). Additionally, validating candidate constraints took about 2.3 times longer. With 2627 seconds, the average total runtime of *ginkgo* was 2.2 times higher with the inductive proof method. Disabling constraint minimization had rather little effect on the generalized constraints' utility in terms of solver runtime, as seen in Figure 2c. However, without constraint minimization, *ginkgo*'s runtime was reduced to 332 seconds (a factor of 3.5 compared to the reference configuration). Interestingly, direct constraint feedback was never considerably useful for the solver (see Figure 2d). Hence, we conclude that learned constraints are indeed strengthened by generalization.

5.2 Generalized Constraint Feedback with Varying Horizons

This experiment evaluates the *generality* of the proven constraints – that is, whether priorly generalized constraints improve the solving performance on similar instances. For this purpose, we use *ginkgo* to extract and generalize constraints on the benchmark instances with fixed horizons. Then, we vary the horizons of the instances and solve them again, after enriching them with the previously generalized constraints.

We reuse the 33 instances with minimal (optimal) horizons from Section 5.1, referring to them as the H_0 set. In addition, we analyze two new benchmark sets. H_{-1} consists of the H_0 instances with horizons reduced by 1, which renders all instances in H_{-1} unsatisfiable. In another benchmark set, H_{+1}, we increase the fixed horizon of the H_0 instances by 1.[8]

[8] The alleged small change of the horizon by 1 is motivated by maintaining the hardness of the problem.

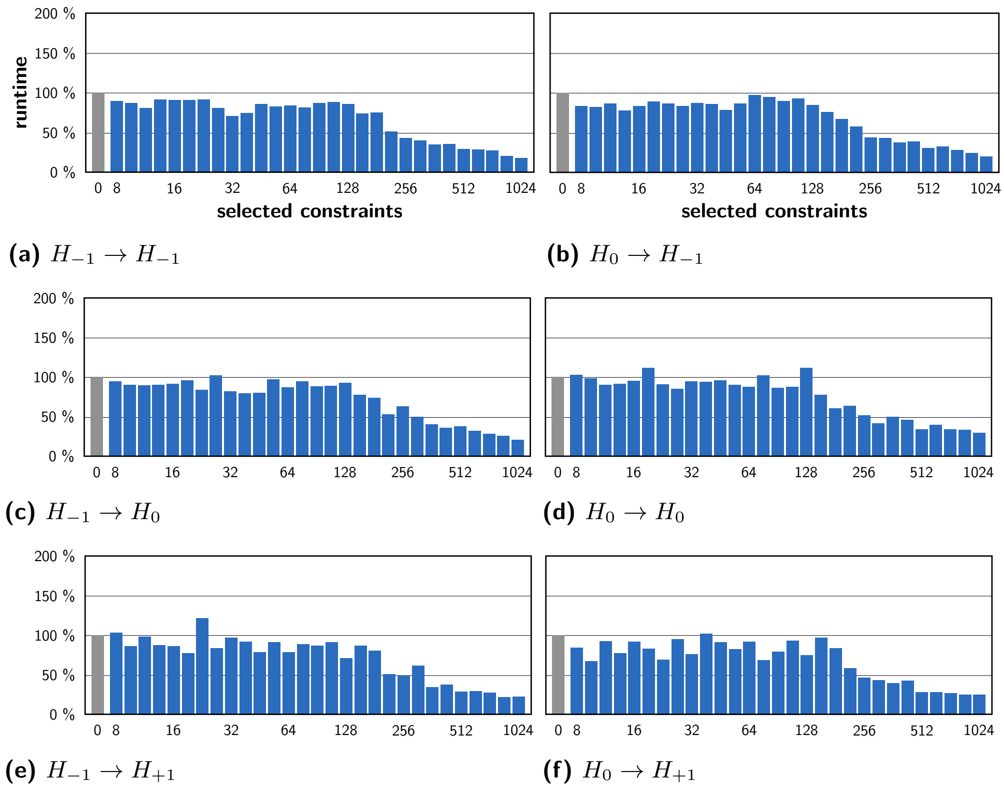


Figure 3 Runtimes after generalized constraint feedback with varied horizons. In setting $H_x \to H_y$, constraints were extracted and generalized with benchmark set H_x and reused for solving H_y.

The benchmark procedure is similar to Section 5.1. This time, constraints are extracted and generalized on a specific benchmark set but then applied to the corresponding instances of another set. For instance, $H_{-1} \to H_0$ refers to the setting where constraints are generalized with H_{-1} and then reused while solving the respective instances in H_0. In total, we study six settings: $\{H_{-1}, H_0\} \to \{H_{-1}, H_0, H_{+1}\}$. The choice of $\{H_{-1}, H_0\}$ as sources reflects the extraction from unsatisfiable and satisfiable instances, respectively. To keep the results comparable across all configurations, we removed five instances whose reference runtime (without feedback) exceeded the time limit of 3600 seconds in at least one of H_{-1}, H_0, and H_{+1}. For this reason, the results shown in Figure 3 refer to the remaining 28 instances. In this experiment, the state-wise validation method and minimization are applied. The benchmark environment is identical to Section 5.1.

As Figure 3 shows, generalized constraint feedback permits varying the horizon with no visible penalty.

Across all six settings, the runtime improvements are very similar (up to 70 or 82 %, respectively). Runtime improvements are somewhat more pronounced when constraints are generalized with H_{-1} rather than H_0. Furthermore, generalized constraint feedback on H_{-1} is slightly more useful than on H_0 and H_{+1}. Apart from this, generalized constraint feedback seems to work well no matter whether the programs at hand are satisfiable or not.

6 Discussion

We have presented the systems *xclasp* and *ginkgo*, jointly implementing a fully automated form of generalized constraint feedback for CDCL-based ASP solvers. This is accomplished in a four-phase process consisting of extraction (and selection), generalization (via abstraction and minimization), validation, and feedback. While *xclasp*'s extraction of integrity constraints is domain-independent, the scope of *ginkgo* is delineated by the chosen proof method. Our focus on inductive and state-wise methods allowed us to study the framework in the context of transition-based systems, including the chosen application area of automated planning. We have demonstrated that our approach allows for reducing the runtime of planning problems by up to 55 %. Moreover, the learned constraints cannot only be used to accelerate a program at hand, but they moreover transfer to other goal situations, altered horizons, and even other initial situations (with the state-wise technique). In the latter case, the learned constraints are general enough to apply to all instances of a fixed planning problem. Interestingly, while both proof methods often failed to prove valid, handcrafted properties, they succeeded on relatively many automatically extracted candidates (about 38 %). Generally speaking, it is worthwhile to note that our approach had been impossible without ASP's first-order modeling language along with its distinction of problem encoding and instance.

Although *xclasp* and *ginkgo* build upon many established techniques, we are unaware of any other approach combining the same spectrum of techniques similarly. In ASP, the most closely related work was done in [26] in the context of the first-order ASP solver *omiga* [6]. Rules are represented as Rete networks, propagation is done by firing rules, and unfolding is used to derive new reusable rules. ASP-based induction was first used for verifying predefined properties in game descriptions [14]. Inductive logic programming in ASP [22, 16] is related in spirit but works from different principles, such as deriving rules compatible with positive and negative examples. In SAT, k-induction [24, 25] is a wide-spread technique in applications to model checking. Our state-wise proof method is similar to 0-induction. In FO(ID), [7] deals with detecting functional dependencies for deriving new constraints, where a constraint's validity is determined by a first-order theorem prover. In CP, automated modeling constitutes an active research area (cf. [21]). For instance, [5] addresses constraint reformulation by resorting to machine learning and theorem proving for extraction and validation. Finally, invariants in transition systems have been explored in several fields, among them general game playing [14], planning [23, 15], model checking [24, 25], and reasoning about actions [18]. While inductive and first-order proof methods are predominant, invariants are either assumed to be given or determined by dedicated algorithms.

Our approach aims at overcoming the restriction of learned knowledge to specific problem instances. However, it may also help to close the gap between highly declarative and highly optimized encodings by enriching the former through generalized constraint feedback.

References

1 M. Alviano, C. Dodaro, N. Leone, and F. Ricca. Advances in WASP. In F. Calimeri, G. Ianni, and M. Truszczyński, editors, *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, pages 40–54. Springer, 2015.

2 A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

3 G. Brewka, T. Eiter, and M. Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

4 F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, F. Ricca, and T. Schaub. ASP-Core-2: Input language format. Available at `https://www.mat.unical.it/aspcomp2013/ASPStandardization/`, 2012.

5 J. Charnley, S. Colton, and I. Miguel. Automated generation of implied constraints. In G. Brewka, S. Coradeschi, A. Perini, and P. Traverso, editors, *Proceedings of the Seventeenth European Conference on Artificial Intelligence (ECAI'06)*, pages 73–77. IOS Press, 2006.

6 M. Dao-Tran, T. Eiter, M. Fink, G. Weidinger, and A. Weinzierl. OMiGA : An open minded grounding on-the-fly answer set solver. In L. Fariñas del Cerro, A. Herzig, and J. Mengin, editors, *Proceedings of the Thirteenth European Conference on Logics in Artificial Intelligence (JELIA'12)*, pages 480–483. Springer, 2012.

7 B. De Cat and M. Bruynooghe. Detection and exploitation of functional dependencies for model generation. *Theory and Practice of Logic Programming*, 13(4-5):471–485, 2013.

8 R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

9 M. Gebser, R. Kaminski, B. Kaufmann, P. Lühne, J. Romero, and T. Schaub. Answer set solving with generalized learned constraints (extended version). Available at `http://www.cs.uni-potsdam.de/wv/publications/`, 2016.

10 M. Gebser, R. Kaminski, M. Knecht, and T. Schaub. plasp: A prototype for PDDL-based planning in ASP. In J. Delgrande and W. Faber, editors, *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, pages 358–363. Springer, 2011.

11 M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188:52–89, 2012.

12 M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

13 E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36(4):345–377, 2006.

14 S. Haufe, S. Schiffel, and M. Thielscher. Automated verification of state sequence invariants in general game playing. *Artificial Intelligence*, 187-188:1–30, 2012.

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

16 M. Law, A. Russo, and K. Broda. Inductive learning of answer set programs. In E. Fermé and J. Leite, editors, *Proceedings of the Fourteenth European Conference on Logics in Artificial Intelligence (JELIA'14)*, pages 311–325. Springer, 2014.

17 V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1-2):39–54, 2002.

18 F. Lin. Discovering state invariants. In D. Dubois, C. Welty, and M. Williams, editors, *Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning (KR'04)*, pages 536–544. AAAI Press, 2004.

19 P. Lühne. Generalizing learned knowledge in answer set solving. Master's thesis, Hasso Plattner Institute, Potsdam, 2015.

20 J. Marques-Silva and K. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

21 B. O'Sullivan. Automated modelling and solving in constraint programming. In M. Fox and D. Poole, editors, *Proceedings of the Twenty-fourth National Conference on Artificial Intelligence (AAAI'10)*, pages 1493–1497. AAAI Press, 2010.

22 R. Otero. Induction of stable models. In C. Rouveirol and M. Sebag, editors, *Proceedings of the Eleventh International Conference on Inductive Logic Programming (ILP'01)*, pages 193–205. Springer, 2001.

23 J. Rintanen. An iterative algorithm for synthesizing invariants. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI'00)*, pages 806–811. AAAI/MIT Press, 2000.

24 M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In W. Hunt and S. Johnson, editors, *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design (FMCAD'00)*, pages 108–125. Springer, 2000.

25 Y. Vizel, G. Weissenbacher, and S. Malik. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE*, 103(11):2021–2035, 2015.

26 A. Weinzierl. Learning non-ground rules for answer-set solving. In D. Pearce, S. Tasharrofi, E. Ternovska, and C. Vidal, editors, *Proceedings of the Second Workshop on Grounding and Transformation for Theories with Variables (GTTV'13)*, pages 25–37, 2013.