Solution Enumeration for Projected Boolean Search Problems

Martin Gebser, Benjamin Kaufmann, and Torsten Schaub*

Institut für Informatik, Universität Potsdam, August-Bebel-Str. 89, D-14482 Potsdam, Germany

Abstract. Many real-world problems require the enumeration of all solutions of combinatorial search problems, even though this is often infeasible in practice. However, not always all parts of a solution are needed. We are thus interested in projecting solutions to a restricted vocabulary. Yet, the adaption of Boolean constraint solving algorithms turns out to be non-obvious provided one wants a repetition-free enumeration in polynomial space. We address this problem and propose a new algorithm computing projective solutions. Although we have implemented our approach in the context of Answer Set Programming, it is readily applicable to any solver based on modern Boolean constraint technology.

1 Introduction

Modern Boolean constraint technology has led to a tremendous boost in solver performance in various areas dealing with combinatorial search problems. Pioneered in the area of Satisfiability checking (SAT; [1–3]) where it has demonstrated its maturity for real-world applications, its usage is meanwhile also advancing in neighboring areas, like Answer Set Programming (ASP; [4]) and even classical Constraint Processing. Although traditionally problems are expressed in terms of satisfiability or unsatisfiability, many real-world applications require surveying all solutions of a problem. For instance, inference in Bayes Nets can be reduced to #SAT (cf. [5]) by counting the number of models. However, the exhaustive enumeration of all solutions is often infeasible. Yet not always all parts of a solution are needed. Restrictions may lead to a significant decrease of computational efforts; in particular, whenever the discarded variables have their proper combinatorics and thus induce a multitude of redundant solutions.

We are thus interested in projecting solutions to a restricted vocabulary. However, the adaption of Boolean constraint solving algorithms turns out to be non-obvious, if one wants a repetition-free enumeration in polynomial space. We address this by proposing a new algorithm for solution projection. Given a problem Δ having solutions $\mathcal{S}(\Delta)$ and a set P of variables to project on, we are interested in computing the set $\{S \cap P \mid S \in \mathcal{S}(\Delta)\}$. We refer to its elements as the *projective solutions* for Δ wrt P. To compute all such projections, we first provide a direct extension of a conflict-driven learning algorithm by means of solution recording. Although this approach is satisfactory when the number of projective solutions is limited, it does not scale since it is exponential in space. After analyzing the particularities of the search problem, we propose a new conflict-driven learning algorithm that uses an elaborated backtracking

* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

scheme and only a linear number of solution-excluding constraints. Although we have implemented our approach in the context of ASP, it is readily applicable to any solving approach based on modern Boolean constraint technology. Lastly, we provide an empirical analysis demonstrating the computational impact of our approach.

2 Background

The idea of ASP is to encode a problem as a logic program such that its answer sets represent solutions to the original problem. More formally, a *logic program* Π is a finite set of *rules* of the form $a \leftarrow b_1, \ldots, b_m, \sim c_{m+1}, \ldots, \sim c_n$, where a, b_i, c_j are atoms for $0 < i \leq m, m < j \leq n$ and $\sim$ is (default) negation. The *answer sets* of Π are particular models of Π satisfying an additional stability criterion. For brevity, we refer the reader to [6] for a formal introduction to ASP.

As a running example, consider the program composed of the following rules:

$$\begin{array}{llllll} x \leftarrow q, r & (1) & y \leftarrow x, \sim q & (5) & z \leftarrow x, \sim r & (9) \\ x \leftarrow \sim y, \sim z & (2) & y \leftarrow \sim x, \sim z & (6) & z \leftarrow \sim x, \sim y & (10) \\ p \leftarrow x & (3) & q \leftarrow x & (7) & r \leftarrow x & (11) \\ p \leftarrow \sim x & (4) & q \leftarrow \sim r & (8) & r \leftarrow \sim q & (12)\ . \end{array}$$

Among the ten (classical) models of this program, we find five answer sets: $\{p, q, y\}$, $\{p, q, z\}$, $\{p, q, r, x\}$, $\{p, r, y\}$, and $\{p, r, z\}$. Projecting them onto the atoms $\{p, q, r\}$ results in only three distinct solutions: $\{p, q\}$, $\{p, q, r\}$, and $\{p, r\}$.

An *assignment* $\mathbf{A}$ is a sequence $(\sigma_1, \ldots, \sigma_n)$ of *literals* σ_i of the form $\mathbf{T}v_i$ or $\mathbf{F}v_i$ where v_i is a (Boolean) variable for $1 \leq i \leq n$; $\mathbf{T}v_i$ expresses that v_i is *true* and $\mathbf{F}v_i$ that it is *false*. We denote the complement of a literal σ by $\overline{\sigma}$, that is, $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$. Also, we let $var(\mathbf{T}v) = var(\mathbf{F}v) = v$. We sometimes abuse notation and identify an assignment with the set of its contained literals. Given this, we access the true and false variables in $\mathbf{A}$ via $\mathbf{A}^{\mathbf{T}} = \{v \mid \mathbf{T}v \in \mathbf{A}\}$ and $\mathbf{A}^{\mathbf{F}} = \{v \mid \mathbf{F}v \in \mathbf{A}\}$. For a canonical representation of (Boolean) constraints, we make use of nogoods [7]. In our setting, a *nogood* is a finite set $\{\sigma_1, \ldots, \sigma_m\}$ of literals, expressing a constraint violated by any assignment $\mathbf{A}$ containing $\sigma_1, \ldots, \sigma_m$. For a set Δ of nogoods, define $var(\Delta) = \bigcup_{\delta \in \Delta} \{var(\sigma) \mid \sigma \in \delta\}$. An assignment $\mathbf{A}$ such that $\mathbf{A}^{\mathbf{T}} \cap \mathbf{A}^{\mathbf{F}} = \emptyset$ and $\{\delta \in \Delta \mid \exists \sigma \in \delta : \overline{\sigma} \in \mathbf{A}\} = \Delta$ is a *solution* for Δ. For a given set P of variables, we call a set $\mathbf{P}$ of literals such that $\mathbf{P}^{\mathbf{T}} \cup \mathbf{P}^{\mathbf{F}} = P$ a *projective solution* for Δ wrt P, if there is some solution $\mathbf{A}$ for Δ such that $\mathbf{P} \subseteq \mathbf{A}$.

A translation of logic programs in ASP into nogoods is developed in [4]. For brevity, we illustrate it by two examples. First, consider the nogoods induced by atom y in the above program. Atom y depends on two bodies: $\{x, \sim q\}$ and $\{\sim x, \sim z\}$ in (5) and (6). We get the nogoods $\{\mathbf{T}y, \mathbf{F}\{x, \sim q\}, \mathbf{F}\{\sim x, \sim z\}\}$, $\{\mathbf{F}y, \mathbf{T}\{x, \sim q\}\}$, and $\{\mathbf{F}y, \mathbf{T}\{\sim x, \sim z\}\}$ by taking for convenience the actual bodies rather than introducing new variables. For instance, the first nogood eliminates solutions where y is true although neither the rule in (5) nor (6) are applicable. In turn, body $\{x, \sim q\}$ induces nogoods $\{\mathbf{F}\{x, \sim q\}, \mathbf{T}x, \mathbf{F}q\}$, $\{\mathbf{T}\{x, \sim q\}, \mathbf{F}x\}$, and $\{\mathbf{T}\{x, \sim q\}, \mathbf{T}q\}$. The last two nogoods deny solutions where the body is true although one of its conjuncts is false.

3 Algorithms for Solution Projection

When enumerating solutions, standard backtracking algorithms like that of Davis, Putnam, Logemann, and Loveland (DPLL; [8, 9]) usually encounter multiple solutions being identical on a projected vocabulary. Such redundancy could easily be avoided by branching on projected before any other variables. However, the limitation of branching can cause an exponential degradation of performance (see below).

Also our enumeration algorithms for projective solutions make use of a decision heuristic: SELECT($\Delta, \nabla, \mathbf{A}, P$) takes a set Δ of (input) nogoods, a set ∇ of (recorded) nogoods, an assignment $\mathbf{A}$, and a set P of variables as arguments. Dynamic heuristics devised for DPLL typically consider Δ and $\mathbf{A}$ for their decisions. In contrast, heuristics devised for Conflict-Driven Clause Learning (CDCL; [1–3]) are far more interested in ∇, containing nogoods derived from conflicts. Finally, as speculated above, a heuristic tailored for the enumeration of projective solutions could pay particular attention to the set P of variables to project on. For instance, OPTSAT [10] makes use of a decision heuristic preferring minimal literals in a partially ordered set. Although OPTSAT does not aim at enumeration, a similar intervention could be used in our setting for canceling redundancies. However, we argue below that constraining the heuristic in such a way can have a drastic negative impact. Hence, we refrain from devising any ad hoc heuristic and leave the internals of SELECT($\Delta, \nabla, \mathbf{A}, P$) unspecified. As a matter of fact, the formal properties of our algorithms are largely independent of heuristics.

Projective Solution Recording. Our goal is the repetition-free enumeration of all projective solutions for a given set Δ of nogoods wrt a set P of variables. To illustrate the peculiarities, we start with a straightforward approach recording all projective solutions in order to avoid recomputation. Our enumeration algorithm is based on CDCL, but presented in terms of nogoods and thus called Conflict-Driven Nogood Learning (CDNL). It deviates from the corresponding decision algorithm, which halts at the first solution found, merely by recording computed projective solutions as nogoods and then searching for alternative solutions.

Algorithm 1 shows our first main procedure for enumerating projective solutions. Its input consists of a set Δ of nogoods, a set P of variables to project on, and a number s of projective solutions for Δ wrt P to compute. Projective solutions are obtained from assignments $\mathbf{A}$ (initialized in Line 1) that are solutions for Δ. The dynamic nogoods in ∇ (initialized in Line 2) are derived from conflicts (cf. Line 9–10). In general, nogoods in ∇ are consequences of those in Δ and may thus be deleted at any time in order to achieve polynomial space complexity. Only such nogoods that are asserting (explained below) must not be deleted from ∇, but their number is bound by the cardinality of $var(\Delta)$. Finally, the decision level dl (initialized in Line 3) counts the number of heuristically selected decision literals in $\mathbf{A}$. The global structure of Algorithm 1 is similar to the one of the decision version of CDNL (or CDCL) by iterating propagation (Line 5) and distinguishing three resulting cases: a conflict (Line 6–11), a solution (Line 12–20), or a heuristic decision (Line 22–24). Function BOOLEANCONSTRAINTPROPAGATION($\Delta \cup \nabla, \mathbf{A}$) first augments $\mathbf{A}$ with implied literals, that is, literals necessarily contained in any solution for $\Delta \cup \nabla$ that extends $\mathbf{A}$. A well-known technique to identify such literals is *unit propagation* (cf. [2, 3]); it iteratively adds complements $\overline{\sigma}$

Algorithm 1: CDNL-RECORDING

Input : A set Δ of nogoods, a set P of variables, and a number s of requested solutions.

```
1  A ← ∅                                                          // assignment
2  ∇ ← ∅                                                          // set of (dynamic) nogoods
3  dl ← 0                                                         // decision level
4  loop
5      A ← BOOLEANCONSTRAINTPROPAGATION(Δ ∪ ∇, A)
6      if ε ⊆ A for some ε ∈ Δ ∪ ∇ then                            // conflict
7          if dl = 0 then exit
8          else
9              (δ, dl) ← CONFLICTRESOLUTION(ε, Δ ∪ ∇, A)
10             ∇ ← ∇ ∪ {δ}
11             A ← A \ {σ ∈ A | dlevel(σ) > dl}
12     else if {δ ∈ Δ | ∃σ ∈ δ : σ̄ ∈ A} = Δ then                  // solution
13         S ← {σ_p ∈ A | var(σ_p) ∈ P}
14         print S
15         s ← s − 2^(|P|−|S|)
16         if s ≤ 0 or max{dlevel(σ_p) | σ_p ∈ S} = 0 then exit
17         else
18             Δ ← Δ ∪ {S}                                        // record solution (persistently)
19             dl ← max{dlevel(σ_p) | σ_p ∈ S} − 1
20             A ← A \ {σ ∈ A | dlevel(σ) > dl}
21     else
22         σ_d ← SELECT(Δ, ∇, A, P)                               // decision
23         dlevel(σ_d) ← dl ← (dl + 1)
24         A ← A ∘ σ_d
```

to $\mathbf{A}$, if $\delta \setminus \mathbf{A} = \{\sigma\}$ for some $\delta \in \Delta \cup \nabla$, until reaching a conflict or a fixpoint. In the context of ASP, propagation also includes unfounded set checks (cf. [4, 11]). In principle, other techniques, such as failed literal detection, could be applied in addition, but they are less common in CDNL (or CDCL). We next detail the cases encountered after propagation, starting with the simplest one of a decision.

Decision. As mentioned above, we do not assume any particular heuristic but stipulate for any literal σ_d returned by SELECT$(\Delta, \nabla, \mathbf{A}, P)$ that $\{\sigma_d, \overline{\sigma_d}\} \cap \mathbf{A} = \emptyset$ and $var(\sigma_d) \in var(\Delta \cup \nabla)$. That is, σ_d must be undecided and occur in the input. For every literal $\sigma \in \mathbf{A}$, $dlevel(\sigma)$ provides its decision level. Based on this, operation $\mathbf{A} \circ \sigma'$ inserts σ' as the last literal of $dlevel(\sigma')$ into $\mathbf{A}$, before any $\sigma \in \mathbf{A}$ such that $dlevel(\sigma) > dlevel(\sigma')$. A decision literal σ_d is always appended to $\mathbf{A}$ (in Line 24).

Conflict. A conflict is encountered whenever some nogood ε is violated by $\mathbf{A}$ (cf. Line 6). If no decision has been made, there is no (further) solution for Δ, and enumeration terminates (Line 7). Otherwise, a reason δ for the conflict is calculated (Line 9) and recorded as a dynamic nogood (Line 10). We assume that the nogood δ returned by CONFLICTRESOLUTION$(\varepsilon, \Delta \cup \nabla, \mathbf{A})$ is violated by $\mathbf{A}$ and contains a *Unique Implication Point* (UIP; [1, 12]), viz., there is some literal $\sigma \in \delta$ such that $dlevel(\sigma) > \max\{dlevel(\sigma') \mid \sigma' \in \delta \setminus \{\sigma\}\}$. We assume conflict resolution to work according to the *First-UIP* scheme [3, 12], resolving ε against nogoods used to derive

implied literals in ε (this is why $\mathbf{A}$ is a sequence; cf. [4, 11]) until reaching the first UIP, which is not necessarily a decision literal. Backjumping (Line 11) then returns to decision level $dl = \max\{dlevel(\sigma') \mid \sigma' \in \delta \setminus \{\sigma\}\}$, where δ implies $\overline{\sigma}$ by unit propagation. Note that δ is the single nogood in $\Delta \cup \nabla$ justifying the inclusion of $\overline{\sigma}$ in $\mathbf{A}$ at decision level dl; such a dynamic nogood is called *asserting*. Even though Algorithm 1 does not mention deletion, dynamic nogoods that are not asserting may be deleted at any time. Since there cannot be more asserting nogoods than literals in $\mathbf{A}$, this permits running the decision version of CDNL in polynomial space. Finally, by altering conflict resolution to simply return all decision literals in $\mathbf{A}$, we can mimic DPLL with Algorithm 1 (rather than explicitly flipping a decision literal, its complement is asserted). Thus, the considerations below apply also to DPLL variants for enumerating projective solutions.

Solution. The last case is that of a solution, viz., an assignment $\mathbf{A}$ containing the complement of at least one literal from each nogood (cf. Line 12). The corresponding projective solutions for Δ wrt P are represented by $\mathbf{S}$, the set of literals in $\mathbf{A}$ over variables in P (cf. Line 13). After printing $\mathbf{S}$ (Line 14), we calculate the number of projective solutions still requested (Line 15). Note that, for $P \setminus (\mathbf{A^T} \cup \mathbf{A^F}) = \{p_1, \ldots, p_k\}$, each of the 2^k sets $\mathbf{S} \cup \{\mathbf{X}_i p_i \mid 1 \leq i \leq k\}$ such that $\mathbf{X}_i \in \{\mathbf{T}, \mathbf{F}\}$ for $1 \leq i \leq k$ is a projective solution for Δ wrt P, so that $\mathbf{A}$ represents $2^{|P|-|\mathbf{S}|}$ of them. If the number of requested projective solutions have been enumerated or if all literals in $\mathbf{S}$ are implied at decision level 0 (independent of decisions), we are done with enumeration (Line 16). Otherwise, our first procedure records $\mathbf{S}$ persistently in Δ (Line 18). In fact, unlike dynamic nogoods in ∇, $\mathbf{S}$ is not a consequence of Δ because its literals belong to a solution for Δ. Hence, we must exclude the deletion of $\mathbf{S}$, and so cannot store it as a dynamic nogood in ∇. Finally, at least one literal of $\mathbf{S}$ has to be unassigned in order to enumerate any further projective solutions. This is accomplished by retracting the maximum decision level of literals in $\mathbf{S}$ as well as all greater decision levels (Line 19–20). In principle, it is also possible to backtrack further or even to restart search from scratch by retracting all decision levels except for 0. The strategy of leaving as many decision levels as possible assigned is guided by the goal of facilitating the discovery of projective solutions nearby $\mathbf{S}$. However, as with nogood deletion, restarts can optionally be included, permitting the customization of backtracking from a solution.

We proceed by stating formal properties of Algorithm 1. The first one, *termination*, follows from the termination of CDNL on unsatisfiable sets of nogoods (cf. [13] for a proof) and the fact that solutions are excluded by strengthening the original problem.

Theorem 1. *Let Δ be a finite set of nogoods, P a set of variables, and s a number. Then, we have that* CDNL-RECORDING(Δ, P, s) *terminates.*

The second property, *soundness*, is due to the condition in Line 12 of Algorithm 1.

Theorem 2. *Let Δ be a finite set of nogoods, P a set of variables, and s a number. For every $\mathbf{S}$ printed by* CDNL-RECORDING(Δ, P, s) *and every $Q \subseteq P$, we have that $\mathbf{S} \cup \{\mathbf{T}q \mid q \in Q \setminus \mathbf{S^F}\} \cup \{\mathbf{F}r \mid r \in P \setminus (Q \cup \mathbf{S^T})\}$ is a projective solution for Δ wrt P.*

The third property, *completeness*, follows from the prerequisite that any nogood in ∇ is a consequence of those in Δ. Hence, no projective solution for Δ wrt P is ever excluded by $\Delta \cup \nabla$ before it was enumerated.[1]

Theorem 3. *Let Δ be a finite set of nogoods, P a set of variables, and $P_\Delta = var(\Delta) \cap P$. For every projective solution $\mathbf{P}$ for Δ wrt P, we have that* CDNL-RECORDING$(\Delta, P_\Delta, 2^{|P_\Delta|})$ *prints some $\mathbf{S} \subseteq \mathbf{P}$.*

Finally, *redundancy-freeness* is obtained from the fact that each already enumerated projective solution is represented by a nogood $\delta \in \Delta$, so that all further solutions for Δ must contain the complement $\overline{\sigma_p}$ of at least one literal $\sigma_p \in \delta$.

Theorem 4. *Let Δ be a finite set of nogoods, P a set of variables, and s a number. For every projective solution $\mathbf{P}$ for Δ wrt P, we have that* CDNL-RECORDING(Δ, P, s) *prints some $\mathbf{S} \subseteq \mathbf{P}$ at most once.*

In the worst case, there are exponentially many (representative literal sets of) projective solutions for Δ wrt P, each of which must be recorded in some way by Algorithm 1. Thus, our next goal is revising Algorithm 1 to work in polynomial space under maintaining its properties, in particular, redundancy-freeness. The peculiarities of this task are listed next. For brevity, we refrain from giving exemplary inputs Δ and P exhibiting the listed possibilities, but it is not difficult to come up with them.

First, for a solution $\mathbf{A}$ for Δ, there can be another solution $\mathbf{B}$ for Δ differing from $\mathbf{A}$ only on variables outside P (requiring a different decision on some variable outside P).

Fact 1. *Let $\mathbf{A}$ be a solution for a set Δ of nogoods containing decision literals $\{\sigma_1, \ldots, \sigma_j\}$. It is possible that there is some solution $\mathbf{B}$ for Δ such that $\{\sigma_p \in \mathbf{A} \mid var(\sigma_p) \in P\} \subseteq \mathbf{B}$, but $\{\overline{\sigma_1}, \ldots, \overline{\sigma_j}\} \cap \mathbf{B} \neq \emptyset$. Then, if $\overline{\sigma_i} \in \mathbf{B}$ for $1 \leq i \leq j$, we have $var(\sigma_i) \notin P$. We conclude that flipping some literal(s) in $\{\sigma_i \mid 1 \leq i \leq j, var(\sigma_i) \notin P\}$ may not exclude repetitions of projective solutions for Δ wrt P.*

Second, for a solution $\mathbf{A}$ for Δ, there can be another solution $\mathbf{B}$ for Δ differing from $\mathbf{A}$ on some variable in P, but not on any decision literal in $\mathbf{A}$ over P.

Fact 2. *Let $\mathbf{A}$ be a solution for a set Δ of nogoods containing decision literals $\{\sigma_1, \ldots, \sigma_j\}$. It is possible that there is some solution $\mathbf{B}$ for Δ such that $\{\sigma_p \in \mathbf{A} \mid var(\sigma_p) \in P\} \not\subseteq \mathbf{B}$, but $\{\sigma_i \mid 1 \leq i \leq j, var(\sigma_i) \in P\} \subseteq \mathbf{B}$. Then, $\mathbf{B}$ includes the decision literals over P from $\mathbf{A}$, still covering different projective solutions for Δ wrt P. We conclude that flipping some literal(s) in $\{\sigma_i \mid 1 \leq i \leq j, var(\sigma_i) \in P\}$ may eliminate non-redundant projective solutions for Δ wrt P.*

Combining Fact 1 and 2, we observe that flipping decision literals over variables outside P does not guarantee redundancy-freeness, while flipping decision literals over P might sacrifice completeness. Hence, with a heuristic free to return an arbitrary decision literal, we do not know which literal of a solution $\mathbf{A}$ for Δ should be flipped. This obscurity could be avoided by deciding variables in P before those outside P. However, such an approach suffers from a negative proof complexity result on unsatisfiable inputs, and for hard satisfiable problems, similar declines are not unlikely.

[1] It is sufficient to consider the set P_Δ of variables occurring in both Δ and P, along with the size $2^{|P_\Delta|}$ of the power set of P_Δ.

Fact 3. *Any restricted decision heuristic that returns a literal σ_d such that $var(\sigma_d) \notin P$ only wrt assignments $\mathbf{A}$ such that $var(\Delta \setminus \{\delta \in \Delta \mid \exists \sigma \in \delta : \overline{\sigma} \in \mathbf{A}\}) \cap P \subseteq \mathbf{A^T} \cup \mathbf{A^F}$ (that is, $var(\sigma) \notin P$ holds for all undecided literals σ in not yet satisfied nogoods of Δ) incurs super-polynomially longer optimal computations than can be obtained with an unrestricted decision heuristic on certain inputs. This handicap follows from Lemma 3 in [14], showing that CDCL with decisions restricted to variables P acting as input gates of Boolean circuits has super-polynomially longer minimum-length proofs of unsatisfiability than DPLL on infinite family $\{\mathrm{EPHP}_n^{n+1}\}$ of Boolean circuits. The circuits in this family can be translated into a set Δ of nogoods [14] such that every assignment $\mathbf{A}$ satisfying $var(\Delta \setminus \{\delta \in \Delta \mid \exists \sigma \in \delta : \overline{\sigma} \in \mathbf{A}\}) \cap P \subseteq \mathbf{A^T} \cup \mathbf{A^F}$ yields an immediate conflict. We conclude that any restricted decision heuristic is doomed to return only literals σ_d such that $var(\sigma_d) \in P$; hence, it handicaps CDNL computations in the sense of Lemma 3 in [14].*

The last fact tells us that any heuristic guaranteeing redundancy-freeness (and completeness) right away must fail on certain inputs. To avoid this, we need to devise a procedure that adaptively excludes redundancies.

Projective Solution Enumeration. Our second procedure for the enumeration of projective solutions for Δ wrt P is shown in Algorithm 2. Its overall structure, iterating propagation before distinguishing the cases of conflict (Line 6–18), solution (Line 19–38), and decision (Line 40–42), is similar to our first algorithm. We thus focus on the differences between both procedures. In this regard, the progress information of Algorithm 2 involves an additional systematic backtracking level bl (initialized in Line 3). The basic idea is to gather decision literals over P at decision levels 1 to bl that are to be backtracked systematically for the sake of enumerating further non-redundant projective solutions. In this way, Algorithm 2 establishes an enumeration scheme that can be maintained in polynomial space, abolishing the need of persistent solution recording. But as mentioned above, an important objective is to avoid interference with the actual search. In particular, before any projective solutions have been found, there is no cause for enforcing systematic backtracking. Hence, systematic backtracking levels are introduced only after finding some projective solutions, but not a priori. The case of a solution is explained next.

Solution. Projective solutions for Δ wrt P are extracted from a solution $\mathbf{A}$ for Δ and counted like in the first algorithm (cf. Line 19–22). As before, enumeration terminates if enough projective solutions have been computed or if the search space has been exhausted (Line 23). If neither is the case, the treatment of the discovered projective solutions in $\mathbf{S}$ distinguishes Algorithm 2 from its predecessor that simply records $\mathbf{S}$. Let us assume that $\mathbf{S}$ has been constructed from at least one heuristically selected literal (Line 31–38), so that alternative decisions may lead to distinct projective solutions. In order to enumerate them, we must certainly flip some decision literal(s) in $\mathbf{A}$, but Fact 1 and 2 tell us that we cannot be sure about which one(s). This obscurity is now dealt with via systematic backtracking, and thus we increment bl (Line 31) in order to introduce a new systematic backtracking level. The introduction involves storing $\mathbf{S}$ in Δ, but now as a nogood $\delta(bl)$ associated with bl (Line 32–33). The other cases of Algorithm 2 are such that $\delta(bl)$ is removed from Δ as soon as bl is retracted, which establishes polynomial space complexity. Until then, $\delta(bl)$ guarantees redundancy-freeness. The next step

Algorithm 2: CDNL-PROJECTION

Input : A set Δ of nogoods, a set P of variables, and a number s of requested solutions.

```
1  A ← ∅                                                          // assignment
2  ∇ ← ∅                                                          // set of (dynamic) nogoods
3  dl ← bl ← 0                                                    // decision and (systematic) backtracking level
4  loop
5      A ← BOOLEANCONSTRAINTPROPAGATION(Δ ∪ ∇, A)
6      if ε ⊆ A for some ε ∈ Δ ∪ ∇ then                            // conflict
7          if dl = 0 then exit
8          else if dl = bl then
9              Δ ← Δ \ {δ(bl)}                                    // remove for polynomial space complexity
10             σ_d ← dliteral(bl)
11             A ← A \ {σ ∈ A | dlevel(σ) = bl}
12             dlevel(σ̄_d) ← dl ← bl ← (bl − 1)
13             A ← A ∘ σ̄_d
14         else
15             (δ, k) ← CONFLICTRESOLUTION(ε, Δ ∪ ∇, A)
16             ∇ ← ∇ ∪ {δ}
17             dl ← max{k, bl}
18             A ← A \ {σ ∈ A | dlevel(σ) > dl}
19     else if {δ ∈ Δ | ∃σ ∈ δ : σ̄ ∈ A} = Δ then                  // solution
20         S ← {σ_p ∈ A | var(σ_p) ∈ P}
21         print S
22         s ← s − 2^(|P|−|S|)
23         if s ≤ 0 or max{dlevel(σ_p) | σ_p ∈ S} = 0 then exit
24         else if max{dlevel(σ_p) | σ_p ∈ S} = bl then
25             Δ ← Δ \ {δ(bl)}                                    // remove for polynomial space complexity
26             σ_d ← dliteral(bl)
27             A ← A \ {σ ∈ A | dlevel(σ) ≥ bl}
28             dlevel(σ̄_d) ← dl ← bl ← (bl − 1)
29             A ← A ∘ σ̄_d
30         else
31             bl ← bl + 1
32             δ(bl) ← S
33             Δ ← Δ ∪ {δ(bl)}                                    // record solution (temporarily)
34             A ← A \ {σ ∈ A | dlevel(σ) ≥ bl}
35             let σ_d ∈ δ(bl) \ A in
36                 dliteral(bl) ← σ_d
37                 dlevel(σ_d) ← dl ← bl
38                 A ← A ∘ σ_d
39     else
40         σ_d ← SELECT(Δ, ∇, A, P)                               // decision
41         dlevel(σ_d) ← dl ← (dl + 1)
42         A ← A ∘ σ_d
```

consists of retracting all literals of decision levels not smaller than bl from $\mathbf{A}$ (Line 34) to make a clean cut on some unassigned literal σ_d (selected in Line 35) from $\delta(bl)$.

Recall Fact 2 telling us that flipping σ_d may eliminate non-redundant projective solutions, hence, it is taken unflipped as decision literal of bl (Line 36–38). In summary, the reassignment of a literal σ_d from $\mathbf{S}$ makes sure that not yet enumerated projective solutions are not excluded by $\mathbf{A}$ (for completeness), while the temporary inclusion of $\delta(bl)$ in Δ prohibits a recomputation of $\mathbf{S}$ (for redundancy-freeness and termination). In the subsequent iterations, Algorithm 2 first exhausts the search space for further projective solutions including σ_d, and afterwards flips σ_d to $\overline{\sigma_d}$ along with removing the then satisfied nogood $\delta(bl)$ from Δ (for polynomial space complexity). In fact, such a systematic backtracking step is performed (in Line 25–29) if the maximum decision level of literals in $\mathbf{S}$ is bl (tested in Line 24), which means that the decision literal σ_d of bl (marked before in Line 36 and recalled in Line 26) must now be flipped for enumerating any further projective solutions. Finally, note that complement $\overline{\sigma_d}$ is assigned (in Line 29) at decision level $(bl-1)$ or the new systematic backtracking level (cf. Line 28), respectively. As a matter of fact, there is no nogood in $\Delta \cup \nabla$ that implies $\overline{\sigma_d}$, so that conflict resolution (as in Line 15) cannot be applied at the new systematic backtracking level.

Conflict. As before, a conflict at decision level 0 means that there are no (further) projective solutions (Line 7). Otherwise, we now distinguish two cases: a conflict at systematic backtracking level bl (Line 9–13) or beyond bl (Line 15–18). As mentioned above, a conflict at bl cannot be analyzed because of literals in $\mathbf{A}$ lacking a reason in $\Delta \cup \nabla$. In fact, any conflict at bl is caused by $\delta(bl)$ or flipped decision literal(s) $\overline{\sigma_d}$ such that σ_d belongs to previously computed projective solutions. Unlike in Algorithm 1, such projective solutions are no longer available in Δ, and the mere reason for the presence of $\overline{\sigma_d}$ in $\mathbf{A}$ is that the search space of σ_d has been exhausted. Thus, a conflict at bl is not analyzed, and systematic backtracking proceeds as with projective solutions located at bl (compare Line 9–13 with Line 25–29). On a conflict beyond bl, conflict resolution (Line 15) returns a dynamic nogood δ (recorded in Line 16), as with Algorithm 1. The modification consists of restricting backjumping (in Line 18) to neither retracting bl nor any smaller decision level (Line 17), even if δ is asserting at a decision level $k < bl$. Note that such an assertion reassigns some literal of previously computed solutions. If this leads to a conflict, $\delta(bl)$ or some flipped literal $\overline{\sigma_d}$ at bl is involved. Then, both are retracted by a systematic backtracking step in the next iteration.

For illustration, consider Figure 1 giving a trace of Algorithm 2 on (nogoods resulting from) the rules in (1)–(12) and $P = \{p, q, r\}$. We give all five assignments $\mathbf{A}_i$ yielding either a conflict or a solution; a resulting nogood is shown below. Column dl provides the decision levels of literals in $\mathbf{A}_i$, where the systematic backtracking level bl is given in $\mathbf{bold}$. For (flipped) decision literals, column l provides the line of Algorithm 2 in which the literal has been assigned; all other literals are inferred by propagation (in Line 5). For simplicity, we do not include variables for bodies of the rules in (1)–(12) in $\mathbf{A}_i$, but note that such variables are functionally dependent. Tracing Algorithm 2, successive decisions on $\mathbf{T}y$, $\mathbf{T}p$, and $\mathbf{T}x$ give rise to the conflicting assignment $\mathbf{A}_1$ by propagation. While $\mathbf{T}q$ is needed for deriving x from the rule in (1), complementary literal $\mathbf{F}q$ is mandatory for deriving y from the rule in (5). This makes us enter conflict resolution (in Line 15), yielding nogood $\{\mathbf{T}x, \mathbf{T}y\}$ and decision level 1 to jump back to. Hence, assignment $(\mathbf{T}y)$ constitutes the basis of $\mathbf{A}_2$. Propagating with $\{\mathbf{T}x, \mathbf{T}y\}$ gives $\mathbf{F}x$; further propagation and decision literal $\mathbf{T}q$ lead to solution

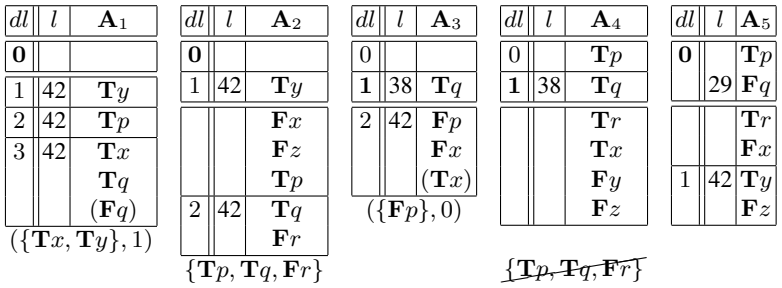

Fig. 1. Trace of Algorithm 2.

$\mathbf{A}_2 = (\mathbf{T}y, \mathbf{F}x, \mathbf{F}z, \mathbf{T}p, \mathbf{T}q, \mathbf{F}r)$, whose projective solution, $\{\mathbf{T}p, \mathbf{T}q, \mathbf{F}r\}$, is printed (in Line 21). At this point, our proceeding starts to deviate from standard CDNL. Given that the maximum decision level 2 of literals $\mathbf{T}p$, $\mathbf{T}q$, and $\mathbf{F}r$ lies above $\mathbf{0}$, we store $\{\mathbf{T}p, \mathbf{T}q, \mathbf{F}r\}$ (in Line 33) to avoid computing answer sets comprising the same projective solution. Afterwards, selecting $\mathbf{T}q$ at the new systematic backtracking level $\mathbf{1}$ makes us first explore further projective solutions containing $\mathbf{T}q$. Assignment $(\mathbf{T}q)$ is then extended to conflicting assignment $\mathbf{A}_3$, and conflict resolution results in the addition of nogood $\{\mathbf{F}p\}$, effective at decision level 0. Nonetheless, the systematic backtracking and decision level remain at $\mathbf{1}$, and further propagation yields solution $\mathbf{A}_4$, comprising projective solution $\{\mathbf{T}p, \mathbf{T}q, \mathbf{T}r\}$. The fact that all its literals are established at $\mathbf{1}$ indicates the exhaustion of the search space for $\mathbf{T}q$. Hence, the projective solution at hand is not recorded, while $\{\mathbf{T}p, \mathbf{T}q, \mathbf{F}r\}$, associated with systematic backtracking level $\mathbf{1}$, is removed to stay in polynomial space. All literals assigned at $\mathbf{1}$ are then retracted from $\mathbf{A}_4$ (in Line 27). Finally, the systematic backtracking level is decremented and the search directed to projective solutions with $\mathbf{F}q$. The construction of solution $\mathbf{A}_5$ thus starts with $\mathbf{T}p$ and $\mathbf{F}q$ at the new systematic backtracking level $\mathbf{0}$ and ends after printing the corresponding projective solution $\{\mathbf{T}p, \mathbf{F}q, \mathbf{T}r\}$. Notably, $\mathbf{A}_5$ still contains a decision literal, $\mathbf{T}y$, but flipping it cannot lead to any further projective solutions.

We conclude this section by providing formal properties of Algorithm 2. As with Algorithm 1, soundness is clear due to verifying solutions (in Line 19) before printing anything. Termination and redundancy-freeness are obtained from the fact that enumerated projective solutions are excluded either by temporarily storing them (in Line 33) or by flipping some of their literals (in Line 13 or 29) upon systematic backtracking. Finally, completeness is guaranteed because temporarily stored projective solutions do not exclude not yet enumerated ones, while a systematic backtracking step is applied only if no further projective solutions are left beyond bl. Notably, any dynamic nogood derived by resolving with temporarily stored projective solutions $\mathbf{S}$ (in Line 15) is universally valid: since the literals of $\mathbf{S}$ are not to be reestablished in the future, $\mathbf{S}$ is indeed a nogood. Given the above considerations, we conclude that Theorem 1, 2, 3, and 4 remain valid if replacing CDNL-RECORDING in their statements with CDNL-PROJECTION. Beyond this, Algorithm 2 runs in polynomial space, in view of the fact that there cannot be more temporarily stored projective solutions and asserting dynamic nogoods than literals in $\mathbf{A}$, while all other dynamic nogoods can be deleted at any time. However, it would be unfair to claim that the exponential savings in space complexity come without a cost: an introduced systematic backtracking level can only be retracted by a system-

atic backtracking step (in either Line 11 or 27), while backjumping (cf. Line 17–18) and optional restarts must leave all decision levels up to bl intact for not losing progress information.[2] However, systematic backtracking levels are introduced only after finding projective solutions, so that negative proof complexity results for procedures restricting decisions a priori [14] do not apply to Algorithm 2.

4 Experiments

We implemented our approach to solution projection within the ASP solver *clasp* (1.2.0-RC3; [4]). Our experiments consider *clasp* using four different types of enumeration: (a) its standard solution enumeration mode [11]; (b) enumeration by appeal to standard solution recording; (c) projective solution recording; (d) projective solution enumeration. Moreover, we implemented and evaluated two refinements of Algorithm 2 differing in the way selections are made in Line 35 and 40, respectively. Variant (d[h]) uses *clasp*'s BerkMin-like decision heuristic to select σ_d in Line 35 (without sign selection); otherwise, simply the first unassigned literal in $\delta(bl)$ is selected. Variant (d[p]) makes use of *clasp*'s progress saving option to direct the choice of σ_d in Line 40. Progress saving enforces sign selection according to the previously assigned truth value and thus directs search into similar search spaces as visited before (cf. [15]). Variant (d[hp]) combines both features, while (d[]) uses none of them. We refrained from testing further solvers because, to the best of our knowledge, no ASP nor SAT solver features the redundancy-free computation of projective solutions. Furthermore, ASP solvers enumerate standard solutions either via systematic backtracking, e.g., *smodels*, or like SAT solvers via solution recording, e.g., *cmodels*. The latter strategy is subsumed by *clasp* variant (b), while the former has in [11] been shown to have no edge over variant (a). Also note that we did not implement any decision heuristic specialized to preferring projected variables, as it had required another customization of *clasp*. All experiments were run on a 3.4GHz PC under Linux, each individual run restricted to 1000s time and 1GB RAM.[3]

In Table 1 and 2, we investigate the relative performance of the different enumeration approaches in terms of the proportion of projected variables. To this end, we consider two highly combinatorial benchmarks, the 11/11-Pigeons "problem" and the 15-Queens puzzle. For both of them, we gradually increase the number of projected variables (in columns #var), viz., the number of monitored pigeons or queens, respectively. The number of obtained projective solutions is given in columns #sol; the two last ones give the number of standard solutions. Columns (a)–(d[hp]) provide the runtimes of the different *clasp* variants in seconds; "$>$1000" stands for timeout. Note that #var and #sol do not affect (a) and (b), which always (attempt to) enumerate all standard solutions. At the bottom of Table 1 and 2, row $\varnothing$ provides the average runtime of each *clasp* variant.

Looking into Table 1, it is apparent that variant (b) and (c), persistently recording either standard or projective solutions, do not scale. For the last problem solved by (c), projecting to 6 out of 11 pigeons, the ratio of standard to projective solutions is 120. Furthermore, all variants of (d) are faster than standard solution enumeration (a) up to 9 out of 11 pigeons, at which point there are twice as many standard as projective

[2] This is similar to the enumeration algorithm for non-projected solutions in [11].

[3] The benchmarks are available at: `http://www.cs.uni-potsdam.de/clasp/`

#var	#sol	(a)	(b)	(c)	(d[])	(d[h])	(d[p])	(d[hp])
1	11	100.38	>1000	0.01	0.01	0.01	0.01	0.01
2	110	100.38	>1000	0.01	0.01	0.01	0.01	0.01
3	990	100.38	>1000	0.05	0.07	0.06	0.07	0.07
4	7920	100.38	>1000	0.60	0.35	0.34	0.35	0.35
5	55440	100.38	>1000	9.08	1.67	1.68	1.61	1.67
6	332640	100.38	>1000	281.05	6.34	6.32	6.50	6.34
7	1663200	100.38	>1000	>1000	20.63	20.17	21.04	20.39
8	6652800	100.38	>1000	>1000	49.97	51.20	50.10	49.18
9	19958400	100.38	>1000	>1000	88.77	88.73	89.63	91.18
10	39916800	100.38	>1000	>1000	114.17	119.36	119.12	114.82
11	39916800	100.38	>1000	>1000	114.30	113.92	116.80	118.83
∅		100.38	>1000	480.98	36.03	36.53	36.84	36.62

Table 1. Benchmark Results: 11/11-Pigeons.

#var	#sol	(a)	(b)	(c)	(d[])	(d[h])	(d[p])	(d[hp])
1	15	243.14	773.57	0.01	0.02	0.01	0.02	0.01
2	182	243.14	773.57	0.08	0.08	0.08	0.14	0.12
3	1764	243.14	773.57	0.79	0.63	0.66	1.47	1.37
4	13958	243.14	773.57	11.69	5.79	6.08	10.91	11.51
5	86360	243.14	773.57	158.40	40.71	43.71	63.76	69.88
6	369280	243.14	773.57	454.33	153.49	168.46	219.87	226.75
7	916096	243.14	773.57	>1000	331.42	357.31	444.69	437.23
8	1444304	243.14	773.57	>1000	463.46	461.78	584.59	542.46
9	1795094	243.14	773.57	>1000	512.19	523.86	652.37	577.66
10	2006186	243.14	773.57	>1000	528.36	436.70	647.49	478.34
11	2133060	243.14	773.57	>1000	525.23	407.40	616.43	450.80
12	2210862	243.14	773.57	>1000	516.56	357.22	552.67	384.30
13	2254854	243.14	773.57	>1000	462.83	322.50	496.17	356.18
14	2279184	243.14	773.57	>1000	413.72	283.82	432.62	327.35
15	2279184	243.14	773.57	>1000	250.13	250.06	245.97	249.11
∅		243.14	773.57	641.69	280.31	241.31	331.28	274.20

Table 2. Benchmark Results: 15-Queens.

solutions. For 10 and 11 pigeons, variant (a) is a bit faster than (d). In fact, (a) saves some overhead by not distinguishing projected variables within solutions. Finally, there are no significant differences between the variants of (d), given that the underlying problem is fully symmetric.

With the 15-Queens puzzle in Table 2, search becomes more important than with 11/11-Pigeons. Due to the reduced number of solutions, standard solution recording (b) now completes in less than 1000s, even though it is still slower than all enumeration schemes without persistent recording. We also see that projective solution recording (c) is the worst approach. In fact, its recorded projective solutions consist of #var literals each, while (b) stores decision literals whose number decreases the more solutions have been enumerated. For the variants of (d), we see that the number of projective solutions does not matter that much beyond 7 queens. Rather, heuristic aspects of the search start to gain importance, and variant (d[h]), which aims at placing the most critical queen first, has an edge. In contrast, progress saving alone here tends to misdirect search,

as witnessed by (d[p]). Finally, (a) enumerating standard solutions becomes more efficient than (d) from 7 queens on, where the ratio of standard to projective solutions is about 2.5. As with 11/11-Pigeons, the reason is less overhead; in particular, (a) does not even temporarily store any nogoods for excluding enumerated solutions. The reconvergence between (a) and variants of (d) at 15 queens is by virtue of an implementation trick: if the decision literal at level $(bl + 1)$ in a solution (cf. Line 31–38 in Algorithm 2) is over a variable in P, then *clasp* simply increments bl and backtracks like in Algorithm 1 (Line 19). This shortcut permits unassigning fewer variables.

The benchmarks in Table 3 belong to three different classes. The first one deals with finding Hamiltonian cycles in clumpy graphs containing n clumps of n vertices each. For each value of n, we average over 11 randomly generated instances. Note that, due to high connectivity within clumps, clumpy graphs typically allow for a vast number of Hamiltonian cycles, but finding one is still difficult for systematic backtracking methods. In our experiments, we project Hamiltonian cycles to the edges connecting different clumps, thus, reducing the number of distinct solutions by several orders of magnitude. Second, we study benchmarks stemming from consistency checks of biological networks [16]. The five categories, each containing 30 randomly generated yet biologically meaningful instances, are distinguished by the number n of vertices in a network. The task is to reestablish consistency by flipping observed variations (increase or decrease) of vertices. Solutions are then projected to the vertices whose variations have been flipped, while discarding witnesses for the consistency of the repaired network. After a repair, there are typically plenty of witnesses, so that the number of projective solutions is several orders of magnitude smaller than that of standard ones. The third class considers a variation of Ravensburger's Labyrinth game on quadratic boards with n rows and n columns, each size comprising 20 randomly generated configurations. The idea is that an avatar is guided from a starting to a goal position by moving the rows and columns of the board as well as the avatar itself, and projection consists of disregarding the moves of the avatar. It turns out that Labyrinth instances are pretty difficult to solve, and usually there are not many more standard than projective solutions.

Table 3 shows average runtimes and numbers of timeouts per benchmark category; timeouts are taken as maximum time, viz., 1000s. The rows with $\varnothing/\Sigma$ provide the average runtime and sum of timeouts for each *clasp* variant over all instances of a benchmark class and in total, respectively. For the clumpy graphs and biological networks, denoted by Clumpy and Repair in Table 3, there are far too many standard solutions to enumerate them all with either (a) or (b). Even on the smallest category of Clumpy, (a) and (b) already produce timeouts, while enumerating projective solutions with (c) or (d) is unproblematic. On the larger Clumpy categories, there is no clear winner among (c) and the variants of (d), taking also into account that difficulty and number of projective solutions vary significantly over instances. However, it appears that progress saving (d[p]) and its combination with heuristic (d[hp]) tend to help. In the Repair categories, there are hardly any differences between the variants of (d), and projective solution recording (c) is competitive too. Finally, on Labyrinth, non-projecting enumeration approaches (a) and (b) have an edge on projecting ones. This is not a surprise because there are not many more standard than projective solutions here. The disadvantages of projective solution enumeration are still not as drastic as their advantages are

Benchmark	n	(a)	(b)	(c)	(d[])	(d[h])	(d[p])	(d[hp])
Clumpy	08	204.50/02	468.48/05	0.02/0	0.02/0	0.02/0	0.02/0	0.02/0
	18	>1000/11	>1000/11	99.65/1	104.43/1	105.18/1	81.31/0	79.72/0
	20	>1000/11	>1000/11	255.04/2	254.80/2	313.22/1	219.05/1	118.95/0
	21	>1000/11	>1000/11	603.74/6	612.33/6	619.37/6	396.47/4	318.04/3
	22	>1000/11	>1000/11	144.64/1	266.72/2	275.54/2	410.98/4	321.07/3
$\varnothing/\Sigma$		840.90/46	893.70/49	220.62/10	247.66/11	262.67/10	221.57/9	167.56/6
Repair	2000	>1000/30	>1000/30	126.81/0	118.43/0	118.69/0	113.04/0	112.79/0
	2500	>1000/30	>1000/30	232.57/2	223.07/2	223.37/2	217.17/2	216.22/2
	3000	>1000/30	>1000/30	404.75/6	386.70/5	387.39/5	377.74/5	378.18/5
	3500	>1000/30	>1000/30	322.10/6	312.76/6	312.72/6	306.93/6	306.67/6
	4000	>1000/30	>1000/30	424.23/7	409.50/7	409.84/7	400.44/7	399.78/7
$\varnothing/\Sigma$		>1000/150	>1000/150	302.09/21	290.09/20	290.40/20	283.06/20	282.73/20
Labyrinth	16	52.49/0	58.46/1	59.69/1	61.72/1	59.03/1	61.54/1	59.11/1
	17	165.15/2	162.60/2	198.32/2	220.13/2	196.83/3	220.26/3	198.25/3
	18	212.59/2	218.90/2	289.84/4	298.56/3	253.06/3	286.05/3	257.38/3
	19	238.24/4	241.26/4	260.63/4	266.96/5	245.83/4	264.68/5	250.90/4
	20	319.67/5	324.43/5	355.48/6	359.51/7	343.47/6	360.33/7	346.13/6
$\varnothing/\Sigma$		197.63/13	201.13/14	232.79/17	241.38/18	219.64/17	238.57/19	222.35/17
Total $\varnothing/\Sigma$		708.24/209	718.91/213	264.68/48	266.47/49	262.20/47	257.39/48	242.17/43

Table 3. Benchmark Results: Clumpy, Repair, and Labyrinth.

on other benchmarks. Among the different (d) variants, the use of a heuristic slightly promotes (d[h]), while progress saving alone (d[p]) is not very helpful. Finally, the last row in Table 3 shows that, over all instances, projective solution enumeration variants are not far away from each other, even though (d[hp]) has a slight advance. In fact, enumeration can benefit from the incorporation of search techniques, such as a heuristic or progress saving. Their usefulness, however, depends on the particular benchmark class, so that fine-tuning is needed. Importantly, the enumeration of all projective solutions may still be possible when there are far too many standard solutions, which can be crucial for the feasibility of applications.

5 Discussion

Answer set projection is already supported by almost all ASP systems, given that `hide` and `show` directives are available in the input language. However, up to now, no ASP system was able to enumerate projective solutions without duplicates. Rather, the existing solvers exhaustively enumerate the entire set of solutions and merely restrict the output to visible atoms. This is accomplished either via systematic backtracking or via solution recording; the latter is also done by SAT solvers. To the best of our knowledge, the first dedicated solution enumeration algorithm that integrates with CDNL in polynomial space was proposed in [11] in the context of ASP; cf. variant (a) in Section 4. This algorithm turned out to be competitive for exhaustive solution enumeration, but it cannot be used for redundancy-free solution projection in view of the arguments given in Section 3. Although our new technique has also been implemented for ASP, it

is readily applicable in neighboring areas dealing with Boolean or (with the necessary adaptions) even general constraints.

From a user's perspective, the sometimes intolerable redundancy of exhaustive solution enumeration necessitates the development of wrappers feeding projections of computed solutions as constraints back into a solver. For instance, such a workaround was originally used for the diagnosis task in [16] where certificates are required for solutions. These certificates, however, do neither belong to a projective solution nor can the resulting symmetries be broken by hand. The sketched approach boils down to projective solution recording, which does not scale because of exponential space complexity. If there are too many (projective) solutions to store them all, it is of course also impossible for a user to inspect each of them individually. However, if one is interested in counting occurrences of (combinations of) literals within solutions, enumerating more solutions than can be stored explicitly is tolerable. To enable it, the duplicate-free enumeration of solutions projected to relevant parts is crucial. Finally, abolishing the need of developing wrappers to cut redundancies is already something that should help users to concentrate on the interesting aspects of their applications.

Acknowledgements. This work was funded by DFG under grant SCHA 550/8-1.

References

1. Marques-Silva, J., Sakallah, K.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers **48**(5) (1999) 506–521
2. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. Proc. DAC'01, ACM Press (2001) 530–535
3. Mitchell, D.: A SAT solver primer. Bulletin of the EATCS **85** (2005) 112–133
4. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. Proc. IJCAI'07, AAAI Press (2007) 386–392
5. Davies, J., Bacchus, F.: Using more reasoning to improve #SAT solving. Proc. AAAI'07, AAAI Press (2007) 185–190
6. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
7. Dechter, R.: Constraint Processing. Morgan Kaufmann (2003)
8. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM **7** (1960) 201–215
9. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM **5** (1962) 394–397
10. Giunchiglia, E., Maratea, M.: Solving optimization problems with DLL. Proc. ECAI'06, IOS Press (2006) 377–381
11. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set enumeration. Proc. LPNMR'07, Springer (2007) 136–148
12. Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient conflict driven learning in a Boolean satisfiability solver. Proc. ICCAD'01, IEEE Press (2001) 279–285
13. Ryan, L.: Efficient algorithms for clause-learning SAT solvers. MSc's thesis, SFU (2004)
14. Järvisalo, M., Junttila, T.: Limitations of restricted branching in clause learning. Constraints (To appear; cf. `http://www.tcs.hut.fi/~mjj/`)
15. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisability solvers. Proc. SAT'07, Springer (2007) 294–299
16. Gebser, M., Schaub, T., Thiele, S., Usadel, B., Veber, P.: Detecting inconsistencies in large biological networks with answer set programming. Proc. ICLP'08, Springer (2008) 130–144