A Proof-Producing CSP Solver

Michael Veksler and **Ofer Strichman**
mveksler@tx.technion.ac.il ofers@ie.technion.ac.il
Information systems Engineering, IE, Technion, Haifa, Israel

Abstract

PCS is a CSP solver that can produce a machine-checkable deductive proof in case it decides that the input problem is unsatisfiable. The roots of the proof may be nonclausal constraints, whereas the rest of the proof is based on resolution of signed clauses, ending with the empty clause. PCS uses parameterized, constraint-specific inference rules in order to bridge between the nonclausal and the clausal parts of the proof. The consequent of each such rule is a signed clause that is 1) logically implied by the nonclausal premise, and 2) strong enough to be the premise of the consecutive proof steps. The resolution process itself is integrated in the learning mechanism, and can be seen as a generalization to CSP of a similar solution that is adopted by competitive SAT solvers.

1 Introduction

Many problems in planning, scheduling, automatic test-generation, configuration and more, can be naturally modeled as Constraint Satisfaction Problems (CSP) (Dechter 2003), and solved with one of the many publicly available CSP solvers. The common definition of this problem refers to a set of variables over finite and discrete domains, and arbitrary constraints over these variables. The goal is to decide whether there is an assignment to the variables from their respective domains, which satisfies all the constraints. If the answer is positive the assignment that is emitted by the CSP solver can be verified easily. On the other hand a negative answer is harder to verify, since current CSP solvers do not produce a deductive proof of unsatisfiability.

In contrast, most modern CNF-based SAT solvers accompany an unsatisfiability result with a deductive proof that can be checked automatically. Specifically, they produce a *resolution proof*, which is a sequence of application of a single inference rule, namely the binary *resolution rule*. In the case of SAT the proof has uses other than just the ability to independently validate an unsatisfiability result. For example, there is a successful SAT-based model-checking algorithm which is based on deriving interpolants from the resolution proof (Henzinger et al. 2004).

Unlike SAT solvers, CSP solvers do not have the luxury of handling clausal constraints. They need to handle constraints such as $a < b + 5$, *allDifferent*(x, y, z), $a \neq b$, and so on. However, we argue that the effect of a constraint in a given state can always be replicated with a *signed clause*, which can then be part of a resolution proof. A signed clause is a disjunction between *signed literals*. A signed literal is a unary constraint, constraining a variable to a domain of values. For example, the signed clause $(x_1 \in \{1, 2\} \vee x_2 \notin \{3\})$ constrains[1] x_1 to be in the range $[1, 2]$ or x_2 to be anything but 3. A conjunction of signed clauses is called *signed CNF*, and the problem of solving signed CNF is called *signed SAT*[2], a problem which attracted extensive theoretical research and development of tools (Liu, Kuehlmann, and Moskewicz 2003; Beckert, Hähnle, and Manyá 2000b).

In this article we describe how our arc-consistency-based CSP solver PCS (for a "Proof-producing Constraint Solver") produces deductive proofs when the formula is unsatisfiable. In order to account for propagations by general constraints it uses constraint-specific parametric inference rules. Each such rule has a constraint as a premise and a signed clause as a consequent. These consequents, which are generated during conflict analysis, are called *explanation clauses*. These clauses are logically implied by the premise, but are also strong enough to imply the same literal that the premise implies at the current state. The emitted proof is a sequence of inferences of such clauses and application of special resolution rules that are tailored for signed clauses.

Like in the case of SAT, the signed clauses that are learned as a result of analyzing conflicts serve as 'milestone' atoms in the proof, although they are not the only ones. They are generated by a repeated application of the resolution rule. The intermediate clauses that are generated in this process are discarded and hence have no effect on the solving process itself. In case the learned clause eventually participates in the proof PCS reconstructs them, by using information that it saves during the learning process. We will describe this conflict-analysis mechanism in detail in Section 3 and 4, and compare it to alternatives such as 1-UIP (Zhang et al. 2001), MVS (Liu, Kuehlmann, and Moskewicz 2003) and EFC (Katsirelos and Bacchus 2005) in Section 5. We begin, however, by describing several preliminaries such as CSP

[1]Alternative notations such as $\{1, 2\}$:x_1 and $x_1^{\{1,2\}}$ are used in the literature to denote a signed literal $x_1 \in \{1, 2\}$.

[2]Signed SAT is also called MV-SAT (i.e. Many Valued SAT).

and signed SAT, and by introducing our running example.

2 Preliminaries

2.1 The Constraint Satisfaction Problem (CSP)

A CSP is a triplet $\phi = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{V} = \langle v_1, \ldots, v_n \rangle$ is the set of problem variables, $\mathcal{D} = \langle D_1, \ldots, D_n \rangle$ is the set of their respective domains and $\mathcal{C}$ is the set of constraints over these variables. An assignment α satisfies a CSP ϕ if it satisfies all the constraints in $\mathcal{C}$ and $\forall v_i \in \mathcal{V}. \alpha(v_i) \in D_i$. A CSP is unsatisfiable if there is no assignment that satisfies it.

We will use the example below as our running example.

Example 1 *Consider three intervals of length 4 starting at a, b and c. The CSP requires that these intervals do not overlap and fit a section of length 11. This is clearly unsatisfiable as the sum of their lengths is 12. The domains of a,b and c is defined to be* $[1,8]$*. It is clear that in this case the domains do not impose an additional constraint, since none of these variables can be assigned a value larger than 8 without violating the upper-bound of 11.*

In addition our problem contains three Boolean variables x_1, x_2, x_3 *that are constrained by* $x_1 \vee x_2$, $x_1 \vee x_3$, $(x_2 \wedge x_3) \rightarrow a = 1$. *Although the problem is unsatisfiable even without the constraints over these variables, we add them since the related constraints will be helpful later on for demonstrating the learning process and showing a proof of unsatisfiability that refers only to a subset of the constraints.*

We use $NoOverlap(a, L_a, b, L_b)$ *to denote the constraint* $a + L_a \leq b \vee b + L_b \leq a$. *Overall, then, the formal definition of the CSP is:*

$$\mathcal{V} = \{a, b, c, x_1, x_2, x_3\};$$

$$\mathcal{D} = \begin{cases} D_a = D_b = D_c = [1,8], \\ D_{x_1} = D_{x_2} = D_{x_3} = \{0,1\}; \end{cases}$$

$$\mathcal{C} = \begin{cases} c_1 : NoOverlap(a,4,b,4) & c_4 : x_1 \vee x_2 \\ c_2 : NoOverlap(a,4,c,4) & c_5 : x_1 \vee x_3 \\ c_3 : NoOverlap(b,4,c,4) & c_6 : (x_2 \wedge x_3) \rightarrow a = 1 \,. \end{cases}$$

2.2 Signed clauses

A *signed literal* is a unary constraint, which means that it is a restriction on the domain of a single variable. A positive signed literal, e.g., $a \in \{1,2\}$, indicates an allowed range of values, whereas a negative one, e.g., $a \notin [1,2]$ indicates a forbidden range of values. When the domain of values referred to by a literal is a singleton, we use the equality and disequality signs instead, e.g., $b = 3$ stands for $b \in \{3\}$ and $b \neq 3$ stands for $b \notin \{3\}$. A *signed clause* is a disjunction of signed literals. For brevity we will occasionally write *literal* instead of *signed literal* and *clause* instead of *signed clause*.

Propagation of signed clauses Signed clauses are learned at run-time and may also appear in the original formulation of the problem. Our solver PCS has a propagator for signed clauses, which is naturally based on the *unit clause rule*. A clause with n literals such that $n-1$ of them are false and one is unresolved is called a unit clause. The unit clause rule simply says that the one literal which is unresolved must be asserted. After asserting this literal other clauses may become unit, which means that a chain of propagations can occur.

A clause is ignored if it contains at least one satisfied literal. If all the literals in a clause are false then propagation stops and the process of *conflict analysis* and *learning* begins. We will consider this mechanism in Section 3.

Resolution rules for signed clauses PCS uses Beckert et al.'s generalization of binary resolution to signed clauses in order to generate resolution-style proofs (Beckert, Hähnle, and Manyà 2000a). They defined the *signed binary resolution* and *simplification* rules, and implicitly relied on a third rule which we call *join literals*. In the exposition of these rules below, X and Y consist of a disjunction of zero or more literals, whereas A and B are sets of values.

signed binary resolution

$$\frac{(v \in A \vee X) \quad (v \in B \vee Y)}{(v \in (A \cap B) \vee X \vee Y)}[R_v]$$

simplification

$$\frac{(v \in \emptyset \vee Z)}{(Z)}[S_v]$$

join literals

$$\frac{((\bigvee_i v \in A_i) \vee Z)}{(v \in (\bigcup_i A_i) \vee Z)}[J_v]$$

Resolution over signed clauses gives different results for a different selection of the pivot variable. For example:

$$\frac{(a \in \{1,2\} \vee b \in \{1,2\}) \qquad (a = 3 \vee b = 3)}{(a \in \emptyset \vee b \in \{1,2\} \vee b = 3)}[R_a]$$

$$\frac{(a \in \emptyset \vee b \in \{1,2\} \vee b = 3)}{(b \in \{1,2\} \vee b = 3)}[S_a] \qquad \frac{(b \in \{1,2\} \vee b = 3)}{(b \in \{1,2,3\})}[J_a]$$

gives a different result if the pivot is b instead of a:

$$\frac{(a \in \{1,2\} \vee b \in \{1,2\}) \qquad (a = 3 \vee b = 3)}{(a \in \{1,2,3\})}[R_b + S_b + J_b] \,.$$

In practice PCS does not list applications of the 'join-literals' and 'simplification' rules, simply because they are applied very frequently and it is possible to check the proof without them, assuming this knowledge is built into the proof checker. Such a checker should apply these rules until convergence after each resolution, in order to create the premise of the next step.

3 Learning

In this section we explain the learning mechanism in PCS, and how it is used for deriving proofs of unsatisfiability based, among other things, on the resolution rules that were defined in the previous section. We begin with implication graphs, which are standard representation of the propagation process. In Section 3.2 we will show the conflict analysis algorithm.

3.1 Implication graphs and conflict clauses

A propagation process is commonly described with an *implication graph*. Figure 1 shows such a graph for our running example, beginning from the decision $x_1 = 0$. In this graph vertices describe domain updates. For example the vertex

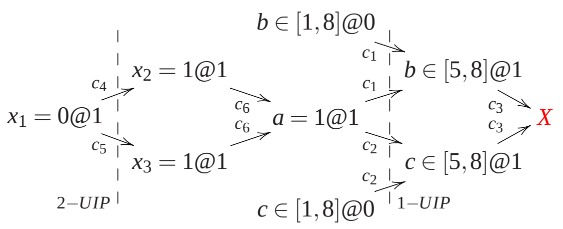

Figure 1: An implication graph corresponding to the running example.

	Constraint	Explanation clause
c_1	NoOverlap(a,4,b,4)	$\omega_1 = (a \notin [1,4] \vee b \notin [1,4])$
c_2	NoOverlap(a,4,c,4)	$\omega_2 = (a \notin [1,4] \vee c \notin [1,4])$
c_3	NoOverlap(b,4,c,4)	$\omega_3 = (b \notin [5,8] \vee c \notin [5,8])$
c_4	$x_1 \vee x_2$	$\omega_4 = (x_1 \neq 0 \vee x_2 \neq 0)$
c_5	$x_1 \vee x_3$	$\omega_5 = (x_1 \neq 0 \vee x_3 \neq 0)$
c_6	$(x_2 \wedge x_3) \rightarrow a = 1$	$\omega_6 = (x_2 \neq 1 \vee x_3 \neq 1 \vee a = 1)$

Table 1: Constraints and explanation clauses for the running example. The explanation clauses refer to the inferences depicted in the implication graph in Figure 1.

labeled with $b \in [5,8]@1$ means that the domain of b was updated to $[5,8]$ at decision level 1. A special case is a vertex labeled with an initial domain, which may only occur at decision level 0, e.g., $b \in [1,8]@0$. A conflict between clauses is signified by X. Directed edges show logical implications, and are annotated with the implying constraint. The incoming edges of each node are always labeled with the same constraint.

A constraint is called *conflicting* if it is evaluated to *false* by the current assignment. In our example the constraint $c_3 = NoOverlap(b,4,c,4)$ is conflicting under the assignment $x_1 = 0$. When such a constraint is detected the solver has to analyze the conflict and infer its cause. Traditionally this process has two roles: to apply learning by producing a new constraint and to select a decision level that the solver should backtrack to. The learned constraint has to be logically implied by the formula, and to forbid, as a minimum, the assignment that caused the conflict. In practice the goal is to produce a more general constraint, such that a larger portion of the search space is pruned. In competitive SAT solvers and CSP solvers such as EFC, the constraint is built such that it necessarily leads to further propagation right after backtracking (this constraint is called an *asserting clause* in SAT).

A standard technique for performing conflict analysis in SAT, which can also be used in CSP is called 1-UIP (for ‘first Unique Implication Point’) (Zhang et al. 2001). The dashed line marked as 1-UIP in Figure 1 marks a cut in the graph that separates the conflict node from the decision and assignments in previous decision levels. There is only one vertex immediately to the left of the line — namely the node labeled with $a = 1@1$ — which is both on the current decision level and has edges crossing this line to the right-hand side. Nodes with this property are called UIPs. UIPs, in graph-theory terms, are *dominators* of the conflicting node with respect to the decision. In other words, all paths from the decision to the conflicting node must go through each UIP. A UIP is called 1-UIP if it is the rightmost UIP. 2-UIP marks the second UIP from the right, etc.

Asserting all the literals immediately on the left of a cut necessarily leads to a conflict. For example, collecting the literals on the left of the 1-UIP cut in Figure 1 shows that $(a = 1 \wedge b \in [1,8] \wedge c \in [1,8])$ imply a conflict. To avoid the conflict the solver can generate a *conflict clause* that forbids this combination, namely $(a \neq 1 \vee b \notin [1,8] \vee c \notin [1,8])$ in this case. PCS produces stronger clauses than those that can be inferred by 1-UIP, by using resolution combined with a simplification step. This is the subject of the next subsection.

3.2 Conflict analysis and learning

Algorithm 1 describes the conflict analysis function in PCS, which is inspired by the corresponding function in SAT (Zhang and Malik 2003). This algorithm traverses the implication graph from right to left, following backwards the propagation order.

We will use the following notation in the description of the algorithm. For a node u, let $lit(u)$ denote the literal associated with u. For a literal l, let $var(l)$ denote the variable corresponding to l. For a set of nodes U, let $vars(U) = \{var(lit(u)) \mid u \in U\}$, and for a clause[3] cl, let $vars(cl) = \{var(l) \mid l \in cl\}$.

A key notion in the algorithm is that of an *explanation clause*:

Definition 1 (Explanation clause) *Let u be a node in the implication graph such that $lit(u) = l$. Let $(l_1,l)\dots(l_n,l)$ be the incoming edges of u, all of which are labeled with a constraint r. A signed clause c is an* explanation clause *of a node u if it satisfies:*

1. $r \rightarrow c$,
2. $(l_1 \wedge \cdots \wedge l_n \wedge c) \rightarrow l$.

We can see from the definition that an explanation clause is strong enough to make the same propagation of the target literal given the same input literals. Note that if the constraint r happens to be a clause, then the notions of explanation clause and *antecedent clause* that is used in SAT, coincide.

Example 2 *Explanation clauses for our running example appear in the third column in Table 1. These clauses are built with respect to the nodes in the implication graph in Figure 1. We will explain how they are generated in Section 3.3.*

The algorithm begins by computing cl, an explanation clause for the conflicting node *conflict-node*. In line 3 it computes the predecessor nodes of *conflict-node* and stores them in *pred*. The function RELVANT ($\langle nodes\rangle$, $\langle clause\rangle$) that is invoked in line 4 returns a subset N of the nodes in $\langle nodes\rangle$ that are relevant for the clause $\langle clause\rangle$, i.e., $vars(N) = vars(\langle clause\rangle)$.

[3]Here we use the standard convention by which a clause can also be seen as a set of literals.

Let *dl* be the latest decision level in *front*. The loop beginning in line 5 is guarded by STOP-CRITERION-MET(*front*), which is true in one of the following two cases:

- There is a single node in level *dl* in *front*, or
- $dl = 0$, and none of the nodes in *front* has an incoming edge.

At each iteration of the loop, CSP-ANALYZE-CONFLICT updates *cl* which, in the end of this process, will become the conflict clause. The set *front* maintains the following invariant just before line 6: *The clause cl is inconsistent with the labels in* front. Specifically, in each iteration of the loop, CSP-ANALYZE-CONFLICT:

- assigns the latest node in *front* in the propagation order to *curr-node*, and removes it from *front*,
- finds an explanation *expl* clause to *curr-node*,
- resolve the previous clause *cl* with *expl*, where $var(lit(curr\text{-}node))$ is the resolution variable (the resolution process in line 9 is as was explained in Section 2.2), and
- adds the predecessors of *curr-node* to *front* and removes redundant nodes as explained below.

The input to the function DISTINCT is a set of nodes $\langle nodes\rangle$. It outputs a maximal subset N of those such that no two nodes are labeled with the same variable. More specifically, for each variable $v \in vars(\langle nodes\rangle)$, let $U(v)$ be the maximal subset of nodes in $\langle nodes\rangle$ that are labeled with v, i.e., for each $u \in U(v)$ it holds that $var(lit(u)) = v$. Then N contains only the right-most node on the implication graph that is in $U(v)$.

The invariance above and other properties of Algorithm 1 are proved in (Veksler and Strichman 2010).

Algorithm 1 Conflict analysis

```
 1: function CSP-ANALYZE-CONFLICT
 2:     cl:= EXPLAIN(conflict-node);
 3:     pred:= PREDECESSORS(conflict-node);
 4:     front:= RELVANT (pred, cl);
 5:     while (¬STOP-CRITERION-MET(front)) do
 6:         curr-node:= LAST-NODE(front);
 7:         front:= front\curr-node;
 8:         expl:= EXPLAIN(curr-node);
 9:         cl:= RESOLVE(cl, expl, var(lit(curr-node)));
10:         pred:= PREDECESSORS(curr-node);
11:         front:= DISTINCT (RELVANT (front ∪ pred, cl));
12:     add-clause-to-database(cl);
13:     return clause-asserting-level(cl);
```

Example 3 *Table 2 demonstrates a run of Algorithm 1 on our running example. Observe that it computes the conflict clause by resolving* ω_3 *with* ω_2*, and the result of this resolution with* ω_1*. The intermediate result, namely the result of the first of these resolutions, is discarded. The resulting conflict clause* $(a \notin [1,4] \vee b \notin [1,8] \vee c \notin [1,8])$ *is stronger than what the clause would be had we used the 1-UIP method, namely* $(a \neq 1 \vee b \notin [1,8] \; \vee c \notin [1,8])$.

Line	Operation		
2	*cl*	$:= (b \notin [5,8] \vee c \notin [5,8])$	$(= \omega_3)$
3	*pred*	$:= \{b \in [5,8]@1, c \in [5,8]@1\}$	
4	*front*	$:= \{b \in [5,8]@1, c \in [5,8]@1\}$	
6	*curr-node*	$:= c \in [5,8]@1$	
7	*front*	$:= \{b \in [5,8]@1\}$	
8	*expl*	$:= (a \notin [1,4] \vee c \notin [1,4])$	$(= \omega_2)$
9	*cl*	$:= (a \notin [1,4] \vee b \notin [5,8] \vee c \notin [1,8])$	
10	*pred*	$:= \{a = 1@1, c \in [1,8]@0\}$	
11	*front*	$:= \{a = 1@1, b \in [5,8]@1, c \in [1,8]@0\}$	
6	*curr-node*	$:= b \in [5,8]@1$	
7	*front*	$:= \{a = 1@1, c \in [1,8]@0\}$	
8	*expl*	$:= (a \notin [1,4] \vee b \notin [1,4])$	$(= \omega_1)$
9	*cl*	$:= (a \notin [1,4] \vee b \notin [1,8] \vee c \notin [1,8])$	
10	*pred*	$:= \{a = 1@1, b \in [1,8]@0\}$	
11	*front*	$:= \{a = 1@1, b \in [1,8]@0, c \in [1,8]@0\}$	
12	add$((a \notin [1,4] \vee b \notin [1,8] \vee c \notin [1,8]))$		
13	return 0		

Table 2: A trace of Algorithm 1 on the running example. The horizontal lines separate iterations.

□

Saving proof data The resolution steps are saved in a list $s_1, \ldots, s_n$, in case they will be needed for the proof. Each step s_i can be defined by a clause c_i and a resolution variable v_i. The first step s_1 has an undefined resolution variable. The sequence of resolutions is well-defined by this list: the first clause is c_1, and the i-th resolution step for $i \in [2,n]$ is the resolution of c_i with the clause computed in step $i-1$, using v_i as the resolution variable. In practice PCS refrains from saving explanation clauses owing to space considerations, and instead it infers them again when printing the proof. It represents each proof step with a tuple $\langle Constraint, Rule, Pivot\rangle$, where *Constraint* is a pointer to a constraint in the constraints database, *Rule* is the parameterized inference rule by which an explanation clause can be inferred (if *Constraint* happens to be a clause then *Rule* is simply NULL), and *Pivot* is a pointer to the resolution variable. The EXPLAIN function saves this information.

3.3 Inferring explanation clauses

We now describe how explanation clauses are generated with the EXPLAIN function.

Every constraint has a propagator, which is an algorithm that deduces new facts. Every such propagator can also be written formally as an inference rule, possibly parameterized. For example, the propagator for a constraint of the form $a \leq b$ when used for inferring a new domain for a, is implemented by computing $\{x \mid x \in D(a) \wedge x \leq \max(D(b))\}$, i.e., by finding the maximal value in the domain of b, and removing values larger than this maximum from $D(a)$. The same deduction can be made by instantiating the inference rule LE(m) below with $m = \max(D(b))$.

$$\frac{a \le b}{(a \in (-\infty, m] \lor b \in [m+1, \infty))} \quad (\mathrm{LE}(m)) .$$

If, for example, the current state is $a \in [1,10]$ and $b \in [2,6]$, then the propagator will infer $a \in [1,6]$. The consequent of LE(6) implies the same literal at the current state, which means that it is an explanation clause. Table 3 contains several such inference rules that we implemented in PCS. In a proof supplement of this article (Veksler and Strichman 2010) we provide a soundness proof for these rules, and also prove that such an inference rule exists for any constraint.

One way to infer the explanation clauses, then, is to record the inference rule, and its parameter if relevant, by which the literal is inferred during propagation (when progressing to the right on the implication graph). An alternative solution, which is implemented in PCS, is to derive the inference rules only during conflict analysis, namely when traversing the implication graph from right to left. The reason that this is more efficient is that propagation by instantiation of inference rules is typically more time consuming than direct implementation of the propagator. Hence performance is improved by finding these rules *lazily*, i.e, only when they participate in the learning process and are therefor potentially needed for the proof.

4 Deriving a proof of unsatisfiability

If the formula is unsatisfiable, PCS builds a proof of unsatisfiability, beginning from the empty clause and going backwards recursively. The proof itself is printed in the correct order, i.e., from roots to the empty clause.

Recall that with each conflict clause, PCS saves the series of proof steps $s_1, \ldots, s_k$ that led to it, each of which is a tuple $\langle Constraint, Rule, Pivot \rangle$. We denote by $s_i.Cons$, $s_i.Rule$, and $s_i.Pivot$ these three elements of s_i, respectively.

Algorithm 2 receives a conflict clause as an argument — initially the empty clause — and prints its proof. It begins by traversing the proof steps $s_1, \ldots, s_k$ of the conflict-clause. Each such step leads to a recursive call if it corresponds to a conflict-clause that its proof was not yet printed. Next, it checks whether the constraint of each proof step is a clause; if it is not, then it computes its explanation with APPLYRULE. This function returns the explanation clause corresponding to the constraint, based on the rule $s_i.Rule$. After obtaining a clause, in lines 15–17 it resolves it with *cl*, the clause from the previous iteration, and prints this resolution step. Note that the clauses resolved in line 16 can be intermediate clauses that were not made into conflict clauses by the conflict analysis process.

Hence, Algorithm 2 prints a signed resolution proof, while adding an inference rule that relates each non-clausal constraint to a clausal consequent, namely the explanation clause.

Example 4 *First, we need an inference rule for NoOverlap:*

$$\frac{NoOverlap(a, l_a, b, l_b)}{(a \notin [m, n + l_b - 1] \lor b \notin [n, m + l_a - 1])} \quad (\mathrm{NO(m,n)}) ,$$

where m, n are values such that $1 - l_b \le n - m \le l_a - 1$.

Algorithm 2 Printing the proof

```
1:  function PRINTPROOF(conflict-clause)
2:      $Printed \leftarrow Printed \cup$ conflict-clause
3:      $(s_1, \ldots, s_k) \leftarrow$PROOFSTEPS(conflict-clause)
4:      for $i \leftarrow 1, k$ do
5:          if $s_i.C$ is a clause and $s_i.C \notin Printed$ then
6:              PRINTPROOF($s_i.C$)
7:      for $i \leftarrow 1, k$ do
8:          if $s_i.C$ is a clause then $expl \leftarrow s_i.C$
9:          else
10:             $expl \leftarrow$ APPLYRULE($s_i.C, s_i.Rule$)
11:             Print("Rule:", $s_i.Rule$)
12:             Print("Premise:", $s_i.C$, "Consequent:", $expl$)
13:         if $i = 1$ then $cl \leftarrow expl$
14:         else
15:             Print("Resolve", $cl, expl$, "on", $s_i.Pivot$)
16:             $cl \leftarrow Resolve(cl, expl, s_i.Pivot)$
17:             Print("Consequent:", $cl$))
```

1.	NoOverlap$(b,4,c,4)$	premise
2.	$(b \notin [5,8] \lor c \notin [5,8])$	1[NO(5,5)]
3.	NoOverlap$(a,4,c,4)$	premise
4.	$(a \notin [1,4] \lor c \notin [1,4])$	3[NO(1,1)]
5.	$(a \notin [1,4] \lor b \notin [5,8] \lor c \notin [1,8])$	2,4[Resolve(c)]
6.	NoOverlap$(a,4,b,4)$	premise
7.	$(a \notin [1,4] \lor b \notin [1,4])$	6[NO(1,1)]
8.	$(a \notin [1,4] \lor b \notin [1,8] \lor c \notin [1,8])$	5,7[Resolve(b)]
9.	$(a \notin [5,8] \lor c \notin [5,8])$	3[NO(5,5)]
10.	$(a \notin [1,8] \lor b \notin [1,8] \lor c \notin [1,8])$	8,9[Resolve(a)]
11.	$(c \in [1,8])$	premise
12.	$(a \notin [1,8] \lor b \notin [1,8])$	10,11[Resolve(c)]
13.	$(b \in [1,8])$	premise
14.	$(a \notin [1,8])$	12,13[Resolve(b)]
15.	$(a \in [1,8])$	premise
16.	$()$	14,15[Resolve(a)]

Table 4: A deductive proof of the unsatisfiability of the CSP.

Table 4 shows a proof of unsatisfiability of this CSP. This presentation is a beautification of the output of Algorithm 2. Note that the length of the proof does not change if interval sizes increase or decrease. For example, $a, b, c \in [1,80]$ and NoOverlap$(a,40,b,40)$, NoOverlap$(a,40,c,40)$, NoOverlap$(b,40,c,40)$, will require the same number of steps. Also note that the proof does not refer to the variables x_1, x_2 and x_3, since PCS found an unsatisfiable core *which does not refer to constraints over these variables.*

5 Alternative learning mechanisms

While our focus is on extracting proofs, it is also worth while to compare CSP-ANALYZE-CONFLICT to alternatives in terms of the conflict clause that it generates, as it affects both the size of the proof and the performance of the solver.

An alternative to CSP-ANALYZE-CONFLICT, recall, is collecting the literals of the 1-UIP. In Example 3 we saw that 1-UIP results in the weaker conflict clause $(a \neq 1 \lor b \notin [1,8] \lor c \notin [1,8])$. After learning this clause the solver back-

Constraint	Parameters	Inf. rule
All-diff$(v_1,\ldots,v_k)$	Domain D, and a set $V \subseteq \{v_1,\ldots,v_k\}$ such that $1+\lvert D\rvert = \lvert V\rvert$	$\dfrac{\text{All-diff}(v_1,\ldots,v_k)}{(\bigvee_{v\in V} v \notin D)}\ (AD(D,V))$
$a \neq b$	Value m	$\dfrac{a \neq b}{(a \neq m \vee b \neq m)}\ (\textsc{Ne}(m))$
$a = b$	Domain D	$\dfrac{a = b}{(a \notin D \vee b \in D)}\ (\textsc{Eq}(D))$
$a \leq b + c$	Values m, n	$\dfrac{a \leq b+c}{(a \in (-\infty, m+n] \vee b \in [m+1,\infty) \vee c \in [n+1,\infty))}\ (LE_+(m,n))$
$a = b + c$	Values l_b, u_b, l_c, u_c	$\dfrac{a = b+c}{(a \in [l_b+l_c, u_b+u_c] \vee b \notin [l_b,u_b] \vee c \notin [l_c,u_c])}\ (EQ^a_+(l_b,u_b,l_c,u_c))$

Table 3: Inference rules for some popular constraints, which PCS uses for generating explanation clauses. The last rule is a *bound consistency* propagation targeted at a.

tracks to decision level 0, in which the last two literals are false. At this point the first literal is implied, which removes the value 1 from $D(a)$, giving $D'(a) = [2,8]$. In contrast, Algorithm 1 produces the clause $(a \notin [1,4] \vee b \notin [1,8] \vee c \notin [1,8])$ (see line 12 in Table 2). This clause also causes a backtrack to level 0, and the first literal is implied. But this time the range of values $[1,4]$ is removed from $D(a)$, giving the smaller domain $D''(a) = [5,8]$. This example demonstrates the benefit of resolution-based conflict analysis over 1-UIP, and is consistent with the observation made in (Liu, Kuehlmann, and Moskewicz 2003).

Another alternative is the MVS algorithm, which was described in (Liu, Kuehlmann, and Moskewicz 2003) in terms of traversing the assignment stack rather than the implication graph. MVS essentially produces the same conflict clause as Algorithm 1, but it assumes that the input formula consists of signed clauses only, and hence does not need explanation clauses. We find Algorithm 1 clearer than MVS as its description is much shorter and relies on the implication graph rather than on the assignment stack. Further, it facilitates adoption of well-known SAT techniques and relatively easy development of further optimizations. In (Veksler and Strichman 2010) we present several such optimizations that allow CSP-ANALYZE-CONFLICT to trim more irrelevant graph nodes and learn stronger clauses.

A third alternative is the generalized-nogoods algorithm of EFC (Katsirelos and Bacchus 2005). There are two main differences between the learning mechanisms:

- EFC generates a separate explanation of each removed *value*. PCS generates an explanation for each propagation, and hence can remove *sets* of values. This affects not only performance: PCS's conflict analysis algorithm, unlike EFC's, will work in some cases with infinite domains, e.g., intervals over real numbers.
- EFC generates an explanation eagerly, after each constraint propagation. In contrast PCS generates an explanation only in response to a conflict, and hence only for constraints that are relevant for generating the conflict clause.

Performance PCS performs reasonably well in comparison with state of the art solvers. In the CSC09 competition (Dongen, Lecoutre, and Roussel 2009), in the *n*-ary constraints categories, an early version of PCS achieved the following results, out of 14 solvers. In the 'extension' subcategory: 6-th in UNSAT, 9-th in SAT, 9-th in total. In the 'intension' subcategory: 1-st in UNSAT, 4-th in SAT, 4-th in total. We intend to publish separately detailed experimental results together with a description of the various optimizations in PCS.

References

Beckert, B.; Hähnle, R.; and Manyà, F. 2000a. The 2-sat problem of regular signed cnf formulas. In *ISMVL*, 331–336.

Beckert, B.; Hähnle, R.; and Manyá, F. 2000b. The sat problem of signed cnf formulas. 59–80.

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

Dongen, M. V.; Lecoutre, C.; and Roussel, O. 2009. Fourth international CSP solver competition. Available on the web at `http://www.cril.univ-artois.fr/CPAI09/`.

Henzinger, T. A.; Jhala, R.; Majumdar, R.; and McMillan, K. L. 2004. Abstractions from proofs. In *POPL*, 232–244.

Katsirelos, G., and Bacchus, F. 2005. Generalized nogoods in CSPs. In Veloso, M. M., and Kambhampati, S., eds., *AAAI*, 390–396. AAAI Press / The MIT Press.

Liu, C.; Kuehlmann, A.; and Moskewicz, M. W. 2003. Cama: A multi-valued satisfiability solver. In *ICCAD*, 326–333. IEEE Computer Society / ACM.

Veksler, M., and Strichman, O. 2010. A proof-producing CSP solver (a proof supplement). Technical Report IE/IS-2010-02, Industrial Engineering, Technion, Haifa, Israel.

Zhang, L., and Malik, S. 2003. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE*, 10880–10885.

Zhang, L.; Madigan, C. F.; Moskewicz, M. W.; and Malik, S. 2001. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, 279–285.