Learning general constraints in CSP

Michael Veksler Ofer Strichman

Information Systems Engineering, Technion, Haifa, Israel
`mveksler@tx.technion.ac.il` `ofers@ie.technion.ac.il`

Abstract. We present a new learning scheme for CSP solvers, which is based on learning (general) constraints rather than generalized no-goods or signed-clauses that were used in the past. The new scheme is integrated in a conflict-analysis algorithm reminiscent of a modern systematic SAT solver: it traverses backwards the conflict graph and gradually builds an asserting conflict constraint. This construction is based on new inference rules that are tailored for various pairs of constraints types, e.g., $x \leq y_1 + k_1$ and $x \geq y_2 + k_2$, or $y_1 \leq x$ and $[x, y_2] \not\subseteq [a, b]$. The learned constraint is stronger than what can be learned via signed resolution. Our experiments show that our solver HCSP backtracks orders of magnitude less than other state-of-the-art solvers, and is overall on par with the winner of this year's MiniZinc challenge.

1 Introduction

The ability of CSP solvers to learn new constraints during the solving process possibly shortens run-time by an exponential factor (see, e.g., [20]). Despite this fact, and in contrast to SAT solvers, only few CSP solvers use learning, owing to the difficulty of making it cost-effective. Learning in a limited form was present in early CSP solvers, where it was called *nogood learning* [10]. Nogoods are defined as partial assignments that cannot be extended to a full solution. Later *generalized nogoods* [20] (g-nogoods for short) were proposed, which allow *non-assignments* as well, e.g., a g-nogood $(x \not\leftarrow 1, y \leftarrow 1)$ means that an assignment in which x is assigned anything but 1 and y is assigned 1 cannot be extended to a solution. This formalism is convenient for representing knowledge obtained by propagators. The g-nogood above, for example, can result from removing 1 from the domain of x, which leads by propagation to removing 1 from the domain of y. G-nogoods may be exponentially stronger than nogoods, as shown in [20].

A more general and succinct representation of learned knowledge is in the form of *signed clauses*. Such clauses are disjunctions of *signed literals*, where a signed literal has the form $v \in D$ or $v \notin D$ (called positive and negative signed literals, respectively), where v is a variable and D is a domain of values. Beckert et al. [5] studied the satisfiability problem of signed CNF, i.e., satisfiability of a conjunction of signed clauses. They proposed an inference system, based on simplification rules and a rule for binary resolution of signed clauses:

$$\frac{((v \in A) \vee X) \quad ((v \in B) \vee Y)}{(v \in (A \cap B) \vee X \vee Y)} \quad \text{[Signed Resolution}(v)] \tag{1}$$

where X and Y consist of a disjunction of zero or more literals, A and B are sets of values, and v is called the *pivot* variable. Note that in case v is Boolean and A, B are complementary Boolean domains (e.g., $A = \{0\}, B = \{1\}$) then this rule simplifies to the standard resolution rule for propositional clauses that is used in SAT, namely the consequent becomes $(X \vee Y)$.

As we showed in an earlier publication [27], we used this rule in our CSP solver HCSP (short for HaifaCSP)[1], as part of a general learning scheme based on signed clauses. Using a special inference rule for each type of non-clausal constraint, HCSP inferred a signed clause e that *explains* a propagation by that constraint. This means that e is implied by the constraint, but at the same time is strong enough to make the same propagation as the constraint, at the same state. Using such explanations in combination with rule (1) for resolving signed clauses, HCSP can generate a signed *conflict clause* via *conflict analysis*. By construction this clause is *asserting* (i.e., it necessarily leads to additional propagation after backtracking). In contrast to the CSP solver EFC [20], which generates a g-nogood *eagerly* for each removed *value*, HCSP generates a signed explanation clause *lazily*, only as part of conflict analysis. Lazy learning of g-nogoods was also implemented on top of MINION [14]. There has also been work on extending explanations with new Boolean variables, which encode equalities and inequalities [19,25], and more recently constraint-specific inference [15], such as partial sums in the case of linear constraints. In all these works there is no direct inference between general constraints.

In this article we study a different learning scheme, which is based on inference rules with non-clausal consequents. Non-clausal learning has been studied before in the context of several first-order quantifier-free theories: Pseudo-Boolean constraints (see, e.g., Sect. 22.6.4 in [6] and [11]), difference constraints [9], and integer linear constraints, e.g., [18,24]. The congruence-closure algorithm for equality logic with uninterpreted functions, which is implemented in most SMT solvers, can also be seen as inferring non-clausal constraints, since it infers new equalities. In all of these cases such learning was shown to improve the search, which motivated us to develop such a scheme for CSP, that is strongly tied to the conflict-analysis procedure. What we suggest here is very general, as it can be used with most of the constraints that are supported by modern CSP solvers, and allows non-clausal inference between different types of constraints.

Our main goal in introducing this scheme is to learn a conflict constraint that is logically stronger and easier to compute than its clausal counterpart. The emphasis is on the first of these goals as it may improve the search itself. To that end, we propose a generic inference rule called *Combine* that for many popular (pairs of) constraints indeed fulfills these two goals. For example, suppose that in a state in which the domains of three variables are defined by $x \in \{2, 6, 10, 14, \ldots, 30\}, y_1 \in \{8, 12, 16, 20\}, y_2 \in \{1, 2, 3, \ldots, 9\}$, the constraint $c_1 \doteq y_1 \leq x$ propagates $x \in \{10, 14, \ldots, 30\}$, which leads to a contradiction with a constraint $c_2 \doteq x \leq y_2$. During conflict-analysis, HCSP now infers from this propagation the constraint $[y_1, y_2] \subseteq [8, 9] \implies x \in [8, 9]$, which is clearly

[1] In [27] it was still called PCS, for Proof-producing Constraint Solver.

implied by c_1, c_2 (square brackets denote a range). Rewriting this constraint as $[y_1, y_2] \not\subseteq [8, 9] \vee x \in [8, 9]$, we see that each of the disjuncts has less variables than the set of input variables, which potentially makes it easier to solve. Instantiations of *Combine* always have this property. For some combinations of rules we do not use *Combine* since the result is too complicated to derive or too computationally expensive to support. In such cases we revert to clausal explanations.

Our experimental results indicates that indeed the new scheme is better than clausal explanation. For reference, we also compared HCSP to Mistral [16], CPX [4] and iZplus [13], where the last two won the second and first places, respectively, in the 'free-search, single-core' track of the 2014 'MiniZinc Challenge' (a CSP competition). HCSP performs better than these tools in terms of average run-time and the number of runs it is able to complete within the given time limit, although in optimization problems iZplus typically finds better solutions. HCSP performs an order of magnitude less backtracks than Mistral and three orders of magnitudes less backtracks than CPX, which proves that the constraints it learns are far more effective in pruning the search.

The rest of the article is structured as follows. The next section covers background material, including the learning framework that we use and clausal explanations [27]. Sections 3 and 4 describe the new set of inference rules, the requirements from them and the proofs that they fulfill these requirements. In Sec. 3 we also explain how we use clausal explanations as a fallback solution when we are unable to infer a general constraint that satisfies the required properties. We conclude in Sec. 5 with an empirical evaluation and some proposals for future research.

2 Background

Our solver HCSP supports all the constraint types specified in the FlatZinc format [23]. The engine of HCSP adopts classical ideas from the CSP and SAT literature. We assume the reader is mostly familiar with those, and only mention several highlights briefly for lack of space. It makes a *decision* (variable ordering) by selecting a variable with the highest ratio of *score* to domain-size, where *score* is calculated similarly to Chaff's VSIDS technique [22]. This can be seen as a variant of the *dom/wdeg* strategy [7]. The value is initially chosen to be the minimal value in the domain, and after that according to the last assigned value, a technique that is typically referred to by the name *phase saving* in SAT [26]. It includes *restarts*, *learning*, and *deletion* of learnt-constraints with low activity. The rest of this section is focused on the learning mechanism.

2.1 Conflict analysis

Conflict-analysis and learning in HCSP is based on the familiar pattern of traversing backward the conflict graph and computing an asserting constraint. Conflict-analysis was used in CSP before, but only while assuming that the constraints are signed clauses, as in MVS [21], or made into signed clauses via explanations (to be described in Sect. 2.2), as in [28,12]. The conflict-analysis in HCSP is not restricted to clausal inference, and includes various adaptations and optimizations as we describe now. Alg. 1 shows pseudo-code of ANALYZE-CONFLICT as implemented in HCSP. It maintains a set of nodes F, which is initialized to the set of nodes that contradict the input constraint cc. In line 4 it performs a *relaxation* of F. Relaxation appeared first in our technical report [28]; A similar idea appeared also in [17]. Relaxation means that each node in F is 'pushed' to the left as long as the constraint $Cons$ remains conflicting. Generally this is possible when domain reductions are redundant, as demonstrated in the following example.

Example 1. Consider the constraints

$$c_1 \doteq y \geq x \quad c_2 \doteq x \geq y \quad c_3 \doteq x > y + u \; . \tag{2}$$

and the conflict graph in Fig. 1 (left).

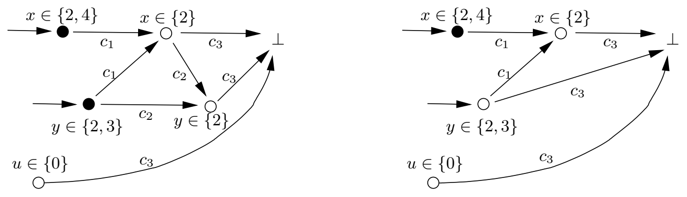


Fig. 1: Part of a conflict graph, based on the constraints in (2). Empty circles represent nodes in the set F. The left and right drawings are before and after relaxation, respectively. Relaxation discovers that the domain reduction by c_2 is not necessary for conflicting the constraint $Cons$ (c_3 in this case).

In Alg. 1, initially $Cons = c_3$, and hence after line 2 $F = \{x \in \{2\}, y \in \{2\}, u \in \{0\}\}$ (those are marked with empty circles). Relaxation in line 4 replaces in F the node $y \in \{2\}$ with the node $y \in \{2,3\}$, because the new F also contradicts the current constraint $Cons$. Fig. 1(right) shows this. The reason that this is possible is that the domain reduction by c_2 is redundant in the current state, because when $u = 0$, c_3 is capable of removing this value by itself. Such cases appear frequently, because the order in which constraints are processed is not optimal. $\square$

Relaxation is necessary for several reasons: a) preventing a situation in which the learned clause is still conflicting immediately after backtracking, instead of being asserting, b) in Sec. 4.3 we rely on relaxation in the development of some of the inference rules, and c) our experiments show that without it many more cases fall back to clausal explanations, because relaxation enables to circumvent them.

Let us return to the description of Alg. 1. In lines 5–9 ANALYZECONFLICT gradually updates the constraint $Cons$. It does so by traversing the conflict graph backwards (i.e., going left, from the conflict node towards the decision node) while updating F and the constraint *Cons* such that the following loop invariants are maintained:

1. ***Invar1.*** *Cons* contradicts the domains defined by F, and is able to detect it via propagation.[2]
2. ***Invar2.*** No two nodes in F refer to the same variable.

It should be clear that these invariants are maintained at the entry to the loop, because of the definition of F, *Cons*, and relaxation. Infer and GetNewSet are targeted towards maintaining it as will be evident later. The traversal stops in line 5 once the function stop detects that *Cons* is asserting, or that it conflicts the domains at decision level 0. In the latter case the function AssertingLevel returns -1 to the solver, which accordingly declares the CSP to be unsatisfiable. In line 8 the current constraint *Cons* is replaced with a constraint that is inferred from *Cons* itself and the antecedent constraint of a node in F. The function Infer is the main contribution of this article and will be discussed in later sections.

Algorithm 1 AnalyzeConflict receives as input the currently conflicting constraint, learns a new constraint *Cons* which is asserting (i.e., necessarily leads to further propagation), and returns the backtrack level. infer, the subject of Sect. 3–4, infers a new constraint. GetNewSet computes the new set of nodes F, as explained in the text.

```
1: function AnalyzeConflict (constraint $cc$)                ▷ $cc$ = conflicting constraint
2:     $F \leftarrow$ the set of nodes contradicting $cc$;
3:     $Cons \leftarrow cc$;
4:     $F \leftarrow$relax $(F, Cons)$;
5:     while !stop ($F$,$Cons$) do        ▷ stop if $Cons$ is asserting or UNSAT detected
6:         $pivot \leftarrow$ node of $F$ that was propagated last;
7:         $antecedent \leftarrow$ incoming constraint of $pivot$;
8:         $Cons \leftarrow$ Infer ($Cons$, $antecedent$, $pivot$, $F$);
9:         $F \leftarrow$ GetNewSet($F$, $Cons$, $pivot$);
10:        Remove from $F$ nodes referring to variables not in $Cons$.
11:        $F \leftarrow$ relax $(F, Cons)$;                ▷ Go left as long as $F$ contradicts $Cons$
12:    Add $Cons$ to the constraints database;
13:    return AssertingLevel ($Cons$,$F$); ▷ the backtracking level, or -1 if UNSAT

14: function GetNewSet(node-set $F$, node $pivot$)
15:     $F \leftarrow (F \setminus \{pivot\}) \cup$ parents of $pivot$;
16:     $F \leftarrow$ distinct ($F$);                ▷ Chooses right-most node of each variable in $F$
17:     Return $F$;
```

Let us now shift our focus to GetNewSet, which updates the set F. Initially it replaces *pivot* with its parents. In case there is more than one node in F representing the same variable, in line 16 the function distinct leaves only the right-most one. The reason that there may be multiple entries of a variable in F

[2] Detection is not a given, because not all constraints have a precise propagator, i.e., they are all sound but not all are complete. Bounds consistency is an example of such imprecise propagation.

is that a parent of *pivot* may represent a variable that already labels a different node in F because of relaxation (line 11) in a previous iteration.

2.2 Clausal Explanations

Generic explanations were used in the past (e.g., [20,14]) for learning of g-nogoods. The scheme we describe here uses inference rules specialized for each constraint type, resulting in signed clauses. Such clausal explanations are important in our context both for understanding the alternative mechanism that we used in [27] (we use it as one of the points of reference for comparing the results), and because we still use it as a fallback solution when, e.g., we reach pairs of constraints for which we do not have an inference rule. Let us begin by formally defining the notion of explanation.

Definition 1 (Clausal explanation). *Let $l_1, \ldots, l_n$ be signed literals at the current state (each literal represents the current domain of a variable), and let c be a constraint that propagates the new signed literal l, i.e., $(l_1 \wedge \ldots \wedge l_n \wedge c) \rightarrow l$. Then a clause e is an* explanation *of this propagation if the following two conditions hold:*

$$c \rightarrow e \tag{3}$$

$$(l_1 \wedge \cdots \wedge l_n \wedge e) \rightarrow l \; . \tag{4}$$

Eq. (3) guarantees that the new clause e is logically implied by an existing constraint, hence we do not lose soundness. Eq. (4) guarantees that it is still strong enough to imply the same literal. It is always possible to derive an explanation from a constraint, regardless of the constraint type [27].

Example 2. The following rule from [27] provides a clausal explanation for an inequality constraint:

$$\frac{x \leq y}{x \in (-\infty, m] \vee y \in [m+1, \infty)} \quad (\text{LE}(m)) \tag{5}$$

where m is a parameter instantiating it (the rule is sound for any m). Note that the consequent is a signed clause. Now consider two literals:

$$l_1 \doteq (x \in [1,3]), l_2 \doteq (y \in [0,2])$$

and the constraint

$$c \doteq x \leq y \; ,$$

which implies in the context of l_1, l_2 the literal $l \doteq x \in [1,2]$. Using (5) with $m = \max(y) = 2$ we obtain the explanation

$$e \doteq (x \in (-\infty, 2] \vee y \in [3, \infty)) \; ,$$

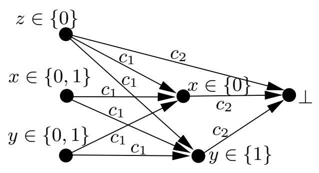
and indeed (3) and (4) hold, since $c \rightarrow e$ and $(l_1 \wedge l_2 \wedge e) \rightarrow l$. In [27] alternatives to choosing $m = \max(y)$ are discussed. □

In [27] we showed how HCSP generates a signed conflict clause with an inference system based on signed resolution (1), that is reminiscent of how SAT solvers use binary resolution. Explanations are used for bridging between non-clausal constraints and a signed clause (as in the example above), and (1) is used for resolving signed clauses.

Example 3. The following demonstrates conflict analysis with clausal explanations. In addition to (5), we will use a variant of this rule for strict inequality:

$$\frac{x < y}{x \in (-\infty, m-1] \vee y \in [m+1, \infty)} \quad \text{(L(m))} \tag{6}$$

We will also use the observation that if $c \rightarrow e$, then $(l \vee c) \rightarrow (l \vee e)$, to handle constraints with disjunctions. Let $D_x = \{0,1\}$, $D_y = \{0,1\}$, $D_z = \{0..100\}$, and $c_1 \doteq (z = 9 \vee x < y)$ $\quad c_2 \doteq (z = 10 \vee x \geq y)$. The conflict graph on the left shows the decision ($D_z = \{0\}$), and then that c_1 propagates $D_x = \{0\}$, $D_y = \{1\}$ in this order, and finally that c_2 detects a conflict. Now $F = \{z \in \{0\}, x \in \{0\}, y \in \{1\}\}$ and *pivot* $= \{y \in \{1\}\}$.



Then c_2 generates the explanation

$$(z \in \{10\} \vee x \in [1, \infty) \vee y \in (-\infty, 0])$$

based on LE(0) (see (5)), and c_1 generates the explanation

$$(z \in \{9\} \vee y \in [1, \infty) \vee x \in (-\infty, -1])$$

based on L(0) (see (6)). Resolving the two explanations on y yields

$$(z \in \{9, 10\} \vee x \notin \{0\}). \tag{7}$$

Now *pivot* $= x \in \{0\}$. c_1 explains the propagation of x with the clause

$$(z \in \{9\} \vee y \in [2, \infty) \vee x \in (-\infty, 0])$$

based on L(1). Resolving it with (7) on x yields

$$(z \in \{9, 10\} \vee x \in (\infty, -1] \vee y \in [2, \infty)) \; . \tag{8}$$

Now F is equal to the three nodes on the left. (8) is now asserting, since e.g., at the previous decision level $z \in \{9, 10\}$ is implied. $\Box$

3 Non-clausal inference: requirements

In Alg. 1 Infer is given the constraints $Cons(x, \vec{y})$ and $antecedent(x, \vec{y})$ with a joint variable x that appears at the node *pivot*, and some set of variables $\vec{y}$, which may or may not be common to both[3]. It outputs a new constraint over $x, \vec{y}$ that is assigned back into *Cons*. In the presentation that follows we will use $c_1(x, \vec{y})$ to denote $Cons(x, \vec{y})$, $c_2(x, \vec{y})$ to denote $antecedent(x, \vec{y})$, and $c^*(x, \vec{y})$ to denote the output constraint. We also define

$$c_{12}(x, \vec{y}) \doteq c_1(x, \vec{y}) \wedge c_2(x, \vec{y}).$$

Typically we will discard the parameters and write c_1, c_2, c^*, c_{12} instead.

Our first requirement from c^* is that it preserves soundness:

$$c_{12} \rightarrow c^* \; . \tag{9}$$

This guarantees that the constraint eventually learned in line 12 is inferred via sound derivations, and hence is guaranteed to be implied by the original CSP.

Let $D'_x, D'_{\vec{y}}$ denote the domains of $x, \vec{y}$ right before the propagation of c_1. Also, let $\vdash_{cp}$ denote the *provability relation* by constraints propagation, i.e., $\phi \vdash_{cp} \psi$ denotes that starting with a set of constraints and domains ϕ, the set of literals ψ is derivable through constraint propagation. Then to preserve *Invar1* (see Sec. 2), our second requirement from c^* is:

$$c^*, D'_x, D'_{\vec{y}} \vdash_{cp} \bot \; . \tag{10}$$

Finally, we aspire to find the strongest c^* that satisfies the above requirements, and which is easy to propagate.

4 Non-clausal inference: rules and their proofs

Rules R1–R7 in Table 1 are triples $\langle c_1, c_2, c^* \rangle$ that satisfy the two requirements (9) and (10). Rules R8 and R9 satisfy (9) but not necessarily (10). We use them to infer constraints, and then test whether they happen to satisfy (10). In addition, we use the following meta-rule for handling disjunctions:

$$\frac{(A \vee c_1) \quad (B \vee c_2)}{A \vee B \vee c^*} \tag{11}$$

If $\langle c_1, c_2, c^* \rangle$ satisfies (9) and (10), then so does (11). Detailed proofs for all of these rules can be found in a technical report extending this article [29].

Example 4. We now show two examples in which the rules lead to stronger learning than explanation-based learning

– Recall example 3, which yielded the conflict clause (8). Given the same conflict graph but using the meta rule (11) with pivot y, we learn instead $z \in \{9, 10\}$, which is clearly stronger.

[3] It is of course not necessarily the case that they share all the variables, but the description is simplified if we do not consider the shared and unshared variables separately, without sacrificing correctness.

– Consider a variant of the example that was described in the introduction: $x \in \{2, 6, 10, 14, \ldots, 30\}, y_1 \in \{8, 12, 16, 20\}, y_2 \in \{1, 2, 3, \ldots, 9\}$, and constraints

$$c_1 \doteq (z \in \{1\} \vee y_1 \leq x) \quad c_2 \doteq (z \in \{1\} \vee x \leq y_2) \; .$$

Suppose we make a decision $z \in \{0\}$. Then c_1 propagates $x \in \{10, 14, \ldots, 30\}$ and c_2 detects a conflict. Using rule R2 with $k1 = k2 = 0$, and the meta rule (11) we obtain:

$$(z \in \{1\} \vee x \in [8, 9] \vee [y_1, y_2] \not\subseteq [8, 9]) \; . \tag{12}$$

On the other hand if we use explanations, c_2's explanation via LE(9) is $(z \in \{1\} \vee x \in (-\infty, 9] \vee y_2 \in [10, \infty))$, c_1's explanation via LE(7) is

$$(z \in \{1\} \vee y_1 \in (-\infty, 7] \vee x \in (8, \infty]) \; ,$$

and resolving these explanations on the pivot x yields

$$(z \in \{1\} \vee x \in [8, 9] \vee y_1 \in (-\infty, 7] \vee y_2 \in [10, \infty)) \; .$$

This constraint is strictly weaker than (12) because the right disjunct of (12) implies $y_1 \leq y_2$.

$\square$

Most of the entries in the table were developed by instantiating a general inference rule called *Combine* (see Sec. 4.1 below), which satisfies these requirements. In some other cases instantiating it turned out to be too complicated and we found c^* without it. Sec. 4.3 includes proofs for some of these other rules.

Since not all combinations of rule types are supported, not all propagators are precise (i.e., logically complete) and not all rules are precise (see R8, R9 in the table), then INFER uses explanation-based inference (see Sec. 2.2) as a fallback solution. Pseudocode of INFER, which is rather self-explanatory, appears in Alg. 2.

Algorithm 2 INFER infers a new constraint c^* from c_1, c_2, which satisfies (9) and (10), the requirements listed in Sec. 3.

```
function INFER(constraint $c_1$, constraint $c_2$, node $pivot$, node-set $F$)
    $F'$ = GETNEWSET ($F, pivot$);
    if the combination of $c_1, c_2$ is supported then
        $con$ = Combine ($c_1$,$c_2$, $pivot$);          ▷ One of the rules in Table 1.
        if $F', con \vdash_{cp} \bot$ then return $con$;          ▷ $con$ satisfies Invar1
    $e_1 \leftarrow explain(c_1, parents(pivot), pivot)$;          ▷ Fallback: use explanations.
    $e_2 \leftarrow explain(c_2, F, \bot)$;
    return resolve($e_1$,$e_2$,$pivot$);          ▷ Signed resolution
```

4.1 A generic inference rule: *Combine*

Let S be some set of values. Then it is not hard to see that the following is a contradiction for any constraint $c(x, \vec{y})$:

$$c(x, \vec{y}) \wedge x \in S \wedge \forall x' \in S. \; \neg c(x', \vec{y}) \; , \tag{13}$$

	c_1	c_2	c^*
R1	$x \in X_1 \vee A_1(\vec{y})$	$x \in X_2 \vee A_2(\vec{y})$	$x \in (X_1 \cap X_2) \ \vee \ A_1(\vec{y}) \vee A_2(\vec{y})$
R2	$y_1 \le x - k_1$	$x \le y_2 - k_2$	$(x \in \left[k_1 + \min(D'_{y_1}), \max(D'_{y_2}) - k_2\right]) \ \vee$ $([y_1, y_2 - k_2 - k_1] \not\subseteq [\min(D'_{y_1}), \max(D'_{y_2}) - k_2 - k_1])$
R3	$y_1 \le x$	$[x, y_2] \not\subseteq [a, b]$	$(a > x \ge \min(D'_{y_1})) \vee$ $([y_1, y_2] \not\subseteq [\min(D'_{y_1}), b])$
R4	$x \le y_1 - k_1$	$[y_2, x - k_2] \not\subseteq [a, b]$	$(\max(D'_{y_1})\text{-}k_1 \ge x > b\text{+}k_2) \vee$ $[y_2, y_1\text{-}k_1\text{-}k_2] \not\subseteq [a, \max(b, \max(D'_{y_1}\text{-}k_1\text{-}k_2))]$
R5	$[y_1, x] \not\subseteq [a_1, b_1]$	$[x, y_2] \not\subseteq [a_2, b_2]$	$(a_2 > x > b_1) \ \vee \ ([y_1, y_2] \not\subseteq [a_1, b_2])$
R6	$[x, y] \not\subseteq [a_1, b_1]$	$[y, x] \not\subseteq [a_2, b_2]$	$(x \in (D'_y \setminus ([a_1, b_1] \cup [a_2, b_2])) \vee$ $y \notin (D'_y \cup [a_1, b_1] \cup [a_2, b_2]))$
R7	$xor(x, \vec{y})$	$xor(x, \vec{z})$	$xor(\vec{y}, \vec{z}, 1)$
R8	$y \le x + k_1$	$x \le y + k_2$	$\begin{cases} -k_1 \le x - y \le k_2 & \text{if } k_1 + k_2 \ge 0 \\ \bot & \text{otherwise} \end{cases}$
R9	$ax +$ $\sum_{i=1}^{n} a_i y_i \ge k_1$	$-ax +$ $\sum_{i=1}^{n} b_i y_i \ge k_2$	$\sum_{i=1}^{n} (a_i + b_i) x_i \ge k_1 + k_2$

Table 1: Triples $\langle c_1, c_2, c^* \rangle$ that we use for deriving conflict constraints. R1 – R7 are rules that satisfy both (9) and (10), whereas R8 – R9 are only guaranteed to satisfy (9). When using them we *test* if they also satisfy (10). The various min, max operators refer to the domain values at the point in time in which the rule is activated.

or, equivalently, that the following implication is valid:

$$c(x, \vec{y}) \rightarrow (x \notin S \vee \exists x' \in S.\ c(x', \vec{y}))\ . \qquad (14)$$

Let $\mathcal{X}$ denote the set of values of x which have no support in $D'_{\vec{y}}$:

$$\mathcal{X} = \{x' \mid \forall \vec{y'} \in D'_{\vec{y}}\ .\ \neg c_{12}(x', \vec{y'})\}\ . \qquad (15)$$

Instantiating (14) with c_{12} for c and with $\mathcal{X}$ for S yields the inference rule that we call *Combine*:

$$\boxed{\frac{c_{12}(x, \vec{y})}{(x \notin \mathcal{X} \vee \exists x' \in \mathcal{X}. c_{12}(x', \vec{y}))} \quad (\mathit{Combine}) \qquad (16)}$$

Since (16) is just an instantiation of (14), then (16) is clearly sound, and hence (9) is satisfied. To satisfy (10) we first prove logical entailment ($\models$), which is weaker than the requirement of (10) for provability ($\vdash_{cp}$).

Lemma 1. $c^*, D'_x, D'_{\vec{y}} \models \bot$.

Proof. In our case $c^* \doteq (x \notin \mathcal{X} \vee \exists x' \in \mathcal{X}.c_{12}(x', \vec{y}))$. Falsely assume that c^* is satisfied for an assignment of values $a \in D'_x, \vec{b} \in D'_{\vec{y}}$ to $x, \vec{y}$, respectively. Consider the two disjuncts of c^*:

– Suppose $x \notin \mathcal{X}$ is satisfied. Considering the definition of $\mathcal{X}$ in (15), this implies that a is supported in c_{12}, or formally

$$\exists \vec{y'} \in D'_{\vec{y}}.\ c_{12}(a, \vec{y'})\ . \tag{17}$$

Based on *Invar1* we know that $c_{12}(x, \vec{y}), D'_x, D'_{\vec{y}} \models \bot$, and hence $\forall x \in D'_x \neg\exists \vec{y} \in D'_{\vec{y}}.\ c_{12}(x, \vec{y})$, and particularly for $x = a$, $\neg\exists \vec{y} \in D'_{\vec{y}}.\ c_{12}(a, \vec{y})$, which contradicts (17).

– Now suppose $\exists x' \in \mathcal{X}.\ c_{12}(x', \vec{y})$ is satisfied. Expanding $\mathcal{X}$ and substituting $\vec{y}$ with its assignment $\vec{b}$ yields

$$\exists x'.\ \forall \vec{y'} \in D'_{\vec{y}}.\ \neg c_{12}(x', \vec{y'}) \wedge c_{12}(x', \vec{b})\ .$$

Since $\vec{b} \in D'_{\vec{y}}$ and $\neg c_{12}(x', \vec{y'})$ is satisfied for all $\vec{y'} \in D'_{\vec{y}}$, then it is satisfied for $\vec{y'} = \vec{b}$. This implies a contradiction: $\exists x'.\ \neg c_{12}(x', \vec{b}) \wedge c_{12}(x', \vec{b})$.

Hence, $x \in D'_x, \vec{y} \in D'_{\vec{y}}$ falsifies c^*, which completes our proof. $\square$

It is trivial to see that this lemma implies (10) when $\vdash_{cp}$ is precise constraint propagation. When imprecise propagation is involved, e.g., $\vdash_{cp}$ is defined by bounds consistency [8], HCSP checks whether the constraint happens to be conflicting, and if not it falls back to clausal explanation.

The relative strength of *Combine*. Two observations about the strength of *Combine* that we prove in [29] are:

– There is no alternative to $\mathcal{X}$ for replacing S in (14) that makes the resulting constraint stronger, and
– The signed clause that we obtain through the explanation mechanism—see Sec. 2.2—cannot yield a stronger consequent.

4.2 Selected rules based on instantiating *Combine*

We now instantiate *Combine* (16) with several specific constraints of interest.

Rule R2: $\boldsymbol{c_1 \doteq y_2 - x \geq k_2 \qquad c_2 \doteq x - y_1 \geq k_1}$
Expanding c_{12} in (15) yields

$$\begin{aligned}\mathcal{X} &= \left\{x \mid \forall \vec{y'} \in D'_{\vec{y}}.\ [y_2 - x < k_2\ \vee\ x - y_1 < k_1]\right\}\\ &= \left\{x \mid \max(D'_{y_2}) - x < k_2\ \vee\ x - \min(D'_{y_1}) < k_1\right\}\\ &= \left\{x \mid \max(D'_{y_2}) - k_2 < x\ \vee\ x < k_1 + \min(D'_{y_1})\right\}\ .\end{aligned}$$

The complement of $\mathcal{X}$ can be written as

$$\mathcal{X}^c = \left[k_1 + \min(D'_{y_1}),\ \max(D'_{y_2}) - k_2\right] \ . \tag{18}$$

Recall (15): $x \notin \mathcal{X} \vee \exists x' \in \mathcal{X}.\ c_{12}(x', \vec{y})$. The right disjunct is equal to:

$$\begin{aligned} &\exists x'.\ x' \in \mathcal{X} \wedge [y_2 - x' \geq k_2 \wedge x' - y_1 \geq k_1] \\ =\ &\exists x'.\ x' \in \mathcal{X} \wedge [y_2 - k_2 \geq x' \geq y_1 + k_1] \\ =\ &\exists x'.\ x' \in \mathcal{X} \wedge x' \in [y_1 + k_1, y_2 - k_2] \ . \end{aligned} \tag{19}$$

We use (18) to rewrite (19):

$$\exists x'.\ x' \notin \left[k_1 + \min(D'_{y_1}),\ \max(D'_{y_2}) - k_2\right] \wedge x' \in [y_1 + k_1, y_2 - k_2] \ ,$$

which implies

$$\begin{aligned} &[y_1 + k_1, y_2 - k_2] \not\subseteq \left[k_1 + \min(D'_{y_1}),\ \max(D'_{y_2}) - k_2\right] \\ =\ &[y_1, y_2 - k_2 - k_1] \not\subseteq \left[\min(D'_{y_1}),\ \max(D'_{y_2}) - k_2 - k_1\right] \ . \end{aligned}$$

Hence, the rule is

$$\frac{y_2 - x \geq k_2 \qquad x - y_1 \geq k_1}{\begin{array}{c}\big(x \in \left[k_1 + \min(D'_{y_1}),\ \max(D'_{y_2}) - k_2\right] \ \vee \\ [y_1,\ y_2 - k_2 - k_1] \not\subseteq \left[\min(D'_{y_1}),\ \max(D'_{y_2}) - k_2 - k_1\right]\big)\end{array}} \tag{20}$$

Rule R7: $c_1 \doteq xor(x, \vec{y}) \qquad c_2 \doteq xor(x, \vec{z})$

Here $\vec{y} \doteq y_1, \ldots, y_n$ and $\vec{z} \doteq z_1, \ldots, z_m$. We assume that $\vec{y}$ and $\vec{z}$ are fully assigned and x is not; we further assume that, w.l.o.g, c_1 propagates the value of x and c_2 detects a conflict.

Under these assumptions it is clear that c_1 and c_2 cannot be simultaneously satisfied by either $x = 0$ or $x = 1$, and hence by definition $\mathcal{X} = \{0, 1\}$, and Combine amounts to:

$$\frac{xor(x, \vec{y}) \qquad xor(x, \vec{z})}{x \notin \{0,1\} \vee \exists x' \in \{0,1\}.\ [xor(x', \vec{y}) \wedge xor(x', \vec{z})]} \ .$$

We can replace $x \notin \{0, 1\}$ with `false` and obtain:

$$\frac{xor(x, \vec{y}) \qquad xor(x, \vec{z})}{\exists x'.\ [xor(x', \vec{y}) \wedge xor(x', \vec{z})]} \ .$$

It is not hard to see that the consequent is equivalent to $xor(\vec{y}) = xor(\vec{z})$, and hence also to $xor(\vec{y}, \vec{z}, 1)$, which brings us to the desired rule:

$$\frac{xor(x, \vec{y}) \qquad xor(x, \vec{z})}{xor(\vec{y}, \vec{z}, 1)} \ .$$

Note that variables that are shared by $\vec{y}$ and $\vec{z}$ can be removed from the *xor* predicate without changing its value.

4.3 Selected rules not based on *Combine*

Rule R3: $\boldsymbol{c_1 \doteq y_1 \leq x \qquad c_2 \doteq [x, y_2] \not\subseteq [a, b]}$
We assume that at the point of conflict, replacing c_2 with $x \leq y_2$ makes c_{12} too weak to detect the conflict. Otherwise we simply use rule R2. Based on this assumption, which we denote by ψ, in [29] we show that here

$$\mathcal{X} = \{x' \mid x' < \min(D'_{y_1}) \vee a \leq x'\} . \tag{21}$$

We propose the following consequent:

$$c^* \doteq x \notin \mathcal{X} \vee [y_1, y_2] \not\subseteq [\min(D'_{y_1}), b] \tag{22}$$
$$= a > x \geq \min(D'_{y_1}) \vee [y_1, y_2] \not\subseteq [\min(D'_{y_1}), b] . \tag{23}$$

Note that c^* still follows our general pattern, by which the pivot is separated and not referred-to by the other disjunct. Since we cannot rely on the correctness of the general rule, we now prove that (23) satisfies (9) and (10):

– Eq. (9): Falsely assume the contrary, i.e., there are x, y_1, y_2 such that

$$a \leq \max(D'_{y_2}) \leq b \wedge \min(D'_{y_1}) \leq \max(D'_x) \wedge a \leq \max(D'_x) \wedge y_1 \leq x$$
$$\wedge [x, y_2] \not\subseteq [a, b] \wedge x \in \mathcal{X} \wedge [y_1, y_2] \subseteq [\min(D'_{y_1}), b] .$$

Expanding $\mathcal{X}$ yields

$$a \leq \max(D'_{y_2}) \leq b \wedge \min(D'_{y_1}) \leq \max(D'_x) \wedge a \leq \max(D'_x) \wedge y_1 \leq x$$
$$\wedge [x, y_2] \not\subseteq [a, b] \wedge (x < \min(D'_{y_1}) \vee a \leq x) \wedge [y_1, y_2] \subseteq [\min(D'_{y_1}), b] .$$

If $x < \min(D'_{y_1})$ then $y_1 \leq x$ implies $y_1 < \min(D'_{y_1})$ which conflicts $[y_1, y_2] \subseteq [\min(D'_{y_1}), b]$. Otherwise, $x \geq a$ and $[x, y_2] \not\subseteq [a, b]$ implies $y_2 > b$ which conflicts $[y_1, y_2] \subseteq [\min(D'_{y_1}), b]$.
– Eq. (10): Proved in [29].

To summarize, the rule is

$$\frac{y_1 \leq x \qquad [x, y_2] \not\subseteq [a, b] \qquad \psi}{a > x \geq \min(D'_{y_1}) \vee [y_1, y_2] \not\subseteq [\min(D'_{y_1}), b]} . \tag{24}$$

Rule R8: $\boldsymbol{c_1 \doteq (y \leq x + k_1) \qquad c_2 = (x \leq y + k_2)}$

Isolating $x - y$ on both sides yields $c_{12}(x, y) = -k_1 \leq x - y \leq k_2$, which is false if $k_1 + k_2 < 0$. Since it is simply a conjunction of the input constraints, then (9) and (10) are satisfied trivially.

5 Experimental results

We performed two sets of experiments as described below. All experiments were ran on a 4 core Intel® Xeon® 2.5GHz.

The 2009 CSP solver competition benchmarks. Here we used a subset of benchmarks of the Fourth International CSP Solver competition [2] (this was the last CSP competition, before the MiniZinc challenge started). Specifically out of over 7000 in the competition's satisfiability benchmark-set, we focused on the 2162 benchmarks that have at least one comparison operator from $\{<, \leq, \geq, >\}$ (the reason being that the rules in Table 1 refer to combinations of constraints based on these operators and constraints that are consequents of these rules). The CPU time limit was set to 1200 seconds. Out of memory and time-outs are called 'fails' in the discussion below.

We compared three different settings: (1) HCSP with general constraints learning based on *Combine* (from hereon—HCSP), (2) HCSP using only clause-based learning with explanations, as described in Sec. 2 (from hereon—Explain)[4], and (3) Mistral [16] latest version (1.550). Fig. 2 compares these three en-

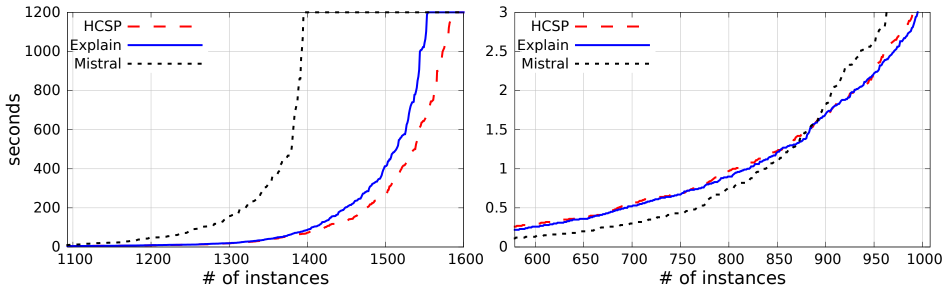


Fig. 2: Number of instances solved within the given time limit comparing HCSP, Explain, and Mistral. (left) Shows the time in linear scale; (right) A zoom-in of the left figure showing the cross-over between Mistral and HCSP occurring after 1-2 seconds.

gines. Memory was limited to 1 GiB. Number of fails in HCSP was 25% less than Mistral. Number of fails ofHCSP was 4.9% less than Explain. The average number of backtracks in HCSP is 2045, in Explain 4389, and in Mistral 49562. This drastic difference in the average backtrack-count indicates that the cost of learning is compensated-for by a better search.

MiniZinc benchmarks. Given the recent results of the MiniZinc challenge, we compared HCSP and Explain to CPX and iZplus, which won the second and first places, respectively, of the 'free-search/single-core' track of the MiniZinc challenge [3].[5] iZplus is based on the iZ-C constraint programming library,

[4] We emphasize that this is a far-improved engine in comparison to [27], owing to numerous optimizations that are beyond the scope of the current article.

[5] An early version of HCSP also participated in that competition and reached the 5th place. Since then we improved HCSP in multiple ways, including better data-

and includes stochastic local search for optimization problems. CPX is based on a lazy clause generation (i.e., lazy reduction to SAT). We used all the 100 benchmarks of the competition, 75 of which are optimization problems. The time-limit was set to 1800 sec., and the memory limit to 3GB. All benchmarks were converted to the FlatZinc format prior to benchmarking. The following table summarizes the results:

	time (avg.)	*time* (med.)	*backtracks*	*success*	*opt.*	*wins*
HCSP	897.8	686.3	35136.2	95	25	35
EXPLAIN	965.0	1232.1	37206.2	93	25	39
CPX	1055.0	1786.5	18451225.5	85	20	30
IZPLUS	972.7	1475.5		88	25	59

Detailed results can be found in [1]. The columns should be interpreted as follows: *time* (avg. and median) – the number of seconds in all benchmarks, including time-out and memout cases; *backtracks* – the number of backtracks in benchmarks in which all engines finished successfully before the time-out[6]; *success* – the number of instances solved within time and memory limits, and in the case of optimization problems found a feasible solution (but not necessarily optimal); *Opt.* – the number of instances in which the solver reached optimality and proved it; *wins* – the number of optimization instances in which the solver reached the best value among the four contenders (ties are counted).

The results show that IZPLUS has more wins than HCSP, and in all other criteria HCSP is better (the *wins* column depends on the contenders. If HCSP and EXPLAIN are on their own, the former wins 65 times and the latter only 62). It is likely that IZPLUS's wins are due to its local search part that improves the objective function once a solution is found, a component that HCSP does not have. Overall in these experiments 40% of the cases explanation was used as a fall-back solution.

Conclusion and future work We have presented a new learning scheme based on inference of general constraints. We presented the development of various inference rules that are necessary for this scheme, but it is clear that there is still a lot of work in deriving such rules for additional popular pairs of constraints which are currently not supported and force HCSP into a fallback solution. In addition, currently learning general constraints is incompatible with producing machine-checkable proofs in case the formula is unsatisfiable, in contrast to our earlier explanation-based method [27]. HCSP is written in C++, contains 23k lines of non-comment code, and its architecture enables the addition of new constraints and new rules without changing the core solver. It is free software available from [1] under the GPL license.

structures and specialized code (instead of generic) to generate explanations for 'element' global constraints, e.g., var_vector[var_i] = var_0.

[6] IZPLUS, a closed-source program, does not print this information.

References

1. The HCSP Constraint Solver web page. `http://tx.technion.ac.il/~mveksler/HCSP/`.
2. Fourth international CSP solver competition. `http://cpai.ucc.ie/09/index.html`, 2009.
3. Minizinc challenge. `http://www.minizinc.org/challenge2014/`, 2014.
4. Opturion CPX – tool description. `http://www.minizinc.org/challenge2014/description_opturion_cpx.txt`, 2014.
5. Bernhard Beckert, Reiner Hähnle, and Felip Manyá. The SAT problem of signed CNF formulas. pages 59–80, 2000.
6. Armin Biere. Bounded model checking. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 457–481. IOS Press, 2009.
7. Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In Ramon López de Mántaras and Lorenza Saitta, editors, *ECAI*, pages 146–150. IOS Press, 2004.
8. Chiu Wo Choi, Warwick Harvey, J. H. M. Lee, and Peter J. Stuckey. Finite domain bounds consistency revisited. In Abdul Sattar and Byeong Ho Kang, editors, *Australian Conference on Artificial Intelligence*, volume 4304 of *Lecture Notes in Computer Science*, pages 49–58. Springer, 2006.
9. Scott Cotton and Oded Maler. Fast and flexible difference constraint propagation for DPLL(T). In Armin Biere and Carla P. Gomes, editors, *SAT*, volume 4121 of *Lecture Notes in Computer Science*, pages 170–183. Springer, 2006.
10. Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artif. Intell.*, 41(3):273–312, 1990.
11. Heidi E. Dixon and Matthew L. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver. In Rina Dechter and Richard S. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada.*, pages 635–640. AAAI Press / The MIT Press, 2002.
12. Thibaut Feydy and Peter J. Stuckey. Lazy clause generation reengineered. In Ian P. Gent, editor, *CP*, volume 5732 of *Lecture Notes in Computer Science*, pages 352–366. Springer, 2009.
13. Toshimitsu Fujiwara. iZplus – tool description. `http://www.minizinc.org/challenge2014/description_izplus.txt`, 2014.
14. Ian P. Gent, Ian Miguel, and Neil C. A. Moore. Lazy explanations for constraint propagators. In Manuel Carro and Ricardo Peña, editors, *PADL*, volume 5937 of *Lecture Notes in Computer Science*, pages 217–233. Springer, 2010.
15. Peter J. Stuckey Geoffrey Chu. Structure based extended resolution for constraint programming. http://arxiv.org/abs/1306.4418.
16. Emmanuel Hebrard. Mistral, a constraints satisfiaction library. In *Third international CSP solver competition*, pages 31–40, 2008.
17. Siddhartha Jain, Ashish Sabharwal, and Meinolf Sellmann. A general nogood-learning framework for pseudo-boolean multi-valued SAT. In Wolfram Burgard and Dan Roth, editors, *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011.* AAAI Press, 2011.

18. Dejan Jovanovic and Leonardo Mendonça de Moura. Cutting to the chase - solving linear integer arithmetic. *J. Autom. Reasoning*, 51(1):79–108, 2013.
19. George Katsirelos and Fahiem Bacchus. Unrestricted nogood recording in CSP search. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 873–877. Springer, 2003.
20. George Katsirelos and Fahiem Bacchus. Generalized nogoods in CSPs. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 390–396. AAAI Press / The MIT Press, 2005.
21. Cong Liu, Andreas Kuehlmann, and Matthew W. Moskewicz. CAMA: A multi-valued satisfiability solver. In *ICCAD*, pages 326–333. IEEE Computer Society / ACM, 2003.
22. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. Design Automation Conference (DAC'01)*, 2001.
23. Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. MiniZinc: Towards a standard cp modelling language. In *Principles and Practice of Constraint Programming–CP 2007*, pages 529–543. Springer, 2007.
24. Robert Nieuwenhuis. The intsat method for integer linear programming. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 574–589. Springer, 2014.
25. Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.
26. Ofer Strichman. Tuning SAT checkers for bounded model checking. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 480–494. Springer, 2000.
27. Michael Veksler and Ofer Strichman. A proof-producing CSP solver. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.
28. Michael Veksler and Ofer Strichman. A proof-producing CSP solver (a proof supplement). Technical Report IE/IS-2010-02, Industrial Engineering, Technion, Haifa, Israel, Jan 2010. Available also from [1].
29. Michael Veksler and Ofer Strichman. Learning non-clausal constraints in csp (long version). Technical report, Technion, Industrial Engineering, IE/IS-2014-05, 2014. Available also from [1].