Constraint solving with a learning mechanism based on general constraints

Michael Veksler
Constraint solving with a learning mechanism based on general constraints

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Michael Veksler

Submitted to the Senate of the Technion — Israel Institute of Technology
Tamuz 5774 Haifa June 2014
The research thesis was done under the supervision of Assoc. Prof. Ofer Strichman

LIST OF PUBLICATIONS

• Chapter 2 is a reprint of the published paper [VS10a]: A Proof-Producing CSP Solver.
  Author contributions: CSP conflict analysis, lazy constraint explanations, proofs of unsatisfiability, various deduction rules used in proofs.

• The technical report [VS10b] extending [VS10a] is in Appendix B: A Proof-Producing CSP Solver (A proof supplement).
  Author contributions: Proofs of correctness for [VS10a], node relaxation\(^1\) of conflict clauses, augmented constraint explanations.

• Chapter 3 is a technical report [VS14] of a paper under submission: Learning non-clausal constraints in CSP (long version).
  Author contributions: Non-clausal conflict analysis, the Combine rule used in conflict analysis.

• The source code of the HCSP CSP solver:
  http://tx.technion.ac.il/~mveksler/HCSP/.
  Author contributions: independently written a CSP solver that, beyond implementing all of the published algorithms, can generate a Craig interpolant.

The generous financial support of the Technion is gratefully acknowledged.

\(^1\)In the original paper the term \textit{rejuvenation} was used instead of relaxation
Contents

List of Figures i
List of Tables iii
List of Algorithms v
Abstract 1
Abbreviations and Notations 3

1 Introduction 5
  1.1 The Constraint Satisfaction Problem (CSP) . . . . . . . . . . . . . . . . 5
  1.1.1 A few words about complexity . . . . . . . . . . . . . . . . . . . . . 6
  1.2 The CSP solver . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
  1.3 The CSP solving algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . 8
    1.3.1 Propagating constraints . . . . . . . . . . . . . . . . . . . . . . . 10
    1.3.2 Decision Making . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
    1.3.3 Conflict Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
  1.4 Research contribution . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

2 A Proof-Producing CSP Solver (Published in AAAI-10) 17
  2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
  2.2 Preliminaries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
    2.2.1 The Constraint Satisfaction Problem (CSP) . . . . . . . . . . . . 19
    2.2.2 Signed clauses . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
  2.3 Learning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
    2.3.1 Implication graphs and conflict clauses . . . . . . . . . . . . . . . 21
    2.3.2 Conflict analysis and learning . . . . . . . . . . . . . . . . . . . . . 22
    2.3.3 Inferring explanation clauses . . . . . . . . . . . . . . . . . . . . . 26
  2.4 Deriving a proof of unsatisfiability . . . . . . . . . . . . . . . . . . . . . 26
# Learning general constraints in CSP (long version)

## 3.1 Introduction

## 3.2 Background

### 3.2.1 Essentials of HCSP

### 3.2.2 Conflict analysis

### 3.2.3 Clausal Explanations

## 3.3 Non-clausal inference: requirements

## 3.4 Non-clausal inference: rules and their proofs

### 3.4.1 A generic inference rule: Combine

### 3.4.2 Selected rules based on instantiating Combine

### 3.4.3 Selected rules not based on Combine

## 3.5 Experimental results

## 4 Architecture and Capabilities of HCSP

### 4.1 The solving algorithm of HCSP

### 4.2 Propagating constraints

#### 4.2.1 The decision heuristic of HCSP

### 4.3 HCSP architecture

#### 4.3.1 HCSP Domains

#### 4.3.2 HCSP Constraints

#### 4.3.3 Decomposing the CSP into basic HCSP constraints

#### 4.3.4 Constraint watches

### 4.4 Solving optimization problems

#### 4.4.1 Introduction to optimization

### 4.5 Generating an interpolant

## Appendix

### A Constraints which HCSP supports

### B A Proof-Producing CSP Solver (A proof supplement)

#### B.1 Introduction to the Proof Supplement

#### B.2 Inference rules

##### B.2.1 Soundness proofs for the inference rules

##### B.2.2 Completeness of inference rules

#### B.3 Algorithms

##### B.3.1 CSP-ANALYZE-CONFLICT algorithm, the proof
B.4 Enhancements and optimizations .................................. 89
   B.4.1 CSP-ANALYZE-CONFLICT node rejuvenation ............... 89
   B.4.2 Augmented explanations ...................................... 90
   B.4.3 Lookahead explanations ...................................... 91

Bibliography ............................................................ 92

Hebrew Abstract ......................................................... א
   H.1 The Constraint Satisfaction Problem (CSP) ...................... א
   H.2 The CSP solver .................................................... ב
   H.3 The CSP solving algorithm ....................................... ב
      H.3.1 Constraint Propagation ................................... ב
      H.3.2 Decisions Making .......................................... ב
      H.3.3 Conflict Analysis .......................................... ב
      H.3.4 Research contribution .................................... ב

List of Figures

1.1 Example with a graph-coloring CSP ........................................ ..... 6
1.2 The solving loop in HCSP ....................................................... 10
2.1 An implication graph corresponding to the running example. ............ 21
3.1 Part of a conflict graph with relaxation .................................... 35
3.2 Comparison of number of instances solved within the given time limit with different algorithms ......................................................... 63
3.3 Comparing the number of backtracks for successful runs (log-scale). .... 64
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Constraints and explanation clauses for the running example. The explanation clauses refer to the inferences depicted in the implication graph in Figure 2.1.</td>
<td>23</td>
</tr>
<tr>
<td>2.2</td>
<td>A trace of Algorithm 2.1 on the running example. The horizontal lines separate iterations.</td>
<td>25</td>
</tr>
<tr>
<td>2.3</td>
<td>Inference rules for some popular constraints, which HCSP uses for generating explanation clauses. The last rule is a bound consistency propagation targeted at a.</td>
<td>27</td>
</tr>
<tr>
<td>2.4</td>
<td>A deductive proof of the unsatisfiability of the CSP.</td>
<td>29</td>
</tr>
<tr>
<td>3.1</td>
<td>Combination rules</td>
<td>42</td>
</tr>
<tr>
<td>A.1</td>
<td>Constraints supported by HCSP</td>
<td>80</td>
</tr>
</tbody>
</table>
List of Algorithms

1.1 The non-recursive MAC solving algorithm .................................. 9
1.2 The HCSP solving algorithm ....................................................... 10
2.1 Conflict analysis ................................................................. 24
2.2 Printing the proof ................................................................. 28
3.1 ANALYZECONFLICT algorithm in HCSP .................................. 37
3.2 COMBINE infers a new constraint $c^*$ from $c_1, c_2$, which satisfies (3.8) and (3.9), the requirements listed in Sec. 3.3. .......................... 43
4.1 The HCSP constraint propagation algorithm ................................. 66
4.2 Value ordering heuristic in HCSP ............................................. 67
4.3 Optimizing an objective variable $o$ in HCSP ............................... 73
4.4 The generation of an interpolant in HCSP .................................. 75
Abstract

The Constraint Satisfaction Problem (CSP) is a known NP-hard problem in the field of Artificial Intelligence. As part of this thesis, I developed a new CSP solver called HCSP, that has several novel capabilities and a reasoning engine that is highly competitive and on many benchmarks the fastest available. It is the first CSP solver capable of generating machine-checkable proofs in case the formula is unsatisfiable (based on an inference system called signed resolution), and also the first CSP solver that is able to generate a Craig interpolant. Both of these capabilities were so far only available in SAT solvers, and the main challenge in adapting them to CSP is the need to bridge between the general constraints comprising the input problem, and an inference system that provides the building blocks of a proof.

HCSP uses novel algorithms that work directly on CSP (i.e., not via translation to CNF SAT like some other CSP solvers) and are able to learn stronger constraints than a SAT solver. During conflict analysis, it is able to learn new non-clausal constraints. For example, from $x \leq y$ and $y \leq z$ it may infer that $x \leq z$, although $x \leq z$ is not an existing predicate in the formula (in contrast to theory propagation in SMT solvers). This capability is based on a new general inference rule, Combine, which can be seen as a generalization of CNF resolution.

The thesis is comprised of a collection of new algorithms and of known algorithms adapted from other domains (mostly SAT) to the realm of CSP.
Abbreviations and Notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSP</td>
<td>5</td>
</tr>
<tr>
<td>$\mathcal{C}$</td>
<td>5</td>
</tr>
<tr>
<td>$\mathcal{D}$</td>
<td>5</td>
</tr>
<tr>
<td>$\mathcal{Y}$</td>
<td>5</td>
</tr>
<tr>
<td>$\perp$</td>
<td>34,66</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 The Constraint Satisfaction Problem (CSP)

Many problems in engineering can be formulated as making choices that have to satisfy a set of constraints. When scheduling flight crews, one needs to consider the constraint that a passenger aircraft may fly only, e.g., with a pilot and 3 assistants on board and that they are not scheduled at the same time on another flight. When choosing a route for a car there are constraints on the possible routes and speed limit. Many times there are complicated interactions between constraints. Consider for example the problem of scheduling lectures in a university. Lectures schedule is constrained by personal preferences of lecturers, rooms that cannot accommodate more than one lecture simultaneously, students that cannot attend more than one lecture at a time, curriculum requirements, and many more. Other real-life examples of problems involving constraints are automated test generation for hardware circuits, nurse shift scheduling, container placement on a ship, car configuration, software-application dependency and update analysis, and many more.

The problem of finding a solution that satisfies all the constraints is called the Constraint Satisfaction Problem (CSP). CSP is a known NP-hard problem (we discuss complexity in some detail in Sect. 1.1.1), and is difficult in practice for a computer because the input of real problems is typically large.

Formally, a Constraint Satisfaction Problem (CSP) is defined by a triple \((\mathcal{C}, \mathcal{V}, \mathcal{D})\) where \(\mathcal{V}\) is a set of variables, \(\mathcal{D}\) is their domains, i.e., for each variable a set of its allowed values, and \(\mathcal{C}\) is a set of constraints over these variables. Effectively, a domain \(D_v \in \mathcal{D}\) can be viewed as the unary constraint \(v \in D_v\). A feasible solution is an assignment to the variables that satisfies all the constraints in \(\mathcal{C}\).

One may observe that special cases of CSP are SAT and Pseudo-Boolean (PB) problems, both of which are still NP-complete, but require a relatively simpler algorithm to
solve. In SAT all the variables are restricted to be Boolean. In PB the variables are Boolean but their values are interpreted as 0 and 1 rather than false and true, respectively, and linear constraints over these variables are allowed, e.g., $2x_1 + 3x_2 \leq 2$, where $x_1, x_2$ are Boolean. There are many use cases for both SAT and PB in the industry, and there are many dedicated solvers for these special cases of CSP.

In this work, as most other works on CSP, domains are restricted to be finite sets of integral values. For example, consider a graph coloring problem where we have to color the nodes of Figure 1.1 in two colors such that no adjacent edges share the same color. In this example we have four nodes numbered 1..4 and edges between them \{(1,2), (1,3), (2,4), (2,3), (3,4)\}. The CSP is modeled by four variables representing the colors of the four nodes, where 1 stands for red and 2 for green. The problem is then:

$$v_1 \neq v_2 \land v_1 \neq v_3 \land v_2 \neq v_4 \land v_3 \neq v_4 .$$

This can be solved, e.g., with $v_1 \leftarrow 1$, $v_2 \leftarrow 2$, $v_3 \leftarrow 2$, $v_4 \leftarrow 1$.

A previously popular representation of CSP was a graph, where variables are represented as nodes and constraints as arcs. This view is consistent with older works such as [Mac77] where a constraint may affect only two variables. The graph representation of the example graph-coloring CSP is shown in Figure 1.1.

Figure 1.1: Example with a graph-coloring CSP

This limitation to binary constraints is unlike newer works, including this one, where a constraint may affect any number of variables. Such a constraint is sometimes called a global constraint or generalized constraint, and a CSP with global constraints is sometimes called a Generalized CSP. A graph model of a generalized CSP uses hyper-arcs to represent the constraints, i.e., arcs that have more than two ends.

1.1.1 A few words about complexity

In [FV98] there is a proof that a CSP with n-ary constraints is NP-complete. This proof has a hidden assumption, which may not hold, that the time complexity for checking a constraint is polynomial. Although this is usually the case, there is nothing in our
definition of CSP\(^1\) that guarantees this. In theory, it is possible to formulate a constraint whose checking complexity is NP-complete, or worse.

For example, consider the following \#P-complete global constraint, which cannot be decomposed into a polynomial number of simpler constraints. Given a set of variables \(v_1, \ldots, v_n\) and a constant \(u\), the constraint is satisfied if the assignment to \(v_1, \ldots, v_n\) corresponds to a CNF that has not more than \(u\) solutions. Clearly propagating this constraint is in \#P.

We now explain how an assignment to \(v_1, \ldots, v_n\) can be interpreted as a particular CNF. These variables represent the stream of literals of the formula reminiscent of the DIMACS format. An assignment such as \(v_4 = 2\) indicates that the fourth literal of the CNF is \(x_2\). Similarly an assignment such as \(v_6 = -3\) indicates that the sixth literal is \(\neg x_3\). Finally, \(0\) indicates the end of the clause, e.g., \(v_5 = 0\) indicates an end of a clause at the fifth location.

Let us demonstrate this encoding with a particular example. Let \(v_1, \ldots, v_7\) be the set of variables in the constraint, and \(u = 5\) be the bound. Then the following assignment:

\[
v_1 = 2, v_2 = -3, v_3 = 0, v_4 = -2, v_5 = 3, v_6 = 1, v_7 = 0
\]

is a solution, because it corresponds to the CNF

\[
(x_2 \lor \neg x_3) \land (\neg x_2 \lor x_3 \lor x_1),
\]

which has 5 satisfying assignments.

On one hand, answering the question whether the constraint is satisfied is in \#P, not in NP. On the other hand, there are too many valid and invalid assignments for pre-computing an explicit relation. This means that a problem with this constraint is not in NP and it is impossible to formulate it as an NP problem. This implies that although CSP can represent NP-complete problems, its complexity may be outside of NP. In fact, it has no upper bound on complexity. Despite this, to the best of my knowledge, all constraints supported by CSP solvers in the public domain can be checked in polynomial time leading to NP complexity for solving practical CSP.

The following subsections explain general concepts of CSP and CSP solving through their relation to this research and their implementation in our solver HCSP.

---

\(^1\)Our definition of constraints allows them to be formulated by any means, including by a Turing machine. However, when constraints are defined as (pre-computed) relations then it is possible to implement constraint-checking in \(O(n)\) where \(n\) is the number of columns in the relation.
1.2 The CSP solver

The HaifaCSP constraint solver, or HCSP for short, was written during this research in order to explore new solving techniques. It is able to solve complex CSPs with many different types of constraints. HCSP was written afresh in C++ with no reliance on external source code (the only exception being a toolkit for parsing the XCSP-1.2 input format [RL09] used for parsing CSP benchmarks used in [DLR09]). HCSP is comprised of 60k lines of code, total, and only 25k if tests, comments, empty lines, braces and semicolons are excluded.

HCSP supports four input formats:

- XCSP-2.1 input format [RL09] used in the 2009 CSP competition [DLR09].
- MiniZinc [MSKS10] format without sets or floating point arithmetic.
- A subset of the OPB input format [MR+06] used in pseudo-Boolean optimization.
- A C-like input format with most of the C-language’s expression language, without side-effects. This input format has also support for minimization/maximization of an objective function and setting of variable domains.

HCSP supports most of the satisfiability problems of the 2009 CSP competition [DLR09]. The only exception are problems with element and cumulative constraints which are described in [BCR05]. The optimization directive of the XCSP-2 format, unlike with MiniZinc format, is not supported. Appendix A has a list of constraints which HCSP supports.

1.3 The CSP solving algorithm

The core of HCSP’s algorithm is based on MAC-3 [Mac77] (Maintain Arc Consistency) which is commonly used in CSP solvers. Although the original MAC-3 was defined over binary constraints, i.e., constraining pairs of variables, it was extended to n-ary constraints. Although many publications refer to this extension as Generalized-MAC-3 (GMAC-3)\(^2\), we will simply call it MAC from this point on.

A slightly different MAC algorithm was described in [SE97]. A non-recursive variant of this algorithm is described in Algorithm 1.1 and serves as the basis for the HCSP solving algorithm. MAC detects a conflict on line 4, which leads simple backtracking without conflict analysis. The simple backtrack-search may lead to repetitions of the

\(^2\)Some publications refer to Generalized-MAC as GAC and others as GMAC
same bad decisions (thrashing). This intuition has been reinforced by [Mit03] showing how good conflict-analysis speeds up the solver.

Algorithm 1.1 The non-recursive MAC solving algorithm

1: function MAC(\(C, V, D\))
2:     while true do
3:         \(D \leftarrow \text{Propagate}(C, V, D)\); \(\triangleright\) If ANY-EMPTY (\(D\)) then do nothing.
4:         if ANY-EMPTY (\(D\)) then
5:             backtrack_level \(\leftarrow\) current_level \(-\) 1;
6:             if backtrack_level \(\geq 0\) then
7:                 (var, value) \(\leftarrow\) decision_at_level(current_level);
7.1:                 \(D \leftarrow \text{backtrack-to}(\text{backtrack}\_\text{level})\);
7.2:                 \(D_{\text{var}} \leftarrow D_{\text{var}} \{\text{value}\}\); \(\triangleright\) This may empty the domain.
8:         else
9:             return UNSAT;
10:     end if
11:     else if ALL-ASSIGNED (\(D\)) then
12:         return \(\text{The solution is in } D\);
13:     else
14:         \(D \leftarrow \text{Decide}(D)\);
15:     end if
16: end while
17: end function

Unlike MAC, EFC [BK] and HCSP analyze conflicts and learn new constraints directly on the CSP. The solving algorithms of HCSP is depicted in Algorithm 1.2 and in Figure 1.2.

The difference between MAC in Algorithm 1.1 and HCSP in Algorithm 1.2 lies in the way conflicts are handled. At line 4 a conflict is detected, in both algorithms. At line 5 both algorithms calculate the backtracking level, i.e., the decision level to backtrack to\(^3\). MAC simply backtracks one level, while HCSP generates a conflict constraint and uses the constraint when deciding on the next backtracking level in line 5.

The HCSP backtracking process is simpler than the non-recursive MAC. In Algorithm 1.1 the additional lines 7.1, 7.3 are required to avoid repeating the same conflict, lines which are unnecessary in Algorithm 1.2. In Algorithm 1.2 the same conflict is avoided by learning a conflict constraint, instead of explicitly negating the decision.

\(^3\)A decision level \(l\) is a state at which there are \(l\) decisions. When backtracking to level \(l\) all domain modifications done since, and including, decision \(l + 1\), are reverted.
Algorithm 1.2 The HCSP solving algorithm

```python
1: function Solve-CSP(\mathcal{C}, \mathcal{V}, \mathcal{D})
2:  while true do
3:    \mathcal{D} ← Propagate (\mathcal{C}, \mathcal{V}, \mathcal{D});
4:    if any-empty (\mathcal{D}) then
5:      backtrack_level ← CSP-Analyze-Conflict (\mathcal{C}, \mathcal{V}, \mathcal{D});
6:      if backtrack_level ≥ 0 then
7:        \mathcal{D} ← BACKTRACK-TO (backtrack_level);
8:      else
9:        return UNSAT;
10:     end if
11:   else if all-assigned (\mathcal{D}) then
12:     return The solution is in \mathcal{D};
13:   else
14:     \mathcal{D} ← Decide (\mathcal{D});
15:   end if
16: end while
17: end function
```

![Diagram](image)

1.3.1 Propagating constraints

Constraint Propagation is the part of MAC that examines constraints and reduces variable domains according to the constraints. In SAT this phase is called BCP (Boolean Constraint Propagation). During this phase the solver examines constraints, one at a time, and removes values from the domains that cannot satisfy the constraint.

Consider, for example, the CSP

\[ D_{x_1} = \{2, 3, 4\}, \quad D_{x_2} = \{1, 2, 3\}, \quad x_1 \leq x_2 \]

In this case neither \( x_1 = 4 \) nor \( x_2 = 1 \) can participate in a satisfying assignment. We say that neither \( x_1 = 4 \) nor \( x_2 = 1 \) are supported by \( x_1 \leq x_2 \) in \( D_{x_1} \) and \( D_{x_2} \). When the solver
propagates \(^4\) \(x_1 \leq x_2\) it removes the unsupported values from the domains, producing new domains:

\[ D'_{x_1} = \{2, 3\}, \ D'_{x_2} = \{2, 3\} \ . \]

With \(D'_{x_1}\) and \(D'_{x_2}\) the constraint \(x_1 \leq x_2\) becomes *arc-consistent*. A constraint is arc-consistent if it supports all values in the current domains. Formally, constraint \(c(x_1, \ldots, x_n)\) is arc-consistent if:

\(\forall i \in \{1, 2, \ldots, n\} \forall x''_i \in D_{x_i} [\exists x'_1 \in D_{x_1} \exists x'_2 \in D_{x_2} \ldots \exists x'_n \in D_{x_1} . [x'_i = x''_i \land c(x'_1, \ldots, x'_n)]] \)

The CSP is said to be *arc-consistent* if all its constraints are arc-consistent. When the CSP is arc-consistent then it is no longer possible to deduce domain reductions by examining one constraint at a time.

The algorithm that propagates a given constraint is the *propagator* for that constraint. If a propagator always removes all values which are unsupported by the constraint then it is called *precise*. An imprecise propagator performs an over-approximated propagation, potentially leaving some unsupported values in the domains. An imprecise propagator never under-approximates, i.e., it never removes a possible solution to the problem.

A propagator may be imprecise for several reasons:

- Algorithmic complexity of precise propagation may be too high to be practical. For example, the constraint catalog [BCR05] contains *two_layer_edge_crossing*, *global_cardinality*, and *weighted_partial_alldiff* whose precise propagation algorithms are NP-hard.

- Memory complexity of precise propagation may be too high. For example, for domain \(D_{x_1} = D_{x_2} = D_{x_3} = \{2, 3, \ldots, 10^9\}\) and constraint \(x_1 = x_2 \times x_3\) precise propagation should reduce the domain of \(x_1\) to

\[ D'_{x_1} = \{x' \in \mathbb{N} | 2 \leq x' \leq 10^9 \land \neg IS\_PRIME(x')\} \ . \]

It is very difficult to define a set representation that can hold this domain efficiently.

- Overall efficiency may be better if the propagator is approximated. For example, the imprecise propagation given by *bounds consistency* [Dec03] has a simple and fast implementation since it has to deal only with domain bounds.

MAC propagates constraints until no propagator reduces any domain. If all propagators are precise then the propagation stops when the CSP is arc-consistent. Even though

\(^4\)In some publications, such as [Mac77], the term *Revise* is used instead of *Propagate* to denote the propagation step that makes a single constraint arc-consistent.

\(^5\)The term *arc* comes from the graph representation of CSP, which was mentioned in Section 1.1.
the solver will possibly not reach arc-consistency with imprecise propagators, it is still guaranteed to stop. The guarantee comes from the domains being finite and allowing a finite number of reductions, and since the propagation algorithm stops when nothing is propagated.

### 1.3.2 Decision Making

After MAC reaches arc-consistency, the solver selects a variable according to some variable-ordering heuristics and assigns it a value from its domain according to some value-ordering heuristics. This assignment operation is called a decision. Note that it is possible that this decision leads to a dead-end, i.e., there is no solution to the CSP constrained by the decision.

In the following example the CSP is arc-consistent:

\[
D_{x_1} = \{1, 2, 3\}, \quad D_{x_2} = D_{x_3} = \{1, 2\}, \quad x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_2 \neq x_3
\]

Assume that the solver decides \(x_1 \leftarrow 1\), which makes \(x_1 \neq x_2\) reduce the domain to \(D'_{x_2} = \{2\}\) and \(x_1 \neq x_3\) reduce the domain to \(D'_{x_3} = \{2\}\). Then \(x_2 \neq x_3\) detects a conflict, i.e., no solution is possible with these domains. The solver backtracks, undoing the decision, concludes that \(x_1 = 1\) cannot participate in any solution, and removes it from the domain. This produces a reduced CSP:

\[
D_{x_1} = \{2, 3\}, \quad D_{x_2} = D_{x_3} = \{1, 2\}, \quad x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_2 \neq x_3
\]

With this newly reduced problem, if the solver decides \(x \leftarrow 2\) then it also reaches a conflict. While undoing the decision it concludes that \(D_{x_1} = \{3\}\). Next it decides \(x_2 \leftarrow 1\) which leads to the solution \(x_1 = 3, \ x_2 = 1, \ x_3 = 2\).

Notice that this example has two very similarly bad decisions with similar conflicts. It would be good if the solver could detect the similarity and avoid taking the second bad decision. In order for this to happen the solver has to perform conflict analysis, which is discussed in Subsection 1.3.3.

### 1.3.3 Conflict Analysis

The CSP solution process involves making wrong decisions and backtracking from them. Many CSP solving algorithms were proposed that avoid repeating similar wrong decisions. Initial methods used a no-good to avoid a bad assignment, which works well in solvers that do not have constraint propagation such as BT-search [Dec03]. There were several failed attempts to have conflict analysis in solvers that propagate constraints such as MAC-CBJ [Pro95], which looks for the decision combination that causes a conflict.
It has been shown in [BR96] that MAC-CBJ is not competitive with plain MAC. In order to overcome these issues works such as [JDB00] learned no-goods to avoid repeating bad decisions efficiently. For example, a nogood \((x_1 \leftarrow 1, x_2 \leftarrow 5, x_4 \leftarrow 8)\) means that an assignment with \(x_1 = 1 \land x_2 = 5 \land x_4 = 8\) is impossible and has to be avoided.

Nogoods did not catch on and the reason, as explained in [KB05, Mit03], was that no-goods are too weak to efficiently represent causes for a conflict. Instead of restricting conflict analysis to detecting bad assignments, [KB05] proposes to extend nogoods into generalized-nogoods, or g-nogoods for short. A g-nogood adds an anti-assignment operator \(x \neq n\) to indicate that there would be a conflict only if \(x \neq n\). For example, a g-nogood \((x_1 \neq 1, x_1 \neq 2, x_2 \leftarrow 5)\) means that an assignment with \(x_1 \neq 1 \land x_1 \neq 2 \land x_2 = 5\) is impossible and has to be avoided. This can be expressed as constraint \((x_1 \in \{1, 2\} \lor x_2 \neq 5)\). The EFC [BK] solver would explain every reduced domain in terms of g-nogoods, eagerly, and resolve these g-nogoods during conflict analysis to get the cause of the conflict in terms of a g-nogood.

Another competing technique is called C-Res, i.e., Constraint-Resolution, (introduced in [Mit03]). Each time a constraint is propagated and values are removed, a Boolean clause\(^6\) is created to represent this propagation. When performing conflict analysis, the clauses representing the constraints are resolved using the standard propositional resolution. It is unspecified whether the clauses are generated during constraint propagation or only during conflict analysis.

A later technique named Lazy \(^7\) Clause Generation [OSC09], combines CSP solving techniques with SAT. This technique represents all domains and CSP variables as Boolean indicator variables and perform solving on them. It examines a constraint and if it is inconsistent with the domains then the solver generates an explanation of the inconsistency, i.e., a Boolean-CNF clause which is added to the CNF. The propagation is then performed via BCP [MMZ01b] and so is conflict analysis and learning are performed on the SAT side. The effects of this approach should be similar to the eager generation of g-nogoods in EFC with the difference that SAT solvers manage propagation and learning more efficiently.

Both lazy clause generation [OSC09] and EFC [BK] produce explanations eagerly. This creates a big collection of clauses (or g-nogoods) which may be never required for conflict analysis or for the solution. Another possible disadvantage of [OSC09] is its relatively inefficient domain representation. Each value of variable’s domain requires a Boolean variable, which makes it impractical to represent big domains. In order to be

---

\(^6\)In later works, such as [OSC09], this Boolean clause is named an explanation clause

\(^7\)Lazy refers to the fact that the CNF encoding for the whole CSP is not created up-front. The CNF encoding is created one bit at a time, i.e, lazily, when the constraints are examined in the context of the current partial assignment.
able to maintain big domains it is possible to represent them using a \( \log_2 \) representation, which requires auxiliary variables. This idea was introduced, to an extent, in [AS12].

HCSP extends on these notions and overcomes the inherent limitations of lazy clause generation and EFC’s g-nogoods. It uses many valued SAT [BHM00b] to efficiently represent big domains and to learn clauses that affect them. Unlike the competing techniques it generates explanations during conflict analysis and not during propagation. This lets HCSP take less memory and to learn not only clauses and g-nogoods but also complicated constraints. The description of the HCSP conflict analysis is given in the next chapter.

1.4 Research contribution

The thesis is structured as a collection of (extended-versions of) an article [VS10a] published in AAAI’10, an article currently under submission [VS14], and a proof supplement that was published as a technical report [VS10b]. Chapter 4 includes additional material that is not included in these publications. The main contributions of this research are summarized in the following list:

- **A fast CSP solver.** HCSP is a new solver written as part of this research. It is the fastest\(^8\) solver running the benchmarks of the 2009 CSP competition [DLR09], the last one held using the original format of this solver. Transition to the CSP input language of MiniZinc also shows promising results when comparing them to other solvers supporting the MiniZinc language.

- **Lazy explanation generation.** In Chap. 2 we explain how HCSP refines the basic concept of EFC [BK, KB05] and Lazy Clause Generation [OSC09], i.e., explaining a propagation via a clause, and performs it only during conflict analysis. The lazy generation has a lower memory cost. Lazy clause generation makes it possible to choose the best possible explanation for a given propagation, when there is more than one possibility. Moreover, it allows to generate an augmented explanation, as explained in Subsection B.4.2. An augmented explanations does not explain the propagation, but rather explains why propagating a constraint leads to a conflict.

- **Combine.** Algorithm 3.1 in Subsection 3.2.2 introduces the notion of constraint combination which generalizes binary resolution of many-valued SAT (which is a generalization of binary resolution of Boolean SAT). Introduced in Subsection 3.4.1

\(^8\)HCSP is the fastest if benchmarks with table constraints are excluded. The current implementation of table constraints is inefficient and has to be rewritten.
the combine rule for \(c_1\) and \(c_2\) and pivot \(x\) is defined as:

\[
\begin{align*}
\frac{c_1(x, \bar{y}) \quad c_2(x, \bar{y})}{x \not\in \mathcal{X} \lor \exists x' \in \mathcal{X}.\left[c_1(x', \bar{y}) \land c_2(x', \bar{y})\right]} \quad \text{(combine}(x)\quad).
\end{align*}
\]

Where \(\bar{y}\) is a collection of variables which are constrained by either \(c_1\), \(c_2\), or both and \(D_y\) is their domain. Also:

\[
\mathcal{X} = \{x' \in \mathbb{Z} | \forall \bar{y}' \in D_y.\left[\neg c_1(x', \bar{y}') \lor \neg c_2(x', \bar{y}')\right]\}.
\]

The research shows that combine possess properties which make it work well in conflict analysis. Also, several strong rules are derived using the general combine rule.

- **Non-clausal conflict analysis.** Using the combine rule HCSP employs a novel conflict-analysis algorithm which learns complicated constraints. It is able, for example, to combine \(x_1 + 2x_2 + 7x_3 \geq 0\) with \(x_1 + 3x_2 \geq 5\) and get \(2x_1 + 5x_2 + 7x_3 \geq 5\), and then to combine the result with \(2x_1 + 5x_2 + 7x_3 \leq 4\) and detect that these constraints contradict with no need for further backtracking. Other conflict analysis methods such as [BK, KB05, VS10a] require conflicts of explanation clauses to reach the same conclusion.

By having specialized combination rules only for some pairs of constraints, HCSP slightly resembles SMT.\(^9\) The SMT solver groups constraints into separate theories, processed independently. Much of the inference between constraints is performed independently in each theory, similarly to the combination rules that work only on specific pairs of constraints. But unlike HCSP, SMT inference is mostly visible by asserting existing constraints through conflict analysis and theory propagation, and not by introducing new constraints during conflict analysis.

Other major differences between SMT and HCSP are the decision procedure and propagation. In SMT decisions are made on Boolean variables that encode more complicated constraints: should an atom, i.e., constraint, be satisfied or falsified in order to satisfy the overall formula. In HCSP the decisions are on all variables. In SMT, theory propagation may learn a new Boolean constraint that affects existing constraints (atoms). In HCSP, propagation reduces the domains of variables and only conflict analysis may introduce new constraints, including Boolean constraints that affect exiting constraints.

\(^9\)SMT and CSP traditionally do not focus on the same set of problems, although there are some problems that are dealt with by both. Whereas CSP is typically focused on finite discrete domains, SMT can accommodate any decidable theory, including those over variables of type Real.
• **Proofs.** Section 2.4 describes how HCSP generates machine-checkable proofs for unsatisfiability using basic inference rules. It uses *explanation* rules to generate signed-clauses out of constraints, and a binary signed-resolution to create new clauses. HCSP is the only CSP solver known to us that generates proofs, a feature that was so far only available in SAT solvers.

• **Interpolants.** HCSP can generate interpolants, and is the only CSP solver known to us that has this capability. In Section 4.5 we prove the correctness of Algorithm 4.4, which finds Craig’s Interpolant for a CSP. The algorithm is based on a similar algorithm that is used by some SAT solvers for the case of propositional formulas and binary resolution.
Chapter 2

A Proof-Producing CSP Solver

Abstract

HCSP 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. HCSP 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.

2.1 Introduction

Many problems in planning, scheduling, automatic test-generation, configuration and more, can be naturally modeled as Constraint Satisfaction Problems (CSP) [Dec03], 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.

1Published in AAAI-10
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 [HJMM04].

Unlike SAT solvers, CSP solvers do not have the luxury of handling clausal constraints. They need to handle constraints such as $a < b + 5$, $\text{allDifferent}(x, y, z)$, $a \neq b$, 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 $(x_1 \in \{1, 2\} \lor x_2 \notin \{3\})$ constrains $x_1$ 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, a problem which attracted extensive theoretical research and development of tools [LKM03, BHM00b].

In this article we describe how our arc-consistency-based CSP solver HCSP (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 HCSP reconstructs them, by using information that it saves during the learning process. We will describe this conflict-analysis mechanism in detail in Section 2.3 and 2.4, and compare it to alternatives such as 1-UIP [ZMMM01], MVS [LKM03] and EFC [KB05].

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

3 Signed SAT is also called MV-SAT (i.e. Many Valued SAT).
in Section 2.5. We begin, however, by describing several preliminaries such as CSP and signed SAT, and by introducing our running example.

2.2 Preliminaries

2.2.1 The Constraint Satisfaction Problem (CSP)

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

We will use the example below as our running example.

Example 2.2.1 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 $x_1 \lor x_2$, $x_1 \lor x_3$, $(x_2 \land x_3) \rightarrow a = 1$. 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 $\mathrm{NoOverlap}(a, L_a, b, L_b)$ to denote the constraint $a + L_a \leq b \lor b + L_b \leq a$. Overall, then, the formal definition of the CSP is:

\[
V = \{a, b, c, x_1, x_2, x_3\}; \\
D = \begin{cases} 
D_a = D_b = D_c = [1, 8], \\
D_{x_1} = D_{x_2} = D_{x_3} = \{0, 1\}; \\
\end{cases} \\
C = \begin{cases} 
c_1 : \mathrm{NoOverlap}(a, 4, b, 4) \\
c_2 : \mathrm{NoOverlap}(a, 4, c, 4) \\
c_3 : \mathrm{NoOverlap}(b, 4, c, 4) \\
c_4 : x_1 \lor x_2 \\
c_5 : x_1 \lor x_3 \\
c_6 : (x_2 \land x_3) \rightarrow a = 1. 
\end{cases}
\]

2.2.2 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., \( a \not\in [1, 2] \) 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 \( b \not\in \{3\} \). 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 HCSP 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 2.3.

**Resolution rules for signed clauses**

HCSP uses Beckert et al.’s generalization of binary resolution to signed clauses in order to generate resolution-style proofs [BHM00a]. 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.

\[
\text{signed binary resolution} \\
\frac{(v \in A \lor X) \quad (v \in B \lor Y)}{(v \in (A \cap B) \lor X \lor Y)} [R_v] \\
\text{simplification} \\
\frac{(v \in \emptyset \lor Z)}{(Z)} [S_v]
\]

\[
\text{join literals} \\
\frac{((\bigvee_i v \in A_i) \lor Z)}{(v \in \bigcup_i A_i \lor Z)} [J_v]
\]

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

\[
\frac{(a \in \{1, 2\} \lor b \in \{1, 2\})}{(a \in \emptyset \lor b \in \{1, 2\} \lor b = 3)} [R_a] \\
\frac{(a \in \{1, 2\} \lor b = 3)}{(a = 3 \lor b = 3)} [R_a] \\
\frac{(a \in \emptyset \lor b \in \{1, 2\} \lor b = 3)}{(b \in \{1, 2\} \lor b = 3)} [S_a] \\
\frac{(b \in \{1, 2\} \lor b = 3)}{(b \in \{1, 2, 3\})} [J_a] \\
gives a different result if the pivot is \( b \) instead of \( a \):
In practice HCSP 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.

2.3 Learning

In this section we explain the learning mechanism in HCSP, 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 2.3.2 we will show the conflict analysis algorithm.

2.3.1 Implication graphs and conflict clauses

A propagation process is commonly described with an implication graph. Figure 2.1 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 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
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’) [ZMMM01]. The dashed line marked as 1-UIP in Figure 2.1 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 2.1 shows that \((a = 1 \land b \in [1, 8] \land c \in [1, 8])\) imply a conflict. To avoid the conflict the solver can generate a conflict clause that forbids this combination, namely \((a \neq 1 \lor b \notin [1, 8] \lor c \notin [1, 8])\) in this case. HCSP 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.

### 2.3.2 Conflict analysis and learning

Algorithm 2.1 describes the conflict analysis function in HCSP, which is inspired by the corresponding function in SAT [ZM03]. 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 \( \text{lit}(u) \) denote the literal associated with \( u \). For a literal \( l \), let \( \text{var}(l) \) denote the variable corresponding to \( l \). For a set of nodes \( U \), let \( \text{vars}(U) = \{ \text{var}(\text{lit}(u)) \mid u \in U \} \), and for a clause\(^4\) \( cl \), let \( \text{vars}(cl) = \{ \text{var}(l) \mid l \in cl \} \).

\(^4\)Here we use the standard convention by which a clause can also be seen as a set of literals.
Constraint | Explanation clause
--- | ---
$c_1$ NoOverlap(a,4,b,4) | $\omega_1 = (a \not\in [1,4] \lor b \not\in [1,4])$
$c_2$ NoOverlap(a,4,c,4) | $\omega_2 = (a \not\in [1,4] \lor c \not\in [1,4])$
$c_3$ NoOverlap(b,4,c,4) | $\omega_3 = (b \not\in [5,8] \lor c \not\in [5,8])$
$c_4$ $x_1 \lor x_2$ | $\omega_4 = (x_1 \neq 0 \lor x_2 \neq 0)$
$c_5$ $x_1 \lor x_3$ | $\omega_5 = (x_1 \neq 0 \lor x_3 \neq 0)$
$c_6$ $(x_2 \land x_3) \rightarrow a = 1$ | $\omega_6 = (x_2 \neq 1 \lor x_3 \neq 1 \lor a = 1)$

Table 2.1: Constraints and explanation clauses for the running example. The explanation clauses refer to the inferences depicted in the implication graph in Figure 2.1.

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

**Definition 2.3.1 (Explanation clause)** Let $u$ be a node in the implication graph such that $\text{lit}(u) = l$. Let $(l_1, l) \ldots (l_n, l)$ 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. $(l_1 \land \cdots \land l_n \land c) \rightarrow l$.

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 2.3.1** Explanation clauses for our running example appear in the third column in Table 2.1. These clauses are built with respect to the nodes in the implication graph in Figure 2.1. We will explain how they are generated in Section 2.3.3.

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 Relevant (⟨nodes⟩, ⟨clause⟩) that is invoked in line 4 returns a subset $N$ of the nodes in ⟨nodes⟩ that are relevant for the clause ⟨clause⟩, i.e., $\text{vars}(N) = \text{vars}(⟨\text{clause}⟩)$.

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-node)) is the resolution variable (the resolution process in line 9 is as was explained in Section 2.2.2), 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 ⟨nodes⟩. 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 ∈ vars(⟨nodes⟩), let U(v) be the maximal subset of nodes in ⟨nodes⟩ that are labeled with v, i.e., for each u ∈ 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 2.1 are proved in [VS10b].

Algorithm 2.1 Conflict analysis

1: function CSP-Analyze-Conflict
2:   cl := EXPLAIN (conflict-node);
3:   pred := PREDECESSORS (conflict-node);
4:   front := RELEVANT (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 (RELEVANT (front ∪ pred, cl));
12:   end while
13: add-clause-to-database(cl);
14: return clause-asserting-level(cl);
15: end function

Example 2.3.2 Table 2.2 demonstrates a run of Algorithm 2.1 on our running example. Observe that it computes the conflict clause by resolving ω₃ with ω₂, and the result of this resolution with ω₁. The intermediate result, namely the result of the first of these
resolutions, is discarded. The resulting conflict clause \((a \not\in [1, 4] \lor b \not\in [1, 8] \lor c \not\in [1, 8])\) is stronger than what the clause would be had we used the 1-UIP method, namely \((a \neq 1 \lor b \not\in [1, 8] \lor c \not\in [1, 8])\).

\[
cl := (b \not\in [5, 8] \lor c \not\in [5, 8]) \quad (= \omega_3)
\]

\[
\text{pred} := \{b \in [5, 8]@1, c \in [5, 8]@1\}
\]

\[
\text{front} := \{b \in [5, 8]@1, c \in [5, 8]@1\}
\]

\[
curr-node := c \in [5, 8]@1
\]

\[
\text{front} := \{b \in [5, 8]@1\}
\]

\[
\text{expl} := (a \not\in [1, 4] \lor c \not\in [1, 4]) \quad (= \omega_2)
\]

\[
cl := (a \not\in [1, 4] \lor b \not\in [5, 8] \lor c \not\in [1, 8])
\]

\[
\text{pred} := \{a = 1@1, c \in [1, 8]@0\}
\]

\[
\text{front} := \{a = 1@1, b \in [5, 8]@1, c \in [1, 8]@0\}
\]

\[
curr-node := b \in [5, 8]@1
\]

\[
\text{front} := \{a = 1@1, c \in [1, 8]@0\}
\]

\[
\text{expl} := (a \not\in [1, 4] \lor b \not\in [1, 4]) \quad (= \omega_1)
\]

\[
cl := (a \not\in [1, 4] \lor b \not\in [1, 8] \lor c \not\in [1, 8])
\]

\[
\text{pred} := \{a = 1@1, b \in [1, 8]@0\}
\]

\[
\text{front} := \{a = 1@1, b \in [1, 8]@0, c \in [1, 8]@0\}
\]

\[
\text{add}((a \not\in [1, 4] \lor b \not\in [1, 8] \lor c \not\in [1, 8]))
\]

\[
\text{return 0}
\]

Table 2.2: A trace of Algorithm 2.1 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 HCSP 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 \(\langle \text{Constraint}, \text{Rule}, \text{Pivot} \rangle\), where \text{Constraint} is a pointer to a constraint in the constraints database, \text{Rule} is the parameterized inference rule by which an explanation clause can be inferred (if \text{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.

2.3.3 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 \( \{ x \mid x \in D(a) \land x \leq \max(D(b)) \} \), 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 \( \text{LE}(m) \) below with \( m = \max(D(b)) \).

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

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 \( \text{LE}(6) \) implies the same literal at the current state, which means that it is an explanation clause. Table 2.3 contains several such inference rules that we implemented in HCSP. In a proof supplement of this article [VS10b] 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 HCSP, 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 therefore potentially needed for the proof.

2.4 Deriving a proof of unsatisfiability

If the formula is unsatisfiable, HCSP 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.
Table 2.3: Inference rules for some popular constraints, which HCSP uses for generating explanation clauses. The last rule is a bound consistency propagation targeted at \( a \).

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

Algorithm 2.2 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 \( \text{ApplyRule} \). This function returns the explanation clause corresponding to the constraint, based on the rule \( s_i.\text{Rule} \). After obtaining a clause, in lines 18–20 it resolves it with \( cl \), the clause from the previous iteration, and prints this resolution step. Note that the clauses resolved in line 19 can be intermediate clauses that were not made into conflict clauses by the conflict analysis process.

Hence, Algorithm 2.2 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 2.4.1 First, we need an inference rule for NoOverlap:
Algorithm 2.2 Printing the proof

1: function PRINTPROOF(conflict-clause)
2:   Printed ← Printed ∪ conflict-clause
3:   $(s_1, \ldots, s_k) \leftarrow \text{PROOFSTEPS} (\text{conflict-clause})$
4:   for $i \leftarrow 1, k$ do
5:     if $s_i.C$ is a clause and $s_i.C \notin \text{Printed}$ then
6:       PRINTPROOF $(s_i.C)$
7:     end if
8:   end for
9:   for $i \leftarrow 1, k$ do
10:      if $s_i.C$ is a clause then $expl \leftarrow s_i.C$
11:         else
12:            $expl \leftarrow \text{APPLYRULE} (s_i.C, s_i.Rule)$
13:            Print(“Rule:”, $s_i.Rule$)
14:            Print(“Premise:”, $s_i.C$, “Consequent:”, $expl$)
15:       end if
16:      if $i = 1$ then $cl \leftarrow expl$
17:         else
18:            Print(“Resolve”, $cl$, $expl$, “on”, $s_i.Pivot$)
19:            $cl \leftarrow \text{Resolve}(cl, expl, s_i.Pivot)$
20:            Print(“Consequent:”, $cl$)
21:       end if
22:   end for
23: end function
1. NoOverlap(b, 4, c, 4) premise
2. \((b \notin [5, 8] \vee c \notin [5, 8])\) 1[NO(5,5)]
3. NoOverlap(a, 4, c, 4) premise
4. \((a \notin [1, 4] \vee c \notin [1, 4])\) 3[NO(1,1)]
5. NoOverlap(a, 4, c, 4) premise
6. \((a \notin [1, 4] \vee b \notin [5, 8] \vee c \notin [1, 8])\) 2,4[Resolve(c)]
7. NoOverlap(a, 4, b, 4) premise
8. \((a \notin [1, 4] \vee b \notin [1, 8] \vee c \notin [1, 8])\) 6[NO(1,1)]
9. \((a \notin [5, 8] \vee c \notin [5, 8])\) 5,7[Resolve(b)]
10. \((a \notin [1, 8] \vee b \notin [1, 8] \vee c \notin [1, 8])\) 8,9[Resolve(a)]
11. (c \in [1, 8]) premise
12. \((a \notin [1, 8] \vee b \notin [1, 8])\) 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 2.4: A deductive proof of the unsatisfiability of the CSP.

\[
\text{NoOverlap}(a, l_a, b, l_b) \quad \frac{(a \notin [m, n + l_b - 1] \vee b \notin [n, m + l_a - 1])}{(\text{NO}(m,n))},
\]
where \(m, n\) are values such that \(1 - l_b \leq n - m \leq l_a - 1\).

Table 2.4 shows a proof of unsatisfiability of this CSP. This presentation is a beautification of the output of Algorithm 2.2. 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 HCSP found an unsatisfiable core which does not refer to constraints over these variables.

### 2.5 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 2.3.2 we saw that 1-UIP results in the weaker conflict clause \((a \neq 1 \vee b \notin [1, 8] \vee c \notin [1, 8])\). After learning this clause the solver backtracks 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 2.1 produces the clause \((a \notin [1, 4] \vee b \notin [1, 8] \vee c \notin [1, 8])\) (see line 13 in Table 2.2). 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 [LKM03].

Another alternative is the MVS algorithm, which was described in [LKM03] in terms of traversing the assignment stack rather than the implication graph. MVS essentially produces the same conflict clause as Algorithm 2.1, but it assumes that the input formula consists of signed clauses only, and hence does not need explanation clauses. We find Algorithm 2.1 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 [VS10b] 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 [KB05]. There are two main differences between the learning mechanisms:

- EFC generates a separate explanation of each removed value. HCSP generates an explanation for each propagation, and hence can remove sets of values. This affects not only performance: HCSP’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 HCSP generates an explanation only in response to a conflict, and hence only for constraints that are relevant for generating the conflict clause.

Performance

HCSP performs reasonably well in comparison with state of the art solvers. In the CSC09 competition [DLR09], in the $n$-ary constraints categories, an early version of HCSP 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 HCSP.
Chapter 3
Learning general constraints in CSP
(long version)

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 clear advantage over the state-of-the-art solver Mistral in most types of constraints, averaging 25% reduction in fails (time-out or memory-out), 29% reduction in run-time of instances that both engines solved, and 95.9% average reduction in the number of backtracks, when measured on the last CSP competition (2009) benchmarks that include inequality constraints (a total of 2162 benchmarks).

3.1 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., [KB05]). 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 [Dec90]. Nogoods are defined as partial assignments
that cannot be extended to a full solution. Later generalized nogoods [KB05] (g-nogoods for short) were proposed, which allow non-assignments as well, e.g., a g-nogood \((x \leftarrow 1, y \leftarrow 1)\) 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 [KB05].

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 \(v \not\in D\) (called positive and negative signed literals, respectively), where \(v\) is a variable and \(D\) is a domain of values. Beckert et al. [BHM00b] 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:

\[
\frac{(v \in A \lor X) \ (v \in B \lor Y)}{(v \in (A \cap B) \lor X \lor Y) \ [\text{Signed Resolution}(v)]}
\]  

(3.1)

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 \((X \lor Y)\).

As we showed in an earlier publication [VS10a], we used this rule in our CSP solver HCSP [VS10a] (short for HaifaCSP), 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 for propagations by non-clausal constraints, and rule (3.1) 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 [KB05], 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 [GMM10].

In this article we study a different learning scheme, which is based on inference rules with non-clausal consequents. 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 $c_1 = y_1 \leq x$ propagates $x \in \{10, 14, \ldots, 30\}$, which leads to a contradiction with a constraint $c_2 = x \leq y_2$. During conflict-analysis, HCSP now infers from this propagation the constraint

$$x \in [8, 9] \lor [y_1, y_2] \not\subseteq [8, 9],$$

based on Combine (square brackets denote a range). This constraint is both implied by $c_1, c_2$, and has the form of a disjunction of two constraints, each of which has less variables than the set of input variables. The latter property 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 offer simpler alternatives.

Our experimental results prove that indeed the new scheme is better than clausal explanation. For reference, we also compared HCSP to the state-of-the-art CSP solver Mistral [Heb08]. Our experiments with thousands of benchmarks from the latest competition (in 2009), as we describe in Sec. 3.5, show that comparing to Mistral, HCSP achieves an overall 25% reduction in fails (time-out or memory-out), 29% reduction in run-time of instances that both engines solved, and 95.9% average reduction in the number of backtracks. Perhaps this drastic reduction in backtracks indicates that the cost of learning strong constraints is mitigated by a better search (Mistral itself does not learn constraints).

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 [VS10a]. Sections 3.3 and 3.4 describe the new set of inference rules, the requirements from them and the proofs that they fulfill these requirements. In Sec. 3.3 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. 3.5 with empirical evaluation and some proposals for future research.
3.2 Background

3.2.1 Essentials of HCSP

The engine of HCSP adopts classical ideas from the CSP and SAT literature. We assume the reader is somewhat familiar with those, and only mention several highlights briefly.

HCSP 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 [MMZ01].\(^2\) 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 is SAT [Sht00]. It includes restarts, learning (to be described), and deletion of learnt-constraints with low activity.

HCSP supports all the constraint types used in the 2009 CSP competition. It has precise propagators for the following types of constraints (where \(x_i\) denote variables, \(b_i\) Boolean variables, \(a_i\) constants, and \(\diamond \in \{=, \leq, \geq\}\)):

- \(x_0 = x_1, x_0 = -x_1, x_0 \diamond \text{abs}(x_1), x_0 \diamond x_1 + x_2 + \ldots\) (with wrap on overflow),
- \(x_0 \leq x_1 \times x_2, x_0 \geq x_1 \times x_2, x_0 = \min(x_1, x_2, \ldots), x_0 = \max(x_1, x_2, \ldots), x_0 = (b_0?x_2 : x_3), x \neq 0 \leftrightarrow y = z, x_0 \in \text{Set} \leftrightarrow x_1 < x_2, x_0 - x_1 \geq a_0, (x_0 + a_0 \leq x_1 \lor x_1 + a_1 \geq x_0), x_0 \neq x_1, \text{AllDifferent}(x_0, x_1, \ldots), [x_0, x_1 + a_0] \subseteq [a_1, a_2], [x_0, x_1 + a_0] \not\subseteq [a_1, a_2], a_0 * x_0 + a_1 * x_1 + \ldots \geq a_k, \text{signed-clauses}, \text{and a disjunction of any of the above when there are no shared variables. It has imprecise propagators for}
- \(x_0 = x_1 * x_2, x_0 = x_1 / x_2, x_0 = x_1 \% x_2, x_0 = \text{pow}(x_1, x_2), \text{and table constraints.}

Complex constraints modeled by a language such as XCSP [C0909] are rewritten into more basic ones.

The rest of this section is focused on the learning mechanism.

3.2.2 Conflict analysis

An implication graph \(G(N, E)\) is a directed acyclic graph in which each node \(n \in N\) represents a literal (a variable domain) and each edge \(c \in E\) represents a constraint. Incoming edges to a node \(n\) can only be labeled with the same constraint. Let \((n_1, n), \ldots, (n_k, n)\) be the incoming edges of \(n\), all of which are labeled with a constraint \(c\). This represents the fact that starting with domains \(n_1, \ldots, n_k\) the propagator of \(c\) inferred the domain in \(n\). The constraint \(c\) is called the antecedent of \(n\). Each node is also associated with the decision level in which the domain reduction occurred.

When an implication graph ends with a conflict (a node labeled with \(\bot\)), it is called a conflict graph. We will follow a convention by which this graph is depicted with the roots at the left and the sink at the right, and the horizontal position of a node indicates

\(^2\)This can be seen as a variant of the dom/wdeg strategy [BHLS04].
Figure 3.1: Part of a conflict graph, based on the constraints $c_1 \vdash y \geq x$, $c_2 \vdash x \geq y$, and $c_3 \vdash x > y + u$. Empty circles represent nodes in the set $F$. Drawing (a) is before relaxation, and drawing (b) is right after it. Relaxation discovers that the domain reduction by $c_2$ is not necessary for conflicting the constraint $curr$ ($c_3$ in this case).

HCSP analyzes the conflict graph in order to learn a new constraint, called accordingly a conflict constraint (or a conflict clause in SAT). A conflict constraint is called asserting if there exists a backtrack level in which this constraint necessarily leads to additional propagation. The conflict-analysis function, `ANALYZECONFLICT`, indeed computes this level and returns it to the solver, which backtracks accordingly. Alg. 3.1 shows pseudo-code of `ANALYZECONFLICT` 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 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 3.2.1** Consider the constraints

$$c_1 \vdash y \geq x \quad c_2 \vdash x \geq y \quad c_3 \vdash x > y + u$$

and the conflict graph in Fig. 3.1(a). In Alg. 3.1, 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. 3.1(b) 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.

Relaxation is necessary for several reasons:
• Preventing a situation in which the learned clause is still conflicting immediately after backtracking, instead of being asserting,

• In Sec. 3.4.3 we rely on relaxation in the development of some of the rules.

• Our experiments show that without it many more cases fall back to clausal explanations, because relaxation enables to circumvent them.

Relaxation is a contribution of the current article, although not its focus.

Let us return to the description of Alg. 3.1. In lines 5–9 \textsc{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:

\textit{Invar1.} $curr$ contradicts the domains defined by $F$, and is able to detect it via propagation (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).

\textit{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. \textsc{Combine} and \textsc{GetNewSet} are targeted towards maintaining it as will be evident later. The traversal stops in line 5 once the function \textsc{stop} detects that $Cons$ is asserting, or that it conflicts the domains at decision level 0. In the latter case the function \textsc{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 \textsc{combine} is the main contribution of this article and will be discussed in length in later sections.

Let us now shift our focus to \textsc{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 18 the function \textsc{distinct} leaves only the right-most one. The reason that there may be multiple entries of a variable in $F$ 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.

3.2.3 Clausal Explanations

Generic explanations were used in the past (e.g., [KB05, GMM10]) for learning of g nogoods. The scheme we describe here uses inference rules specialized for each constraint
Algorithm 3.1 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. Combine, the subject of Sect. 3.3–3.4, infers a new constraint. GetNewSet computes the new set of nodes $F$, as explained in the text.

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

16: function GetNewSet (node-set $F$, node pivot)
17: \hspace{1em} $F \leftarrow (F \setminus \{\text{pivot}\}) \cup$ parents of pivot;
18: \hspace{1em} $F \leftarrow$ Distinct ($F$); \text{▷ Chooses right-most node of each variable in $F$}
19: \hspace{1em} Return $F$;
20: \hspace{1em} end function
type, resulting in signed clauses. Such clausal explanations are important in our context both for understanding the alternative mechanism that we used in [VS10a] (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 that we do not directly support in COMBINE. This technique is also based on ANALYZECONFLICT, with a difference only in the implementation of COMBINE.

Let us begin by formally defining the notion of explanation.

Definition 3.2.1 (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., $(l_1 \land \ldots \land l_n \land c) \rightarrow l$. Then a clause $e$ is an explanation of this propagation if the following two conditions hold:

\[ c \rightarrow e \]  
\[ (l_1 \land \ldots \land l_n \land e) \rightarrow l. \]  

Eq. (3.2) guarantees that the new clause $e$ is logically implied by an existing constraint, hence we do not lose soundness. Eq. (3.3) 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 [VS10a].

Example 3.2.2 The following rule from [VS10a] provides a clausal explanation for an inequality constraint:

\[ \frac{x \leq y}{x \in (\neg \infty, m] \lor y \in [m+1, \infty)} \text{ (LE(m))} \]  

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:

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

and the constraint

\[ c \equiv x \leq y, \]

which implies in the context of $l_1, l_2$ the literal

\[ l \equiv x \in [1, 2]. \]

Using (3.4) with $m = \max(y) = 2$ we obtain the explanation

\[ e \equiv (x \in (-\infty, 2] \lor y \in [3, \infty)). \]

and indeed (3.2) and (3.3) hold, since $c \rightarrow e$ and $(l_1 \land l_2 \land e) \rightarrow l$. In [VS10a] alternatives to choosing $m = \max(y)$ are discussed. \[ \square \]
In [VS10a] we showed how HCSP generates a signed conflict clause with an inference system based on signed resolution (3.1), 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 (3.1) is used for resolving signed clauses.

Example 3.2.3 The following demonstrates conflict analysis with clausal explanations. In addition to (3.4), we will use a variant of this rule for strict inequality:

\[
\frac{x < y}{x \in (-\infty, m - 1) \lor y \in [m + 1, \infty)} \quad (L(m))
\]

We will also use the observation that if \( c \rightarrow e \), then \((l \lor c) \rightarrow (l \lor e)\), to handle constraints with disjunctions. Let \( D_x = \{0, 1\}, D_y = \{0, 1\}, D_z = \{0..100\} \), and

\[
c_1 = (z = 9 \lor x < y) \quad c_2 = (z = 10 \lor x \geq y).
\]

The conflict graph on the right 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 \( \text{pivot} = \{y \in \{1\}\} \). At this point \( c_2 \) generates the explanation

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

based on \( LE(0) \) (Eq. (3.4)), and \( c_1 \) generates the explanation

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

based on \( L(0) \) (Eq. (3.5)). Resolving the two explanations on \( y \) yields

\[
(z \in \{9, 10\} \lor x \in \{0\}).
\]

Now \( \text{pivot} = x \in \{0\} \). \( c_1 \) explains the propagation of \( x \) with the clause \((z \in \{9\} \lor y \in [2, \infty) \lor x \in (-\infty, 0])\), based on \( L(1) \). Resolving it with (3.6) on \( x \) yields

\[
(z \in \{9, 10\} \lor x \in (\infty, -1] \lor y \in [2, \infty))\).
\]

Now \( F \) is equal to the three nodes on the left. (3.7) is now asserting, since e.g., at the previous decision level \( z \in \{9, 10\} \) is implied.
3.3 Non-clausal inference: requirements

In Alg. 3.1 COMBINE is given the constraints $Cons(x, \bar{y})$ and $antecedent(x, \bar{y})$ with a joint variable $x$ that appears at the node pivot, and some set of variables $\bar{y}$, which may or may not be common to both. It outputs a new constraint over $x, \bar{y}$ that is assigned back into $Cons$. In the presentation that follows we will use $c_1(x, \bar{y})$ to denote $Cons(x, \bar{y})$, $c_2(x, \bar{y})$ to denote $antecedent(x, \bar{y})$, and $c^*(x, \bar{y})$ to denote the output constraint. We also define $c_{12}(x, \bar{y}) = c_1(x, \bar{y}) \land c_2(x, \bar{y})$.

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:

$$c_{12} \rightarrow c^*. \quad (3.8)$$

This guarantees that the constraint eventually learned in line 13 is inferred via sound derivations, and hence is guaranteed to be implied by the original CSP.

Let $D'_x, D'_y$ denote the domains of $x, \bar{y}$ 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 $\phi$, the set of literals $\psi$ is derivable through constraint propagation. Then to preserve Invar1 (see Sec. 3.2), our second requirement from $c^*$ is:

$$c^*, D'_x, D'_y \vdash_{cp} \bot. \quad (3.9)$$

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

3.4 Non-clausal inference: rules and their proofs

Rules R1–R6 in Table 3.1 are triples $\langle c_1, c_2, c^* \rangle$ that satisfy the two requirements (3.8) and (3.9). Rules R7 and R8 satisfy (3.8) but not necessarily (3.9). We use them to infer constraints, and then test whether they happen to satisfy (3.9). In addition, we use the following meta-rule for handling disjunctions:

$$
\frac{(A \lor c_1) \land (B \lor c_2)}{A \lor B \lor c^*} \quad (3.10)
$$

If $\langle c_1, c_2, c^* \rangle$ satisfies (3.8) and (3.9), then so does (3.10).

\footnote{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.}
Proof

\[(A \lor c_1) \land (B \lor c_2)\]

\[= (A \land B \lor A \land c_2 \lor c_1 \land B \lor c_1 \land c_2)\]

Clearly

\[A \land B \lor A \land c_2 \rightarrow A\]

and

\[c_1 \land B \rightarrow B\]

and

\[c_1 \land c_2 \rightarrow c^*\]

Hence

\[(A \land B \lor A \land c_2) \lor (c_1 \land B) \lor (c_1 \land c_2) \rightarrow A \lor B \lor c^*\]

\[\square \square\]

Example 3.4.1 We now show two examples in which the rules lead to stronger learning than explanation-based learning

- Recall example 3.2.3, which yielded the conflict clause (3.7). Given the same conflict graph but using the meta rule (3.10) with pivot \(y\), we learn instead \(z \in \{9, 10\}\), which is clearly stronger.

- 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

  \[c_1 \equiv (z \in \{1\} \lor y_1 \leq x) \quad c_2 \equiv (z \in \{1\} \lor x \leq y_2) .\]

  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 \(k_1 = k_2 = 0\), and the meta rule (3.10) we obtain:

  \[(z \in \{1\} \lor x \in [8, 9] \lor [y_1, y_2] \not\subseteq [8, 9]) .\]

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

\[(z \in \{1\} \lor y_1 \in (-\infty, 7] \lor x \in (8, \infty)) ,\]

and resolving these explanations on the pivot \(x\) yields

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

This constraint is strictly weaker than (3.11) because the right disjunct of (3.11) implies \(y_1 \leq y_2\).
Most of the entries in the table were developed by instantiating a general inference rule called Combine (see below), which satisfies these requirements. In some other cases instantiating it turned out to be too complicated and we found $c^*$ without it. Sec. 3.4.3 includes proofs for some of these other rules.

<table>
<thead>
<tr>
<th></th>
<th>$c_1$</th>
<th>$c_2$</th>
<th>$c^*$</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>$x \in X_1 \lor A_1(y)$</td>
<td>$x \in X_2 \lor A_2(y)$</td>
<td>$x \in (X_1 \cap X_2) \lor A_1(y) \lor A_2(y)$</td>
</tr>
<tr>
<td>R2</td>
<td>$y_1 \leq x - k_1$</td>
<td>$x \leq y_2 - k_2$</td>
<td>$(x \in [k_1 + \min(D'<em>{y_1}), \max(D'</em>{y_2}) - k_1]) \lor ([y_1, y_2 - k_2 - k_1] \subseteq [\min(D'<em>{y_1}), \max(D'</em>{y_2}) - k_2 - k_1])$</td>
</tr>
<tr>
<td>R3</td>
<td>$y_1 \leq x$</td>
<td>$[x, y_2] \not\subseteq [a, b]$</td>
<td>$(a &gt; x \geq \min(D'<em>{y_1})) \lor ([y_1, y_2] \not\subseteq [\min(D'</em>{y_1}), b])$</td>
</tr>
<tr>
<td>R4</td>
<td>$x \leq y_1 - k_1$</td>
<td>$[y_2, x - k_2] \not\subseteq [a, b]$</td>
<td>$(\max(D'<em>{y_1}) - k_1 \geq x &gt; b + k_2) \lor ([y_2, y_1 - k_1 - k_2] \not\subseteq [a, \max(b, \max(D'</em>{y_1} - k_1 - k_2))])$</td>
</tr>
<tr>
<td>R5</td>
<td>$[y_1, x] \not\subseteq [a_1, b_1]$</td>
<td>$[x, y_2] \not\subseteq [a_2, b_2]$</td>
<td>$(a_2 &gt; x &gt; b_1) \lor ([y_1, y_2] \not\subseteq [a_1, b_2])$</td>
</tr>
<tr>
<td>R6</td>
<td>$[x, y] \not\subseteq [a_1, b_1]$</td>
<td>$[y, x] \not\subseteq [a_2, b_2]$</td>
<td>$(x \in (D'_y - ([a_1, b_1] \cup [a_2, b_2])) \lor y \not\subseteq (D'_y - ([a_1, b_1] \cup [a_2, b_2])))$</td>
</tr>
</tbody>
</table>
| R7 | $y \leq x + k_1$ | $x \leq y + k_2$ | $\begin{cases} -k_1 \leq x - y \leq k_2 \text{ if } k_1 + k_2 \geq 0 \\
\bot \text{ otherwise} \end{cases}$ |
| R8 | $ax + \sum_{i=1}^n a_i y_i \geq k_1$ | $-ax + \sum_{i=1}^n b_i y_i \geq k_2$ | $\sum_{i=1}^n (a_i + b_i) x_i \geq k_1 + k_2$ |

Table 3.1: Triples $(c_1, c_2, c^*)$ that we use for deriving conflict constraints. The top part include rules that satisfy both (3.8) and (3.9), whereas the others are only guaranteed to satisfy (3.8). When using them we test if they satisfy (3.9). Combinations of a pair of linear constraints can be brought to the form expected by R8 if the coefficients of $x$ have opposite signs, via multiplication by a positive constant. Note that the coefficients $a_i, b_i$ may be 0 in R8.

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 R7, R8 in the table), then COMBINE uses explanation-based inference (see Sec. 3.2.3) as a fallback solution. Pseudocode of COMBINE, which is rather self-explanatory, appears in Alg. 3.2.
Algorithm 3.2 Combine infers a new constraint $c^*$ from $c_1, c_2$, which satisfies (3.8) and (3.9), the requirements listed in Sec. 3.3.

1.-1: function Combine(constraint $c_1$, constraint $c_2$, node pivot, node-set $F$)
2.-1: $F' = \text{GetNewSet}(F, \text{pivot})$;
3.-1: if the combination of $c_1, c_2$ is supported then
4.-1: $\text{con} = \text{infer}(c_1, c_2, \text{pivot})$; \hfill \triangleright \text{One of the rules in Table 3.1.}
5.-1: if $F', \text{con} \vdash cp \bot$ then return con; \hfill \triangleright \text{con satisfies Invar1}
6.-1: end if
7.-1: end if
8.-1: $e_1 \leftarrow \text{explain}(c_1, \text{parents}(\text{pivot}), \text{pivot})$; \hfill \triangleright \text{Fallback: use explanations.}
9.-1: $e_2 \leftarrow \text{explain}(c_2, F, \bot)$;
10.-1: return resolve($e_1, e_2, \text{pivot}$); \hfill \triangleright \text{Signed resolution}
11.-1: end function

3.4.1 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 $c(x, \overline{y})$:

$$c(x, \overline{y}) \land x \in S \land \forall x' \in S. \neg c(x', \overline{y}) \ , \tag{3.12}$$

or, equivalently, that the following implication is valid:

$$c(x, \overline{y}) \rightarrow (x \not\in S \lor \exists x' \in S. c(x', \overline{y})) \ . \tag{3.13}$$

Let $X$ denote the set of values of $x$ which have no support in $D'_y$:

$$X = \{ x' | \forall \overline{y} \in D'_y . \neg c_{12}(x', \overline{y}) \} \ . \tag{3.14}$$

Instantiating (3.13) with $c_{12}$ for $c$ and with $X$ for $S$ yields the inference rule that we call Combine:

$$\frac{c_{12}(x, \overline{y})}{(x \not\in X \lor \exists x' \in X. c_{12}(x', \overline{y}))} \quad \text{(Combine)} \quad \tag{3.15}$$

Since (3.15) is just an instantiation of (3.13), then (3.15) is clearly sound, and hence (3.8) is satisfied. To satisfy (3.9) we first prove logical entailment ($\models$), which is weaker than the requirement of (3.9) for provability ($\vdash cp$).

Lemma 3.4.1 $c^*, D'_x, D'_y \models \bot$. 

43
Proof In our case \( c^* = (x \notin \mathcal{X} \vee \exists x' \in \mathcal{X}. c_{12}(x', \overline{y})) \). Falsely assume that \( c^* \) is satisfied for an assignment of values \( a \in D'_x, \overline{b} \in D'_y \) to \( x, \overline{y} \), respectively. Consider the two disjuncts of \( c^* \):

- Suppose \( x \notin \mathcal{X} \) is satisfied. Considering the definition of \( \mathcal{X} \) in (3.14), this implies that \( a \) is supported in \( c_{12} \), or formally
  \[
  \exists y' \in D'_y. c_{12}(a, \overline{y}). \tag{3.16}
  \]
  Based on \textit{Invar1} we know that \( c_{12}(x, \overline{y}), D'_x, D'_y \models \bot \), and hence \( \forall x \in D'_x. \neg \exists \overline{y} \in D'_y. c_{12}(x, \overline{y}) \), and particularly for \( x = a, \neg \exists \overline{y} \in D'_y. c_{12}(a, \overline{y}) \), which contradicts (3.16).

- Now suppose \( \exists x' \in \mathcal{X}. c_{12}(x', \overline{y}) \) is satisfied. Expanding \( \mathcal{X} \) and substituting \( \overline{y} \) with its assignment \( \overline{b} \) yields
  \[
  \exists x'. \forall \overline{y} \in D'_y. \neg c_{12}(x', \overline{y}) \land c_{12}(x', \overline{b}).
  \]
  Since \( \overline{b} \in D'_y \) and \( \neg c_{12}(x', \overline{y}) \) is satisfied for all \( \overline{y} \in D'_y \), then it is satisfied for \( \overline{y} = \overline{b} \).
  This implies a contradiction: \( \exists x'. \neg c_{12}(x', \overline{b}) \land c_{12}(x', \overline{b}) \).

Hence, \( x \in D'_x, \overline{y} \in D'_y \) falsifies \( c^* \), which completes our proof. \( \square \) \( \square \) It is trivial to see that this lemma implies (3.9) when \( \vdash_{cp} \) is precise constraint propagation. When imprecise propagation is involved, e.g., \( \vdash_{cp} \) is defined by bounds consistency [CHLS06], 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 \textit{Combine} that we prove below are:

- There is no alternative to \( \mathcal{X} \) for replacing \( S \) in (3.13) that makes the resulting constraint stronger, and

- The signed clause that we obtain through the explanation mechanism—see Sec. 3.2.3—cannot yield a stronger consequent.

Lemma 3.4.2 Consider all possible formulas of the form \( \psi(x, \overline{y}) \equiv (x \notin P \lor \varphi(\overline{y})) \), for a given set \( P \). The strongest possible \( \varphi(\overline{y}) \), which meets all the requirements is \( \exists x' \in P.c_{12}(x', \overline{y}) \). In other words, for any \( \varphi(\overline{y}) \) which makes \( \psi(x, \overline{y}) \) meet the requirements, the following is satisfied:
  \[
  \exists x' \in P.c_{12}(x', \overline{y}) \models \varphi(\overline{y})
  \]
Proof. By negation, assume that there is an assignment $b$ to $y$ which satisfies $\exists x' \in P.c_{12}(x', b)$ but not $\varphi(b)$. This means that there is an $a \in P$ which satisfies $c_{12}(a, b)$ when $\neg \varphi(b)$. Because $c_{12}(a, b)$ is satisfied, (3.8) mandate that $\psi(a, b)$ should also be satisfied. According to the definition of $\psi$ we conclude that either $a \not\in P$ or $\varphi(b)$ have to be satisfied. But since $\neg \varphi(b)$ and, as defined, $a \in P$ we conclude that $\psi$ is unsatisfied with $a, b$. This conflict the initial assumption. This leads to the conclusion that if $\exists x' \in P.c_{12}(x', y)$ is satisfied, then $\varphi(y)$ must also be satisfied. □

Note that this lemma refers to $P$, and not $X$. This means that it does not rule out the possibility where $\exists x' \in P.c_{12}(x', y)$ is stronger than $\exists x' \in X.c_{12}(x', y)$. It is quite possible that the smaller the set $P$ is and the weaker the literal $x \not\in P$ is, the stronger $\exists x' \in P.c_{12}(x', y)$ becomes.

**Lemma 3.4.3** Consider all possible $\psi(x, y) \equiv (x \not\in P \lor \exists x' \in P.c_{12}(y))$ which satisfy the requirements. $P$ must satisfy

$$D'_x \subseteq P.$$ 

Proof. By negation assume that $D'_x \not\subseteq P$, i.e., there is a value $a$ such that $a \in D'_x$ and $a \not\in P$. Consider how this affects (3.9). According to (3.9),

$$x \not\in P \lor \exists x' \in P.c_{12}(x', y), D'_x, D'_y \vdash_{cp} \bot.$$ 

Since $a \in D'_x$ then we can replace $x$ with $a$ in the above formula, and get

$$a \not\in P \lor \exists x' \in P.c_{12}(x', y), D'_x \vdash_{cp} \bot.$$ 

But since $a$ was defined such that $a \not\in P$ then the above formula becomes

$$true \lor \exists x' \in P.c_{12}(x', y), D'_x \vdash_{cp} \bot.$$ 

This basically says $true \vdash_{cp} \bot$, which is impossible. It implies that the assumption that $D'_x \not\subseteq P$ is incorrect. □ □

**Lemma 3.4.4** Consider all possible $\psi(x, y) \equiv (x \not\in P \lor \exists x' \in P.c_{12}(y))$ which satisfy the requirements. $P$ must satisfy

$$P \subseteq X.$$ 

Where $X$ was defined above as

$$X = \{ x'| \forall y \in D'_y [ c_{12}(x', y) ] \}.$$ 

45
Proof According to the definition of \( \mathcal{X} \), the lemma can be reformulated as

\[
\forall a \in P \forall y' \in D' \quad \neg c_{12}(a, y')
\]

Assume, by negation, that this is not correct. In other words there are \( a \in P \) and \( b \in D'_y \) such that \( c_{12}(a, b) \). We will show that this conflicts (3.9).

Due to (3.9) we know that

\[
x \not\in P \lor \exists x' \in P \quad c_{12}(x', b), D'_x \vdash \bot
\]

Since \( b \in D'_y \), the formula above implies

\[
x \not\in P \lor \exists x' \in P \quad c_{12}(x', b), D'_x \vdash \bot
\]

Assuming \( \vdash_{cp} \) is precise this implies

\[
\forall x \in D'_x, \neg [x \not\in P \lor \exists x' \in P \quad c_{12}(x', b)]
\]

We now push the negation down, and get

\[
\forall x \in D'_x, [x \in P \land \forall x' \in P \quad \neg c_{12}(x', b)]
\]

First, we see that this implies \( D'_x \subseteq P \). Since \( x \) is independent in the formula we conclude that

\[
\forall x' \in P \quad \neg c_{12}(x', b)
\]

Now we go back to \( a \in P \) and \( b \in D'_y \) which guarantee \( c_{12}(a, b) \), and combine it with the formula above. This means that we can assign \( x' = a \), which leads to \( \neg c_{12}(a, b) \), which conflicts with the guarantee of \( c_{12}(a, b) \). This means that our assumption that the lemma is incorrect was wrong, hence \( P \subseteq \mathcal{X} \). □ □

This lemma implies that the literal \( x \not\in \mathcal{X} \) is the strongest possible. This does not imply anything regarding the strength of the second part of the formula, i.e., \( \exists x' \in \mathcal{X} \quad [c_{12}(x', \overline{y})] \) may be weakened by strengthening \( x \not\in P \).

Also note that the previous lemmas bound \( P \) to \( D'_x \subseteq P \subseteq \mathcal{X} \).

Theorem 3.4.1 There is no alternative \( \psi(x, \overline{y}) \), different than \( c^* \), which is stronger than \( c^* \) with \( \mathcal{X} \). In other words

\[
\psi(x, \overline{y}) \neq [x \not\in \mathcal{X} \lor \exists x' \in \mathcal{X} \quad c_{12}(x', \overline{y})]
\]

\[
\Downarrow
\]

\[
(\psi(x, \overline{y}) \not\equiv [x \not\in \mathcal{X} \lor \exists x' \in \mathcal{X} \quad c_{12}(x', \overline{y})]).
\]
**Proof** Recall that we require that $\psi(x, y)$ to be of the form $x \not\in P \lor \varphi(y)$. Lemma 3.4.2 shows that for a given $P$, the strongest possible $\varphi(y)$ is $\exists x' \in P.c_{12}(x', y)$. This leaves us to prove that

$$x \not\in P \lor \exists x' \in P.c_{12}(x', y)$$

is not stronger than

$$x \not\in \mathcal{X} \lor \exists x' \in \mathcal{X}.c_{12}(x', y).$$

According to Lemma 3.4.4 because $P$ complies with the given requirements then $P \subseteq \mathcal{X}$. If $P = \mathcal{X}$ the two formulas are equivalent and neither are stronger, otherwise $P \subset \mathcal{X}$.

Assume that $P \subset \mathcal{X}$, this means that there is a such that $a \in \mathcal{X}$ and $a \not\in P$. As a result, for $x = a$ the literal $x \not\in P$ is true and the literal $x \not\in \mathcal{X}$ is false. In this situation the $P$ based formula, i.e., $x \not\in P \lor \exists x' \in P.c_{12}(x', y)$ is true, but the $\mathcal{X}$ based formula depends solely on

$$\exists x' \in \mathcal{X}.c_{12}(x', y).$$

We look for a case where this formula is falsified when $x = a$. It can be falsified, i.e., not a tautology, since otherwise this would conflict (3.9). This means that there is an assignment $\bar{b}$ to $\bar{y}$ such that the formula is falsified. We have found $x = a$ and $y = \bar{b}$ for which the $P$ based formula is satisfied and the $\mathcal{X}$ based formula is falsified. This means that the $P$ based formula is not stronger than the $\mathcal{X}$ based formula.

Because the $P$ based formula is the strongest possible form of $x \not\in P \lor \varphi(y)$, this implies that any $\psi(x, y)$ that satisfies the requirements is not stronger than

$$x \not\in \mathcal{X} \lor \exists x' \in \mathcal{X}.c_{12}(x, y).$$

Note that this theorem does not say that, with $\mathcal{X}$, the resulting constraint is stronger than any other possibility; it says that no other constraint is stronger. In other words, there need not be a strict ordering of constraints.

### 3.4.2 Selected rules based on instantiating **Combine**

We now instantiate **Combine** (3.15) with several specific constraints of interest. The derivations rely on various properties of the domains before propagation $D'_x, D'_y$ and right after it $D''_x, D''_y$. By definition

$$c_1, D'_x, D'_y \vdash_{cp} (x \in D''_x \land y \in D''_y).$$

(3.17)

We make the following observations about these domains:

1. The domain of $x$, and possibly domains of variables in $\bar{y}$, are reduced by $c_1$:

$$D''_x \subseteq D'_x, \quad D''_y \subseteq D'_y.$$  

(3.18)

47
2. Owing to Invar1, in the context of $D''_x, D''_y$, $c_2$ detects a conflict:

$$c_2(x, \overline{y}), D''_x, D''_y \vdash_{cp} \bot.$$ \hfill (3.19)

3. $c_1$ cannot detect a conflict on its own in the context of $D'_x, D'_y$:

$$c_1, D'_x, D'_y \not\vdash_{cp} \bot.$$ \hfill (3.20)

We now use these observations when instantiating Combine.

**Rule R1:**

$$c_1 \overset{\vdash}{=} x \in X_1 \lor A_1(\overline{y})$$
$$c_2 \overset{\vdash}{=} x \in X_2 \lor A_2(\overline{y})$$

$A_1$ and $A_2$ are disjunctions of zero or more literals over the variables of $\overline{y}$. Expanding $c_{12}$ in (3.14) yields

$$X = \{x' \mid \forall \overline{y}' \in D'_y \ (x' \notin X_1 \land \neg A_1(\overline{y})) \lor (x' \notin X_2 \land \neg A_2(\overline{y}))\}.$$ 

From (3.17) and (3.18) we know that $c_1(x, \overline{y}), D'_x, D'_y \vdash_{cp} x \in D''_x$ and $D'_x \neq D''_x$, which implies that $A_1(\overline{y}), D'_y \models \bot$, and consequently simplifies the above to

$$X = \{x' \mid \forall \overline{y}' \in D'_y, x' \notin X_1 \lor (x' \notin X_2 \land \neg A_2(\overline{y}))\}.$$ 

Note that the propagation of $c_1$ in the context of $D'_y, D'_x$ results in $D''_x = X_1 \cap D'_x$ and $D''_y = D'_y$. Since $D'_y = D''_y$ and, according to (3.19), $c_2(x, \overline{y}), D''_x, D''_y \vdash_{cp} \bot$, then $A_2(\overline{y}), D'_y \models \bot$. This means that $\forall \overline{y}' \in D'_y, \neg A_2(\overline{y})$, which simplifies the above formula to

$$X = \{x' \mid \forall \overline{y}' \in D'_y, x' \notin X_1 \lor x' \notin X_2\}.$$ 

Since the inner part does not depend on $\overline{y}$ the formula is further simplified to

$$X = \{x' \mid x' \notin X_1 \lor x' \notin X_2\}.$$ 

Using this definition of $X$ we examine $c^*$:

$$c^*(x, D'_y) = (x \in (X_1 \cap X_2) \lor \exists x' \notin (X_1 \cap X_2). c_{12}(x', \overline{y})).$$

Let $A' = \exists x' \notin (X_1 \cap X_2). c_{12}(x', \overline{y})$. This simplifies the above to

$$c^*(x, D'_y) = (x \in (X_1 \cap X_2) \lor A').$$ \hfill (3.21)
We split the quantifier in $A'$ into three cases:

$$A' = \exists x' \in (X_1 \setminus X_2). \ c_{12}(x', \overline{y}) \lor$$
$$\exists x' \in (X_2 \setminus X_1). \ c_{12}(x', \overline{y}) \lor$$
$$\exists x' \not\in (X_2 \cup X_1). \ c_{12}(x', \overline{y})).$$

After taking the definitions of $c_{12}, \ c_1,$ and $c_2$ into account:

$$A' = (\exists x' \in (X_1 \setminus X_2). \ A_2(\overline{y}) \lor$$
$$\exists x' \in (X_2 \setminus X_1). \ A_1(\overline{y}) \lor$$
$$\exists x' \not\in (X_2 \cup X_1). \ A_1(\overline{y}) \land A_2(\overline{y})).$$

Next, we eliminate the $\exists$ quantifier and get

$$A' = ((X_1 \setminus X_2 \neq \emptyset \land A_2(\overline{y})) \lor$$
$$\quad (X_2 \setminus X_1 \neq \emptyset \land A_1(\overline{y})) \lor$$
$$\quad ((X_2 \cup X_1)^C \neq \emptyset \land A_1(\overline{y}) \land A_2(\overline{y}))).$$

We simplify this further by showing that $X_1 \setminus X_2 \neq \emptyset$ and $X_2 \setminus X_1 \neq \emptyset,$ which leads to $A' = (A_1(\overline{y}) \lor A_2(\overline{y})).$

According to (3.17),(3.18), $(x \in X_1 \lor A_1(\overline{y})), \ D'_x, D''_\overline{y} \models x \in D'_x,$ where $D''_x \subset D'_x,$ which implies that $A_1(\overline{y}), \ D''_\overline{y} \models \bot.$ Similarly $(x \in X_1 \lor A_1(\overline{y})), \ D''_x, D'_\overline{y} \models \bot \ implies \ that \ A_2(\overline{y}), \ D'_\overline{y} \models \bot.$ These facts show that $A_1$ and $A_2$ are falsified in this context, meaning that we can focus only on $x \in X_1$ and $x \in X_2$ parts of $c_1$ and $c_2$ there.

For the following we assume that $\vdash_{cp}$ for signed-clauses is precise, i.e., $\psi \models \phi$ iff $\psi \vdash_{cp} \phi.$

- $X_2 \setminus X_1 \neq \emptyset.$ Because $c_2(x, \overline{y}), \ D'_x, D''_\overline{y} \not\models_{cp} \bot$ and $\vdash_{cp}$ is assumed to be precise then there is an assignment $a \in D'_x, \ b \in D''_\overline{y}$ such that $c_2(a, b)$ is satisfied. Since $A_2(\overline{y}), \ D''_\overline{y} \models \bot$ and $b \in D''_\overline{y}$ we know that $A_2(\overline{b}) \models \bot,$ which implies that $a \in X_2.$ Further, since $(c_1(x, \overline{y}) \land c_2(x, \overline{y})), \ D'_x, D''_\overline{y} \models \bot$ and $c_2(a, \overline{b})$ is satisfied then $c_1(a, \overline{b}) \models \bot,$ i.e., $(a \in X_1 \lor A_1(\overline{b})), \ D''_\overline{y} \models \bot.$ This leads to $a \not\in X_1,$ which together with $c \in X_2$ implies $X_2 \setminus X_1 \neq \emptyset.$

- $X_1 \setminus X_2 \neq \emptyset.$ Because $c_1(x, \overline{y}), \ D'_x, D''_\overline{y} \not\models \bot$ and since $A_1(\overline{y}), \ D''_\overline{y} \models \bot$ we know that there is at least one element $a \in D'_x$ such that $a \in X_1.$ Again, since $(c_1(a, \overline{y}) \land c_2(a, \overline{y})), \ D''_\overline{y} \models \bot$ and $c_1(a, \overline{y}), \ D''_\overline{y} \not\models \bot$ then we know that $c_2(a, \overline{y}), \ D''_\overline{y} \models \bot.$ This implies that $a \in X_2 \models \bot,$ i.e., $a \not\in X_2.$ Since $a \not\in X_2$ and $a \in X_1$ then it follows that $X_1 \setminus X_2 \neq \emptyset.$

49
These two facts simplify $A'$ to

$$A' = (A_2(\overline{y}) \lor A_1(\overline{y})),$$

and correspondingly, according to (3.21),

$$c^*(x, \overline{y}) = (x \in (X_1 \cap X_2) \lor A_1(\overline{y}) \lor A_2(\overline{y})). \tag{3.22}$$

Note the equivalence of (3.22) and the result of signed resolution in (3.1).

**Rule R2:** $c_1 \doteq y_2 - x \geq k_2 \quad c_2 \doteq x - y_1 \geq k_1$

Expanding $c_{12}$ in (3.14) yields

$$\mathcal{X} = \{x' \mid \forall y' \in D_y', [y_2 - x < k_2 \lor x - y_1 < k_1]\}$$

$$= \{x' \mid \max(D'_{y_2}) - x < k_2 \lor x - \min(D'_{y_1}) < k_1\}$$

$$= \{x' \mid \max(D'_{y_2}) - k_2 < x \lor x < k_1 + \min(D'_{y_1})\}.$$

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

$$\mathcal{X}^c = [k_1 + \min(D'_{y_1}), \max(D'_{y_2}) - k_2]. \tag{3.23}$$

Recall (3.14): $x \not\in \mathcal{X} \lor \exists x' \in \mathcal{X}. c_{12}(x', \overline{y})$. The right disjunct is equal to:

$$\exists x'. x' \in \mathcal{X} \land [y_2 - x' \geq k_2 \land x' - y_1 \geq k_1]$$

$$= \exists x'. x' \in \mathcal{X} \land [y_2 - k_2 \geq x' \geq y_1 + k_1]$$

$$= \exists x'. x' \in \mathcal{X} \land x' \in [y_1 + k_1, y_2 - k_2]. \tag{3.24}$$

We use (3.23) to rewrite (3.24):

$$\exists x'. x' \not\in [k_1 + \min(D'_{y_1}), \max(D'_{y_2}) - k_2] \land x' \in [y_1 + k_1, y_2 - k_2],$$

which implies

$$[y_1 + k_1, y_2 - k_2] \not\subseteq [k_1 + \min(D'_{y_1}), \max(D'_{y_2}) - k_2]$$

$$= [y_1, y_2 - k_2 - k_1] \not\subseteq [\min(D'_{y_1}), \max(D'_{y_2}) - k_2 - k_1].$$

Hence, the rule is

$$y_2 - x \geq k_2 \quad x - y_1 \geq k_1 \quad (x \in [k_1 + \min(D'_{y_1}), \max(D'_{y_2}) - k_2] \lor$$

$$[y_1, y_2 - k_2 - k_1] \not\subseteq [\min(D'_{y_1}), \max(D'_{y_2}) - k_2 - k_1]). \tag{3.25}$$
Rule R4: \[ c_1 \doteq y_1 - x \geq k_1 \quad \text{and} \quad c_2 \doteq [y_2, x - k_2] \not\subseteq [a_2, b_2] \]

Note that R2 is implied by \( c_1, c_2 \). If \( y_1 - x \geq k_1 \) and \( x - y_2 \geq k_2 \) conflict \( D'_x \) and \( D'_y \) then we can simply use R2. This allows us to concentrate on the case where propagating \([y_2, x - k_2] \not\subseteq [a_2, b_2] \) \( D'_x \) and \( D'_y \) removes at least one value from the domain \( x \) more than \( x - y_2 \geq k_2 \). The extra value removed, depicted \( \gamma \), should satisfy

\[
(\exists y_2 \in D'_y, \gamma - y_2 \geq k_2) \land \forall y_2 \in D'_y, [y_2, \gamma - k_2] \subseteq [a_2, b_2].
\]

This is simplified to

\[
\gamma - \min(D'_y) \geq k_2 \land \left[ \min(D'_y), \gamma - k_2 \right] \subseteq [a_2, b_2]
\]

From this we conclude, according to the definition of interval inclusion, that

\[
\min(D'_y) \leq \gamma - k_2 \land \left( \min(D'_y) > \gamma - k_2 \lor (a_2 \leq \min(D'_y) \leq \gamma - k_2 \leq b_2) \right).
\]

Since \( \min(D'_y) \leq \gamma - k_2 \) and \( \min(D'_y) > \gamma - k_2 \) contradict it is possible to eliminate the disjunct \( \min(D'_y) > \gamma - k_2 \) and get

\[
\min(D'_y) \leq \gamma - k_2 \land a_2 \leq \min(D'_y) \leq \gamma - k_2 \leq b_2.
\]

In which we have two copies of \( \min(D'_y) \leq \gamma - k_2 \) which one of them can be eliminated

\[
a_2 \leq \min(D'_y) \leq \gamma - k_2 \leq b_2.
\]

Subsequently

\[
a_2 \leq \min(D'_y) \leq b_2. \tag{3.26}
\]

The next step is to find the value of \( X \).

\[
X = \{ x' \mid \forall y \in D'_y, \neg c_{12}(x', y) \}
\]

we get

\[
X = \{ x' \mid \forall y \in D'_y, [y_1 - x < k_1 \lor [y_2, x - k_2] \subseteq [a_2, b_2]] \}
\]

Considering the properties of \( \subseteq \) and of \( < \) we replace \( y_1 \) with \( \max(D'_y) \) and and \( y_2 \) with \( \min(D'_y) \):

\[
X = \{ x \mid \max(D'_y) - x < k_1 \lor [\min(D'_y), x - k_2] \subseteq [a_2, b_2] \}
\]

When we expand according to the definition of \( \subseteq \) we get:

\[
X = \{ x \mid \max(D'_y) - x < k_1 \lor (\min(D'_y) > x - k_2 \lor (\min(D'_y) \geq a_2 \land x - k_2 \leq b_2)) \}
\]
We apply $\min(D'_{y_2}) \geq a_2$, from Eq.(3.26), on the above equation

$$X = \{x | \max(D'_{y_1}) - x < k_1 \lor (\min(D'_{y_2}) > x - k_2 \lor x - k_2 \leq b_2)\}.$$  

We also apply $b_2 \geq \min(D'_{y_2})$, from Eq.(3.26), on the above equation

$$X = \{x | \max(D'_{y_1}) - x < k_1 \lor x - k_2 \leq b_2\}.$$  

Normalizing $x$ gives

$$X = \{x | \max(D'_{y_1}) - k_1 < x \lor x \leq b_2 + k_2\}.$$  

In terms of intervals, this can be written as

$$X = (b_2 + k_2, \max(D'_{y_1}) - k_1)^C,$$

and if we assume that all values are integers then

$$X = [b_2 + k_2 + 1, \max(D'_{y_1}) - k_1]^C.$$  

$c^* \equiv x \not\in X \lor \exists x' \in X. (y_1 - x' \geq k_1 \land [y_2, x' - k_2] \not\subseteq [a_2, b_2])$.

We explore the right disjunct

$$\exists x' \in X. (y_1 - k_1 \geq x' \land [y_2, x' - k_2] \not\subseteq [a_2, b_2]).$$  

Applying $X$ we get

$$\exists x'. [\max(D'_{y_1}) - k_1 < x' \land y_1 - x' \geq k_1 \land [y_2, x' - k_2] \not\subseteq [a_2, b_2]) \lor$$

$$\exists x'. [x' \leq b_2 + k_2 \land y_1 - x' \geq k_1 \land [y_2, x' - k_2] \not\subseteq [a_2, b_2]].$$  

Expanding the definition of $\not\subseteq$ gives

$$\exists x'. [\max(D'_{y_1}) - k_1 < x' \land y_1 - x' \geq k_1 \land y_2 \leq x' - k_2 \land y_2 < a_2] \lor$$

$$\exists x'. [\max(D'_{y_1}) - k_1 < x' \land y_1 - x' \geq k_1 \land y_2 \leq x' - k_2 \land y_2 < a_2 \land x' > b_2] \lor$$

$$\exists x'. [x' \leq b_2 + k_2 \land y_1 - x' \geq k_1 \land y_2 \leq x' - k_2 \land y_2 < a_2] \lor$$

$$\exists x'. [x' \leq b_2 + k_2 \land y_1 - x' \geq k_1 \land y_2 \leq x' - k_2 \land x' - k_2 > b_2].$$  

Now, $x' \leq b_2 + k_2$ conflicts $x' - k_2 > b_2$ in the last disjunct, which implies:

$$\exists x'. [\max(D'_{y_1}) - k_1 < x' \land y_1 - x' \geq k_1 \land y_2 \leq x' - k_2 \land y_2 < a_2] \lor$$

$$\exists x'. [\max(D'_{y_1}) - k_1 < x' \land y_1 - x' \geq k_1 \land y_2 \leq x' - k_2 \land x' - k_2 > b_2] \lor$$

$$\exists x'. [x' \leq b_2 + k_2 \land y_1 - x' \geq k_1 \land y_2 \leq x' - k_2 \land y_2 < a_2].$$  

52
We move $x'$ and replace $<$ with $\leq$ as a preparation to eliminate $x'$:

$$
\exists x'. \left[ \max(D'_{y_1}) - k_1 + 1 \leq x' \land y_1 - k_1 \geq x' \land y_2 + k_2 \leq x' \land y_2 < a_2 \right] \lor
$$

$$
\exists x'. \left[ \max(D'_{y_1}) - k_1 + 1 \leq x' \land y_1 - k_1 \geq x' \land y_2 + k_2 \leq x' \land x' \geq b_2 + k_2 + 1 \right] \lor
$$

$$
\exists x'. \left[ x' \leq b_2 + k_2 \land y_1 - k_1 \geq x' \land y_2 + k_2 \leq x' \land y_2 < a_2 \right].
$$

We eliminate $x'$ in:

$$
\left[ \max(D'_{y_1}) - k_1 + 1 \leq y_1 - k_1 \land y_2 + k_2 \leq y_1 - k_1 \land y_2 < a_2 \right] \lor
$$

$$
\left[ \max(D'_{y_1}) - k_1 + 1 \leq y_1 - k_1 \land y_2 + k_2 \leq y_1 - k_1 \land y_1 - k_1 \geq b_2 + k_2 + 1 \right] \lor
$$

$$
\left[ y_2 + k_2 \leq b_2 + k_2 \land y_1 - k_1 \geq y_2 + k_2 \land y_2 < a_2 \right].
$$

After some normalization:

$$
\left[ \max(D'_{y_1}) + 1 \leq y_1 \land y_2 + k_2 \leq y_1 - k_1 \land y_2 < a_2 \right] \lor
$$

$$
\left[ \max(D'_{y_1}) + 1 \leq y_1 \land y_2 + k_2 \leq y_1 - k_1 \land y_1 - k_1 \geq b_2 + k_2 + 1 \right] \lor
$$

$$
\left[ y_2 \leq b_2 \land y_1 - k_1 \geq y_2 + k_2 \land y_2 < a_2 \right].
$$

In the last disjunct, since $y_2 < a_2$ and we know that $a_2 \leq b_2$ then $y_2 \leq b$ is redundant and the last disjunct becomes

$$
y_1 - k_1 \geq y_2 + k_2 \land y_2 < a_2.
$$

This subsumes the first disjunct:

$$
\max(D'_{y_1}) + 1 \leq y_1 \land y_2 + k_2 \leq y_1 - k_1 \land y_2 < a_2
$$

So we are left with

$$
\left[ \max(D'_{y_1}) + 1 \leq y_1 \land y_2 + k_2 \leq y_1 - k_1 \land y_1 - k_1 \geq b_2 + k_2 + 1 \right] \lor
$$

$$
\left[ y_1 - k_1 \geq y_2 + k_2 \land y_2 < a_2 \right].
$$

We now define $k_1 + k_2 = k^*$ which leaves us with:

$$
\left[ \max(D'_{y_1}) + 1 \leq y_1 \land y_2 + k^* \leq y_1 \land y_1 \geq b_2 + k^* + 1 \right] \lor
$$

$$
\left[ y_1 \geq y_2 + k^* \land y_2 < a_2 \right].
$$

53
Combining conjunctions:

\[
\begin{align*}
[y_2 + k^* &\leq y_1 \land y_1 \geq \max(b_2 + k^* + 1, \max(D'_{y_1}) + 1)] \lor \\
y_1 \geq y_2 + k^* \land y_2 < a_2
\end{align*}
\]

This can be written as a interval constraint:

\[
[y_2 + k^*, y_1] \not\subseteq [a_2 + k^*, \max(b_2 + k^*, \max(D'_{y_1}))]
\]

We can subtract \( k^* \) from all sides and get:

\[
[y_2, y_1 - k_1 - k_2] \not\subseteq [a_2, \max(b_2, \max(D'_{y_1}) - k_1 - k_2)]
\]

The rule then becomes

\[
y_1 - x \geq k_1 \quad [y_2, x - k_2] \not\subseteq [a_2, b_2]
\]

\[
b_2 + k_2 < x \leq \max(D'_{y_1}) - k_1 \lor \\
y_2, y_1 - k_1 - k_2] \not\subseteq [a_2, \max(b_2, \max(D'_{y_1}) - k_1 - k_2)]
\]

**Rule R5:**  
\[
c_1 \equiv [y_1, x] \not\subseteq [a_1, b_1] \\
c_2 \equiv [x, y_2] \not\subseteq [a_2, b_2]
\]

We demand that at the point of conflict, replacing either one of the constraints with a plain \( \leq \) will not detect a conflict. If replacing with \( \leq \) detects the conflict then we simply employ R2. This means that \( [y_1, x] \not\subseteq [a_1, b_1] \) removes at least one value from \( D'_{x} \) more than \( y_1 \leq x \). The extra value removed \( \gamma \) should satisfy

\[
\min(D'_{y_1}) \leq \gamma \land \forall y_1 \in D'_{y_1}, [y_1, \gamma] \subseteq [a_1, b_1].
\]

This means that

\[
\min(D'_{y_1}) \leq \gamma \land [\min(D'_{y_1}), \gamma] \subseteq [a_1, b_1].
\]

From this we conclude that

\[
a_1 \leq \min(D'_{y_1}) \leq \gamma \leq b_1.
\]

Similarly from the second constraint we conclude that

\[
a_2 \leq \max(D'_{y_2}) \leq b_2.
\]
We require that neither constraints detect a conflict by themselves with $D'_{y_1}$, $D'_{y_2}$, and $D'_{x}$, but when propagated consecutively they reach a conflict. This means that there is at least one value $\gamma \in D'_{x}$ such that

$$\exists y_1 \in D'_{y_1}, [y_1, \gamma] \not\subseteq [a_1, b_1] \wedge$$
$$\forall y_2 \in D'_{y_2}, [\gamma, y_2] \subseteq [a_2, b_2]$$

This can be simplified to

$$[\min(D'_{y_1}), \gamma] \not\subseteq [a_1, b_1] \wedge [\gamma, \max(D'_{y_2})] \subseteq [a_2, b_2].$$

If such $\gamma$ exists then it can be equal to $\max(D'_{x})$ such that

$$[\min(D'_{y_1}), \max(D'_{x})] \not\subseteq [a_1, b_1] \wedge \max(D'_{y_2}) \subseteq [a_2, b_2] \wedge a_2 \leq \max(D'_{y_2}) \leq b_2.$$

To summarize, we have concluded that

$$a_1 \leq \min(D'_{y_1}) \leq b_1 \wedge \min(D'_{y_1}), \max(D'_{x})] \not\subseteq [a_1, b_1] \wedge$$
$$[\max(D'_{x}), \max(D'_{y_2})] \subseteq [a_2, b_2] \wedge a_2 \leq \max(D'_{y_2}) \leq b_2.$$

The interval expressions can be expanded such that

$$a_1 \leq \min(D'_{y_1}) \leq b_1 \wedge \min(D'_{y_1}) \leq \max(D'_{x}) \wedge$$
$$(\min(D'_{y_1}) < a_1 \lor \max(D'_{x}) > b_1) \wedge (\max(D'_{x}) > \max(D'_{y_2}) \lor (\max(D'_{x}) \geq a_2 \wedge \max(D'_{y_2}) \leq b_2)) \wedge a_2 \leq \max(D'_{y_2}) \leq b_2.$$

This is simplified to

$$a_1 \leq \min(D'_{y_1}) \leq b_1 < \max(D'_{x}) \wedge \max(D'_{x}) \geq a_2 \wedge a_2 \leq \max(D'_{y_2}) \leq b_2.$$

We will depict this as

$$\psi \triangleq (a_1 \leq \min(D'_{y_1}) \leq b_1 < \max(D'_{x}) \wedge \max(D'_{x}) \geq a_2 \wedge a_2 \leq \max(D'_{y_2}) \leq b_2). \quad (3.27)$$

We require that the resulting constraint be of the form

$$x \not\in \mathcal{X} \lor [y_1, y_2] \not\subseteq [a^*, b^*],$$

and meet the soundness and completeness requirements. First we look for $\mathcal{X}$ which is defined by

$$\mathcal{X} = \{ x' | \forall y_1 \in D'_{y_1}, \forall y_2 \in D'_{y_2}, [y_1, x'] \subseteq [a_1, b_1] \lor [x', y_2] \subseteq [a_2, b_2] \}.$$
We expand the interval operators to

\[ \mathcal{X} = \{ x' \mid \forall y_1 \in D_{y_1}' \forall y_2 \in D_{y_2}' \}. \]

\[ y_1 > x' \lor (a_1 \leq y_1 \land x' \leq b_1) \lor x' > y_2 \lor (a_2 \leq x' \land y_2 \leq b_2) \}. \]

Since \( y_1 \) is bounded only from below and \( y_2 \) only from above then we can safely rewrite this expression for the worse cases of \( y_1 \) and \( y_2 \) which are \( \min(D_{y_1}') \) and \( \max(D_{y_2}') \):

\[ \mathcal{X} = \{ x' \mid \min(D_{y_1}') > x' \lor (a_1 \leq \min(D_{y_1}') \land x' \leq b_1) \lor x' > \max(D_{y_2}') \lor (a_2 \leq x' \land \max(D_{y_2}') \leq b_2) \}. \]

From the initial assumptions, as expressed by Equation (3.27), we know that \( a_1 \leq \min(D_{y_1}') \) and \( \max(D_{y_2}') \leq b_2 \) are true. From, this we conclude that that \( (a_1 \leq \min(D_{y_1}')) \land x' \leq b_1 \) can be simplified to \( x' \leq b_1 \) and that \( (a_2 \leq x' \land \max(D_{y_2}') \leq b_2) \) can be simplified to \( a \leq x'. \) This yields

\[ \mathcal{X} = \{ x' \mid \min(D_{y_1}') > x' \lor x' \leq b_1 \lor x' > \max(D_{y_2}') \lor a \leq x' \}. \]

Equation (3.27) also states that \( \min(D_{y_1}') \leq b_1 \) and \( a_2 \leq \max(D_{y_2}') \). This makes both \( \min(D_{y_1}') > x' \) and \( x' > \max(D_{y_2}') \) redundant in the expression above. So the expression becomes

\[ \mathcal{X} = \{ x' \mid x' \leq b_1 \lor a \leq x' \}. \]

\[ X = (-\infty, b_1] \cup [a_2, \infty). \]

We will look for the values of \( a^* \) and \( b^* \) which produce the strongest \( y_1, y_2 \not\subseteq [a^*, b^*] \) part which meets our requirements. This can be achieved by finding the smallest \( a^* \) and the biggest possible \( b^* \). First we require that the constraint is sound, i.e., derivable from the two original constraints:

\[ \forall x \forall y_1 \forall y_2. [\psi \land ([y_1, x] \not\subseteq [a_1, b_1] \land [x, y_2] \not\subseteq [a_2, b_2]) \rightarrow (x \not\in \mathcal{X} \lor [y_1, y_2] \not\subseteq [a^*, b^*])]. \]

Where \( \psi \) is defined in Equation (3.27). At first glance it seems that \( \psi \) does not effect the expression. We will omit \( \psi \) as it will, at most, strengthen the expression. Assume by negation that this is not so, i.e., there are \( x, y_1, y_2 \) that violate it. This means that

\[ [y_1, x] \not\subseteq [a_1, b_1] \land [x, y_2] \not\subseteq [a_2, b_2] \land (x \leq b_1 \lor a_2 \leq x) \land [y_1, y_2] \subseteq [a^*, b^*]. \]

From \( [y_1, x] \not\subseteq [a_1, b_1] \land [x, y_2] \not\subseteq [a_2, b_2] \) we conclude that \( y_1 \leq x \leq y_2 \). This means that \( [y_1, y_2] \subseteq [a^*, b^*] \) can be replaced with \( y_1 \geq a^* \land y_2 \leq b^* \):

\[ [y_1, x] \not\subseteq [a_1, b_1] \land [x, y_2] \not\subseteq [a_2, b_2] \land (x \leq b_1 \lor a_2 \leq x) \land y_1 \geq a^* \land y_2 \leq b^*. \]
The best case scenario is when \( y_1 = a^* \) and \( y_2 = b^* \) as they maximize the intervals \( [y_1, x] \) and \( [x, y_2] \). This gives

\[
[a^*, x] \not\subseteq [a_1, b_1] \land [x, b^*] \not\subseteq [a_2, b_2] \land (x \leq b_1 \lor a_2 \leq x).
\]

Consider the two possibilities for \( x \), as expressed in the above expression in \( x \leq b_1 \lor a_2 \leq x \). If \( x \leq b_1 \) then \( [a^*, x] \not\subseteq [a_1, b_1] \) implies \( a^* < a_1 \) and if \( x \geq a_2 \) then \( [x, b^*] \not\subseteq [a_2, b_2] \) implies \( b^* > b_2 \). In order to violate this, i.e., there will be no \( x \) to satisfy this equation, we require that \( a^* \geq a_1 \land b_2 \leq b^* \). So it seems that the strongest constraint that matches the requirements is

\[
x \not\in \mathcal{X} \lor [y_1, y_2] \not\subseteq [a_1, b_2].
\]

When

\[
\mathcal{X} = \{x' \mid x' \leq b_1 \lor a_2 \leq x'\}.
\]

- Soundness: We need to check if

\[
(a_1 \leq \min(D'_{y_1}) \leq b_1 \land a_2 \leq \max(D'_{y_2}) \leq b_2 \land [y_1, x] \not\subseteq [a_1, b_1] \land [x, y_2] \not\subseteq [a_2, b_2] ) \quad = \quad (x \not\in \mathcal{X} \lor [y_1, y_2] \not\subseteq [a_1, b_2]).
\]

By negation assume that this is not so, i.e., there are \( x, y_1, y_2 \) such that

\[
a_1 \leq \min(D'_{y_1}) \leq b_1 \land a_2 \leq \max(D'_{y_2}) \leq b_2 \land [y_1, x] \not\subseteq [a_1, b_1] \land [x, y_2] \not\subseteq [a_2, b_2] \land
\]

\[
x \in \mathcal{X} \land [y_1, y_2] \subseteq [a_1, b_2].
\]

We expand the definition of \( \mathcal{X} \) and get

\[
a_1 \leq \min(D'_{y_1}) \leq b_1 \land a_2 \leq \max(D'_{y_2}) \leq b_2 \land [y_1, x] \not\subseteq [a_1, b_1] \land [x, y_2] \not\subseteq [a_2, b_2] \land
\]

\[
(b_1 \geq x \lor x \geq a_2) \land [y_1, y_2] \subseteq [a_1, b_2]]
\]

If \( x \leq b_1 \) then \( [y_1, x] \not\subseteq [a_1, b_1] \) implies \( y_1 < a_1 \) which conflicts \( [y_1, y_2] \subseteq [a_1, b_2] \). Otherwise, \( a_2 \geq x \) and \( [x, y_2] \not\subseteq [a_2, b_2] \) implies \( y_2 > b_2 \) which conflicts \( [y_1, y_2] \subseteq [a_1, b_2] \). This means that there exists no \( x \) that satisfies the above expression, which conflicts our assumption that the new constraint is not derived from the original two interval constraints.
• Completeness: We need to check that

$$\psi \rightarrow (x \notin X \lor [y_1, y_2] \notin [a_1, b_2], D_{x}^\prime, D_{y_1}^\prime, D_{y_2}^\prime \vdash \emptyset)$$.

Where $\psi$ is an expression defined by Equation (3.27). By negation we assume that

$$\psi \rightarrow \exists x \in D_{x}^\prime \exists y_1 \in D_{y_1}^\prime \exists y_2 \in D_{y_2}^\prime, [x \notin X \lor [y_1, y_2] \notin [a_1, b_2]]$$.

The part of $x$ can be simplified such that

$$\psi \rightarrow \exists y_1 \in D_{y_1}^\prime \exists y_2 \in D_{y_2}^\prime, [D_{x}^\prime \setminus X \neq \emptyset \lor [y_1, y_2] \notin [a_1, b_2]]$$.

But according to the construction of $X$ we know that $D_{x}^\prime \setminus X = \emptyset$, so the expression is further simplified to

$$\psi \rightarrow \exists y_1 \in D_{y_1}^\prime \exists y_2 \in D_{y_2}^\prime, [y_1 \notin [a_1, b_2] \lor [y_1, y_2] \subset [a_1, b_2]]$$.

We expand it according to the definition of interval constraints:

$$\psi \rightarrow \exists y_1 \in D_{y_1}^\prime \exists y_2 \in D_{y_2}^\prime, [y_1 \leq y_2 \land (y_1 < a_1 \lor b_2 < y_2)]$$.

Since $y_1$ is bounded only from above and $y_2$ is bounded from below then we can replace $y_1$ with $\min(D_{y_1}^\prime)$ and $y_2$ with $\max(D_{y_2}^\prime)$:

$$\psi \rightarrow [\min(D_{y_1}^\prime) \leq \max(D_{y_2}^\prime) \land (\min(D_{y_1}^\prime) < a_1 \lor b_2 < \max(D_{y_2}^\prime))]$$.

From the definition of $\psi$ we know that $\min(D_{y_1}^\prime) < a_1$ and $b_2 < \max(D_{y_2}^\prime)$ are false, so $(\min(D_{y_1}^\prime) < a_1 \lor b_2 < \max(D_{y_2}^\prime))$ is false and the expression above is falsified. This means that our assumption was wrong, i.e., our completeness requirement is not violated.

To summarize this part, the Combination rule is

$$\frac{\begin{array}{c} [y_1, x] \not\subseteq [a_1, b_1] \\ [x, y_2] \not\subseteq [a_2, b_2] \end{array}}{a_2 > x > b_1 \lor [y_1, y_2] \not\subseteq [a_1, b_2]} \psi.$$

(3.28)

**Rule R6:**

$$c_1 = [x, y] \not\subseteq [a_1, b_1] \quad c_2 = [y, x] \not\subseteq [a_2, b_2]$$

We have $c_1 \land c_2 \rightarrow x = y$, because otherwise, e.g., if $x < y$, then $c_2$ is trivially false. Since $x = y$ then their joint value cannot be contained in either of the ranges $[a_1, b_1], [a_2, b_2]$. Hence,

$$c_{12} = (x = y) \land x \not\subseteq [a_1, b_1] \land x \not\subseteq [a_2, b_2],$$

(3.29)

58
which implies that
\[
X' = \{ x' \mid \forall y \in D'_y, \, x \neq y \lor x \in [a_1, b_1] \lor x \in [a_2, b_2] \} \\
= D'_y \cup [a_1, b_1] \cup [a_2, b_2] .
\]

The consequent of \textit{Combine} is
\[
c^* = (x \not\in X \lor \exists x \in X. \, x = y \land x \not\in [a_1, b_1] \land x \not\in [a_2, b_2]) \\
= (x \not\in X \lor (y \in X \land y \not\in [a_1, b_1] \land y \not\in [a_2, b_2])) \\
= (x \not\in X \lor ((y \in [a_1, b_1] \lor y \in [a_2, b_2] \lor y \not\in D'_y) \land y \not\in [a_1, b_1] \land y \not\in [a_2, b_2])) \\
= (x \not\in X \lor (y \not\in D'_y \land y \not\in [a_1, b_1] \land y \not\in [a_2, b_2])) \\
= (x \in (D'_y \setminus ([a_1, b_1] \cup [a_2, b_2])) \lor y \not\in (D'_y \cup [a_1, b_1] \cup [a_2, b_2])).
\]

Hence, the resulting rule in this case is
\[
\frac{[x, y] \not\subseteq [a_1, b_1] \quad [y, x] \not\subseteq [a_2, b_2]}{(x \in (D'_y \setminus ([a_1, b_1] \cup [a_2, b_2])) \lor y \not\in (D'_y \cup [a_1, b_1] \cup [a_2, b_2]))} \quad (3.30)
\]

### 3.4.3 Selected rules not based on \textit{Combine}

\textbf{Rule R3:} \quad c_1 \doteq y_1 \leq x \quad 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 \(\psi\), we now develop \(X\). \(\psi\) means that \([x, y_2] \not\subseteq [a, b]\) removes at least one value from \(D'_y\) more than \(x \leq y_2\). The extra value removed \(\alpha\) should satisfy
\[
\alpha \leq \max(D'_y) \land \forall y_2 \in D'_y. \, [\alpha, y_2] \subseteq [a, b] .
\]

This means that
\[
\alpha \leq \max(D'_y) \land [\alpha, \max(D'_y)] \subseteq [a, b] .
\]

From this we conclude that
\[
a \leq \alpha \leq \max(D'_y) \leq b .
\]

We require that neither constraints detect a conflict by themselves with \(D'_y\), \(D'_y\), and \(D'_x\), but when propagated consecutively they reach a conflict. This means that there is at least one value \(\alpha \in D'_y\) such that
\[
\exists y_1 \in D'_y. \, y_1 \leq \alpha \land \forall y_2 \in D'_y. \, [\alpha, y_2] \subseteq [a, b]
\]
This can be simplified to
\[
\min(D'_{y_1}) \leq \alpha \land [\alpha, \max(D'_{y_2})] \subseteq [a, b].
\]
If such \( \alpha \) exists then it can be equal to \( \max(D'_x) \) such that
\[
\min(D'_{y_1}) \leq \max(D'_x) \land [\max(D'_x), \max(D'_{y_2})] \subseteq [a, b].
\]
To summarize, we have concluded that
\[
a \leq \max(D'_{y_2}) \leq b \land \min(D'_{y_1}) \leq \max(D'_x) \land [\max(D'_x), \max(D'_{y_2})] \subseteq [a, b].
\]
The interval expression can be expanded such that
\[
a \leq \max(D'_{y_2}) \leq b \land \min(D'_{y_1}) \leq \max(D'_x) \land (\max(D'_x) > \max(D'_{y_2}) \lor \max(D'_x) > \max(D'_{y_2}) \land a \leq \max(D'_x)).
\]
This is simplified to
\[
a \leq \max(D'_{y_2}) \leq b \land \min(D'_{y_1}) \leq \max(D'_x) \land (\max(D'_x) > \max(D'_{y_2}) \lor a \leq \max(D'_x)).
\]
Since \( a \leq \max(D'_{y_2}) \) then if \( \max(D'_x) > \max(D'_{y_2}) \) is satisfied then \( \max(D'_x) > \max(D'_{y_2}) \geq a \), i.e., \( D'_x \geq a \), is implied. This means that \( \max(D'_x) > \max(D'_{y_2}) \) is redundant in the above expression, generating:
\[
a \leq \max(D'_{y_2}) \leq b \land \min(D'_{y_1}) \leq \max(D'_x) \land a \leq \max(D'_x).
\]
We will depict this as
\[
\psi' \triangleq (a \leq \max(D'_{y_2}) \leq b \land \min(D'_{y_1}) \leq \max(D'_x) \land a \leq \max(D'_x)).
\]
We require that the resulting constraint be of the form
\[
x \not\in \mathcal{X} \lor [y_1, y_2] \not\subseteq [a^*, b^*],
\]
and meet the soundness and completeness requirements. First we look for \( \mathcal{X} \) which is defined by
\[
\mathcal{X} = \{ x' | \forall y_1 \in D'_{y_1}, \forall y_2 \in D'_{y_2}, [y_1 > x' \lor [x', y_2] \subseteq [a, b]] \}. \]
We expand the interval operator to
\[
\mathcal{X} = \{ x' | \forall y_1 \in D'_{y_1}, \forall y_2 \in D'_{y_2}, [y_1 > x' \lor x' > y_2 \lor (a \leq x' \land y_2 \leq b)] \}. \]
Since $y_1$ is bounded only from below and $y_2$ only from above then we can safely rewrite this expression for the worse cases of $y_1$ and $y_2$ which are $\min(D'_{y_1})$ and $\max(D'_{y_2})$:

$$\mathcal{X} = \{x' | \min(D'_{y_1}) > x' \land x' > \max(D'_{y_2}) \land (a \leq x' \land \max(D'_{y_2}) \leq b)\}.$$ 

From the initial assumptions, as expressed by Equation (3.31), we know that $\max(D'_{y_2}) \leq b$ is true. From, this we conclude that that $(a \leq x' \land \max(D'_{y_2}) \leq b)$ can be simplified to $a \leq x'$. This yields

$$\mathcal{X} = \{x' | \min(D'_{y_1}) > x' \land x' > \max(D'_{y_2}) \land a \leq x'\}.$$ 

Equation (3.31) also states that $a \leq \max(D'_{y_2})$ which makes $x' > \max(D'_{y_2})$ redundant in $x' > \max(D'_{y_2}) \land a \leq x$. So the expression becomes

$$\mathcal{X} = \{x' \mid x' < \min(D'_{y_1}) \lor a \leq x'\}.$$ (3.32)

We propose the following consequent:

$$c^* \triangleq x \notin \mathcal{X} \lor [y_1, y_2] \not\subseteq [\min(D'_{y_1}), b]$$ (3.33)

$$= a > x \geq \min(D'_{y_1}) \lor [y_1, y_2] \not\subseteq [\min(D'_{y_1}), b].$$ (3.34)

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 (3.34) satisfies (3.8) and (3.9):

- **Eq. (3.8):** Falsely assume the contrary, i.e., there are $x, y_1, y_2$ such that

  $$a \leq \max(D'_{y_2}) \leq b \land \min(D'_{y_1}) \leq \max(D'_{y_1}) \land a \leq \max(D'_{y_1}) \land y_1 \leq x$$

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

  Expanding $\mathcal{X}$ yields

  $$a \leq \max(D'_{y_2}) \leq b \land \min(D'_{y_1}) \leq \max(D'_{y_1}) \land a \leq \max(D'_{y_1}) \land y_1 \leq x$$

  $$\land [x, y_2] \not\subseteq [a, b] \land (x < \min(D'_{y_1}) \lor a \leq x) \land [y_1, y_2] \subseteq [\min(D'_{y_1}), b].$$

  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. (3.9):** We need to check that

  $$\psi' \rightarrow (x \notin \mathcal{X} \lor [y_1, y_2] \not\subseteq [\min(D'_{y_1}), b], D'_x, D'_{y_1}, D'_{y_2} \vdash \emptyset),$$

  61
where $\psi'$ is an expression defined by Equation (3.31). Falsely assume that

$$
\psi' \rightarrow \exists x \in D'_x \exists y_1 \in D'_{y_1} \exists y_2 \in D'_{y_2}. x \notin X \lor [y_1, y_2] \not\subseteq [\min(D'_{y_1}), b].
$$

The part of $x$ can be simplified such that

$$
\psi' \rightarrow \exists y_1 \in D'_{y_1} \exists y_2 \in D'_{y_2}. D'_x \setminus X \neq \emptyset \lor [y_1, y_2] \not\subseteq [\min(D'_{y_1}), b].
$$

But according to the construction of $X$ we know that $D'_x \setminus X = \emptyset$, so the expression is further simplified to

$$
\psi' \rightarrow \exists y_1 \in D'_{y_1} \exists y_2 \in D'_{y_2}. [y_1, y_2] \not\subseteq [\min(D'_{y_1}), b],
$$

or

$$
\psi' \rightarrow \exists y_1 \in D'_{y_1} \exists y_2 \in D'_{y_2}. y_1 \leq y_2 \land (y_1 < \min(D'_{y_1}) \lor b < y_2).
$$

Since $y_1$ is bounded only from above and $y_2$ is bounded from below then we can replace $y_1$ with $\min(D'_{y_1})$ and $y_2$ with $\max(D'_{y_2})$:

$$
\psi' \rightarrow [\min(D'_{y_1}) \leq \max(D'_{y_2}) \land (\min(D'_{y_1}) < \min(D'_{y_1}) \lor b < \max(D'_{y_2}))].
$$

min($D'_{y_1}$) < min($D'_{y_1}$) is clearly false, and from the definition of $\psi'$ we know that $b < \max(D'_{y_2})$ is false. This falsifies min($D'_{y_1}$) < min($D'_{y_1}$) \lor b < max($D'_{y_2}$) from the expression, hence the whole expression is also falsified. This means that our assumption was wrong, i.e., (3.9) is not violated.

To summarize, the rule is

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

(3.35)

Rule R7: $c_1 = (y \leq x + k_1)$ $\quad c_2 = (x \leq y + k_2)$

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 (3.8) and (3.9) are satisfied trivially.

### 3.5 Experimental results

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
Figure 3.2: Number of instances solved within the given time limit comparing HCSP, EXPLAIN, and MISTRAL. (a) Shows the time in linear scale; (b) A zoom-in of the left figure showing the cross-over between MISTRAL and HCSP occurring after 1-2 seconds.

explanations, as described in Sec. 3.2 (from hereon—EXPLAIN)\(^4\) (3) MISTRAL [Heb08] latest version (1.550).

To evaluate the three alternatives we used a subset of benchmarks of the Fourth International CSP Solver competition [C0909]. 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 \{<, ≤, ≥, >\} (the reason being that the rules in Table 3.1 refer to combinations of constraints based on these operators and constraints that are consequents of these rules).

All tools were compiled with gcc-4.8.1 for 32-bit Intel architecture, and were run on a 4 core Intel® Xeon® 2.5GHz with 3GiB RAM. We have set a hard limit of 1GiB memory usage and 1200 seconds of CPU-time. Out of memory and time-outs are called ‘fails’ in the discussion below.

Fig. 3.2 compares the three engines. Number of fails in HCSP was 25% less than MISTRAL. Number of fails when using Combine was 4.9% less than EXPLAIN. In Fig.3.3(a) we see that the conflict analysis of HCSP is not beneficial for runs below 1.5 seconds, where MISTRAL ran faster.

Fig. 3.3 compares the number of backtracks considering only non-failing runs in all solvers (log-scale). The average number of backtracks in HCSP is 2045, in EXPLAIN 4389, and in MISTRAL 49562. As noted in the introduction, this drastic difference in the average backtrack-count indicates that the cost of learning is compensated-for by a better search.

HCSP is written in C++, contains 23k lines of non-comment code, and its architecture

\(^4\)We emphasize that this is a far-improved engine in comparison to four years ago [VS10a] owing to numerous optimizations that are beyond the scope of the current article, including an improved decision heuristic, better choice of explanations, and common-sub-expression elimination.
enables the addition of new constraints and new rules without changing the core solver. It is free software [Vek] under the GPL license.

**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 [VS10a].
Chapter 4

Architecture and Capabilities of HCSP

4.1 The solving algorithm of HCSP

4.2 Propagating constraints

AC3 [Mac77] maintains a queue of constraints which contains constraints that need to be revised. Every iteration of the algorithm removes a constraint from the queue and performs a \textit{revise} operation over it, i.e., the algorithm propagates the constraint. Unlike AC3, HCSP maintains two queues:

- A queue of variables that have not been checked since their domains have been reduced. The algorithm removes a variable from the variable queue and revises the constraints affecting this variable. As an optimization, not all affecting constraints are revised, only those that meet the watch criterion. Watches are described in Subsection 4.3.4. This is the prime queue of the algorithm, i.e., in most cases this will be the only queue used during propagation.

- A queue of constraints that should be activated. The constraints in this queue are considered only when the variable queue is empty. A constraint may enter this queue only after it was encountered by the variable queue, and only if the constraint’s propagation algorithm requests to do so.

This mechanism resembles Minion which, as described in [GMN], also uses a variable queue for all constraints except for \texttt{AllDifferent} which goes into the constraint queue. Many papers including [SS08] suggest using a priority constraint queue, where slow constraints get a lower priority. It would be interesting to attempt constraint priority queue for propagating constraints in HCSP.

The propagation algorithm in HCSP is described in Algorithm 4.1.
Algorithm 4.1 The HCSP constraint propagation algorithm

1: function Propagate($C, V, D, Q_v$)
2:     $Q_c \leftarrow \emptyset$
3:     while $Q_v \neq \emptyset \lor Q_c \neq \emptyset$ do
4:         if $Q_v \neq \emptyset$ then
5:             $(v, c) \leftarrow \text{pop}_\text{front}(Q_v)$  \hspace{1em} \text{o}f If $c \neq \bot$, then $c \in C$ inserted $v$ into $Q_v$
6:             for all $r \in \text{Watched}(v) \setminus \{c\}$ do
7:                 result$\leftarrow \text{Revise}(r, D, Q_v, Q_c)$ \hspace{1em} \text{propagate} $r$ and update $Q_v, Q_c$
8:                     if Empty(result) then
9:                         return CONFLICT
10:                    end if
11:             end for
12:         else
13:             $r \leftarrow \text{pop}_\text{front}(Q_c)$
14:             result$\leftarrow \text{Revise}(r, D, Q_v, Q_c)$
15:                     if Empty(result) then
16:                         return CONFLICT
17:                    end if
18:         end if
19:     end while
20: end function

21: function Revise($r, D, Q_v, Q_c$)
22:     result$\leftarrow \text{propagate}(r, D)$
23:     if result $= \text{CONFLICT}$ then
24:         return result
25:     else if result $= \text{POSTPONE}$ then
26:         if $r \notin Q_c$ then $Q_c.\text{push}_\text{back}(r)$
27:         end if
28:     else
29:         $D \leftarrow D$ apply changes from result.
30:         for all $v'$ modified by result do
31:             $\text{push}_\text{back}(Q_v,(v', r))$
32:         end for
33:     end if
34: end function

4.2.1 The decision heuristic of HCSP

The decision heuristic of HCSP is based on a mix of Chaff’s VSIDS [MMZ+01a], dom/wdeg strategy [BHLS04], and phase saving [Sht00] as follows. HCSP maintains a weight value for each variable and makes decisions based on that value. The weights, which are at the core of the decision heuristic, are floating-point values which are modified according to the following strategy:

- The initial weight of a variable is set to the number of constraints that constrain it.
• Each constraint that participates in conflict analysis increases the weights of all affected variables by a factor.

• After each backtrack the increase factor is increased by approximately 4%, effectively lowering the weights by 4%.

For a variable \( v \) that participated in a recent conflict analysis HCSP calculates \( h(v) = |D_v|/\text{weight}(v) \), and for all other variables it calculates \( h(v) = 8 \times |D_v|/\text{weight}(v) \). Then the decision heuristics selects the variable with the lowest value of \( h(v) \). Since a variable that participated in the latest conflict analysis gets lower \( h(v) \) it has a bigger chance of being selected. Measurements show that this difference in \( h(v) \) improves performance.

This is similar to the conclusions of [BS10] which showed that sometimes it is beneficent to retry the same variable after backtrack, and in other cases it is beneficent to try a different variable. They showed that instead of a-priori choosing either one of the methods it is better to combine both. They lean towards, without restricting to, repeating the same variable. Their experiments show that not only the combined method never performs much worse than the alternatives, it sometimes performs better than the alternatives.

The variable ordering policy of HCSP resembles \( \text{dom}/w\text{deg} \) strategy of CSP, where both \( \text{weight}(v) \) and \( w\text{deg} \) increase with conflicts. The two main differences are that: (1) \( w\text{deg} \) counts only the conflicting constraints when \( \text{weight}(v) \) counts all constraint participating in conflict analysis; (2) \( \text{weight}(v) \), unlike \( w\text{deg} \), does not depend on the current domains of neighboring variables.

The decision is performed by Algorithm 4.2.

\begin{algorithm}
\caption{Value ordering heuristic in HCSP}
\begin{algorithmic}[1]
\Function{SelectValue}{\( D_v \)}
\If{previously-assigned}
\State \( d \leftarrow \text{previous}(v) \)
\If{\( d \in D_v \)}
\State \Return \( d \)
\ElsIf{\( d \geq \min(D_v) \land d \leq (\min(D_v) \ast 2 + \max(D_v))/3 \)}
\State \Return \( \min(D_v) \)
\Comment{The first 1/3 of the domain}
\ElsIf{\( d \leq \max(D_v) \land d \geq (\max(D_v) \ast 2 + \min(D_v))/3 \)}
\State \Return \( \max(D_v) \)
\Comment{The third 1/3 of the domain}
\ElsIf{\( \text{random}(2) = 0 \)}
\State \Return \( \max(D_v) \)
\EndIf
\EndIf
\EndFunction
\end{algorithmic}
\end{algorithm}
Once the variable, \( v \), is decided HCSP has to choose which value to assign to it. If the variable was previously assigned to value \( d \) and \( d \in D_v \) then the decision is set to \( v \leftarrow d \). Otherwise, the range of values \([\min(D_v), \max(D_v)]\) is partitioned into three partitions. If \( d \) is in the lower partition then the decision is \( v \leftarrow \min(D_v) \), and if \( d \) is in the higher partition then the decision is \( v \leftarrow \max(D_v) \), otherwise the value is selected randomly between \( \min(D_v) \) and \( \max(D_v) \). Lastly, if the variable was never assigned before then HCSP chooses the smallest value, i.e., \( \min(D_v) \).

Experimentation with other heuristics showed that there is an advantage in selecting \( \min(D_v) \) or \( \max(D_v) \) over other values. For example, if \( D_v = [1,1000] \) and the decision \( v \leftarrow 500 \) leads to a conflict then conflict analysis may conclude that \( v \neq 500 \). This conclusion will create a hole in the domain such that \( D_v = [1,499] \cup [501,1000] \). Decision that assign \( v \) values which are not on the boundaries of \( D_v \) create more holes in \( D_v \). Domains with holes take more memory, possibly exhausting memory, and take more time to propagate. The added costs do not seem to be worth-while even in cases when the number of backtracks is lowered.

### 4.3 HCSP architecture

#### 4.3.1 HCSP Domains

Holding domains efficiently is not trivial. Using C++’s `std::set<int>` will work reasonably well for small domains, but will be impractical for domains with hundreds of elements or more. There are many known alternatives for representing domains which are out of scope of this document. Instead, I will outline the way HCSP represents domains.

In HCSP there are two alternative representations for domains, `hcsp::big_iset` and `hcsp::small_iset`. Currently, it is decided at compilation time with which domain type to build HCSP. The representations are:

- **small_iset** represents sets restricted to the range \([0,31]\). The set is implemented as 32 bit values, where bit \( b_i = 1 \) means that value \( i \) is in the set.

- **big_iset** is restricted to the range \([-2^{31}, 2^{31})\). The set is implemented as a collection of non-overlapping intervals. For example, \( D = [1,6] \cup [10,100] \cup [102,110] \). Because this representation is inefficient for the common case of strides, such as \( 1,3,5,\ldots,101 \), strides have a special representations. Every `big_iset` is restricted to one stride, e.g., \( 1,3,5,\ldots,101 \) is represented as \([1,101]\) with skip size=2. For convenience, HCSP uses the Matlab syntax `1:2:101` to indicate the set \([1,3,5,\ldots,101]\).
Currently, there is no compact representation for sets such as 1:3:101 ∪ 102:105:120 since it involves two different strides. This set is represented as an explicit list of 58 elements rather than two stride ranges. It will be a good project to rewrite big_iset as a specialized implementation of a balanced search tree rather than the current reliance on std::map which is slow and inflexible.

For the future it is a good idea to either get rid of small_iset or to make it work in combination with big_iset.

4.3.2 HCSP Constraints

Constraints are the most extensible interface of HCSP. New constraints can be written by extending the hcsp::constraint_propagator base-class without touching any other part of HCSP. The interface of constraint_propagator has many aspects and parts, most of which have reasonable defaults. There are only four methods that must be defined for new constraints:

- **propagate(const constraint_runtime_info &info):**
  This method defines how the constraint should be propagated. The propagator uses the info object to access and update variable domains. The only way propagator can access CSP variables is through info. The propagator does not know if info refers to CSP variables or to, e.g., mock variables used for testing the propagator. Note that this method may maintain internal data structure and modify it during propagation. For example, it may maintain two-watch literals [MMZ+01b] as in SAT.

- **conflicts(const constraint_runtime_info &info) const:**
  This method detects if propagate would detect a conflict on the current domains, and does this without modifying neither the domains nor the internal state of the propagator.

- **constraint_check(const constraint_runtime_info &info) const:**
  Assuming that info (there is a short description of info in propagate() above) contains fully assigned domains, tell if this assignment satisfies the constraint. This method is used mostly for the automated testing of propagate().

- **clone() const:**
  Create an identical copy of the current propagator.

There are many other methods with reasonable defaults, which may be overridden. There are two reasons to override the default methods: (1) to overcome the poorer
complexity of the defaults; (2) to implement features such as combine() for conflict analysis.

4.3.3 Decomposing the CSP into basic HCSP constraints

HCSP decomposes complex constraints into basic constraints through reification [COC97], a process which is similar to Tseitin transformation in SAT. A reified constraint has the form

\[ c(\bar{y}) \leftrightarrow x = 1 \]

where \( \bar{y} \) is a set of variables which \( x \) is not one of them, and \( c(\bar{y}) \) is a constraint over these variables.

In most cases HCSP prefers one-sided reification, akin to one-sided Tseitin, which has proven to make HCSP work faster. This transformation has the form \( l \lor c(\bar{y}) \), where \( l \) is a literal such as \( x = 0 \) or \( x = 1 \). The transformation decomposes the disjunction of two constraints \( (x_1 \geq x_2 \lor x_3 \leq x_4 + x_5) \) into three individual constraints:

\[
D_{y_1}, D_{y_2} \in \{0, 1\} \\
c_1 \doteq (y_1 = 0 \lor x_1 \geq x_2) \\
c_2 \doteq (y_2 = 0 \lor x_3 \leq x_4 + x_5) \\
c_3 \doteq (y_1 = 1 \lor y_2 = 1)
\]

where \( D_{y_1} \) and \( D_{y_2} \) are the domains of the auxiliary variables \( y_1 \) and \( y_2 \).

Besides reification, HCSP decomposes big arithmetic expressions into their basic operations. Consider for example:

\[ x_1 \leq \text{abs}(x_2) + \max(x_3, x_4) \]

This constraint is broken up into three basic constraint while introducing auxiliary variables \( y_1 \), \( y_2 \), and \( y_3 \):

\[
y_1 \leq \text{abs}(x_2) \\
y_2 \leq \max(x_3, x_4) \\
y_3 \leq y_1 + y_2 \\
y_3 - x_1 \geq 0
\]

This decomposition is slightly wasteful since it could have used \( x_3 \) instead of introducing \( y_3 \). This waste does not seem to impact HCSP performance due to the solving process, and it simplifies common sub-expression elimination.
This is unlike the standard conversion from MiniZinc to FlatZinc, which is to create constraints with ‘=’ over ‘≤’. Also, the FlatZinc specification\(^1\) has, among others, only the equivalent of \(\max(a, b) = c\) and does not have \(\max(a, b) ≥ c\). For the example above, the FlatZinc model will look like:

\[
\begin{align*}
    z_1 &= \text{abs}(x_2) \land \\
    z_2 &= \max(x_3, x_4) \land \\
    z_3 &= z_1 + z_2 \land \\
    x_1 &≤ z_3 .
\end{align*}
\]

The implication of the difference between FlatZinc and HCSP in this regard is that HCSP decomposition seemingly enlarges the search space since the auxiliary variables \(y_i\) are less constrained. It turns out that tighter decomposition hurts performance more than it helps due to: (1) processing ‘≤’ is much faster than ‘=’, and (2) the way HCSP analyzes conflicts. In effect, conflict analysis can get rid of many auxiliary variables constrained by ‘≤’ , such as \(y_1\), which are usually used as pivots in Combine or Resolve rules. On the other hand, with ‘=’ it is more difficult for the conflict analysis to eliminate the more constrained auxiliary variables \(z_i\).

### 4.3.4 Constraint watches

Constraint watches have a long history in SAT and CSP. The Chaff [MMZ+01b] SAT solver used a 2-watched literals scheme to make a major advancement in SAT solving. Similarly, a variety of watches has been defined in [SS04] and later used in Minion [GJM06] which had a significant performance improvement for constraint propagation.

A watch is an algorithmic and data-structure entity that connects a variable and a constraint and waits for a specific event. Consider for example the constraint \(x ≠ y\). If the domains are \(D_x = \{1, 2, \ldots, 5\}\), \(D_y = \{1, 2, \ldots, 5\}\) then \(x ≠ y\) is arc-consistent. It will remain arc-consistent for \(D'_x = \{1, 2, 3, 4\}\), \(D''_x = \{2, 3, 4\}\) , or even \(D'''_x = \{3, 4\}\). This constraint may stop being consistent only when \(|D_x| = 1\) or \(|D_y| = 1\) and not before. It makes no sense to visit \(x ≠ y\) before one of the variables is fully assigned. A FINALIZATION_WATCH watch over \(x\) will visit \(x ≠ y\) only when \(x\) is fully assigned, and similarly for a watch over \(y\).

This is not a big win for a single constraint like \(x ≠ y\), but if there are many \(≠\) constraints connected to \(x\) then none of them would be revised before \(x\) is fully assigned.

\(^1\)This example uses a more compact syntax than the verbose FlatZinc syntax, but the semantics is similar.
For that, HCSP groups similar watches together so that an event is tested only once. This way if there are \( n \) constraints \( x \neq y_1, x \neq y_2, \ldots, x \neq y_n \), then it costs only \( O(1) \) to maintain them when \( |D_x| > 1 \).

HCSP supports watches similar to Minion:

- **FINALIZATION_WATCH**, as discussed in the example, visits the constraint when the variable is fully assigned. (e.g. in \( x \neq y \))

- **LOWER_BOUND_WATCH** visits the constraint when domain reduction changed the lower bound of the domain. (e.g. for \( x \) in \( x \leq y \) after the initial propagation only changes to \( D_x \) that increase \( \min(D_x) \) may cause \( x \leq y \) to reduce \( D_y \)).

- **UPPER_BOUND_WATCH** visits the constraint when domain reduction changed the upper bound of the domain. This is the opposite of **LOWER_BOUND_WATCH**. (e.g. for \( y \) in \( x \leq y \))

- **MODIFICATION_WATCH** visits the constraint every time the domain of the variable is reduced. This is the equivalent of the default behavior for solvers that do not support watches.

- **VALUE_WATCH**(\( t \)), for variable \( v \), visits the constraint when value \( t \) is removed from domain \( D_v \). A Boolean clause in HCSP would implement 2-watch literals by setting two value watches on two of its variables, waiting for either 0 or 1 to be removed from the \( \{0,1\} \) domain.

### 4.4 Solving optimization problems

#### 4.4.1 Introduction to optimization

HCSP supports both minimization and maximization problems, but for simplicity we will describe only minimization. The solving algorithm for maximization is not much different.

A minimization problem has, on top of the constraints, a requirement to find a solution that minimizes an objective function \( O(v_1, \ldots, v_m) \). HCSP effectively minimizes the solution by minimizing the objective variable \( o \) that contains the evaluation of the objective variable \( o = O(v_1, \ldots, v_m) \).

Minimization (maximization) is done in two circles:

- The inner circle is the normal solving process with a specialized decision heuristic.

- The top-level circle reruns the solver with a smaller bound on the objective function.
The solver selects the decision variable using the same heuristic for minimization as for satisﬁability. However, when it chooses the value to assign, it selects the smallest domain value instead of the best performing heuristic. By doing so HCSP strives to return a solution with a small value for the objective without much effort, and without any optimality claims.

**Algorithm 4.3** Optimizing an objective variable \( o \) in HCSP

1: **function** OPTIMIZE\( (C, V, D, o) \)
2:     best ← UNSAT
3:     **while** true **do**
4:         result ← solve\( (C, V, D, o) \)
5:         if result ∈ \{UNSAT, TIMEOUT\} ∧ best = UNSAT then
6:             **return** \( \langle \text{result}, \{\} \rangle \)
7:         **end if**
8:         if result = UNSAT then
9:             **return** \( \langle \text{OPTIMAL}, \text{best} \rangle \)
10: **end if**
11: if result = TIMEOUT then
12:     **return** \( \langle \text{SUBOPTIMAL}, \text{best} \rangle \)
13: **end if**
14: value ← result\[o\] \( \triangleright o \text{ is part of the result} \)
15: best ← result
16: if minimizing **then**
17:     \( D_o \leftarrow D_o \cap (−\infty, \text{value} − 1] \) \( \triangleright D \text{ was not affected by ‘solve’} \)
18: **else**
19:     \( D_o \leftarrow D_o \cap [\text{value} + 1, \infty) \)
20: **end if**
21: if \( D_o = \emptyset \) then
22:     **return** \( \langle \text{OPTIMAL}, \text{best} \rangle \)
23: **end if**
24: **end while**
25: **end function**

The top level algorithm shown in Algorithm 4.3, optimizes \( o \) by repeatedly solving the CSP with smaller domains of \( o \) until the problem is no longer satisfiable. The last solution, before \( D_o \) is too small to be satisfiable, is the optimal one. Because HCSP only reduces the domain of \( o \) and does not assign a single value, some iterations will produce an assignment which is signiﬁcantly better than the previous assignment. This, combined with the decision heuristic, lets HCSP make big steps towards the optimal value.

Successive iterations learn from the previous iterations:

- Conflict constraints are retained between iterations, helping to avoid repeating similar conﬂicts.
• Previous assignments to variables, i.e., *phase saving* [Sht00], such that the next iteration reaches a similar solution relatively fast.

• Counters used for decision heuristic are retained, so that previous conflicts affect decision making.

Experiments have shown that it may be beneficial to tune the decision heuristic for variables other than the optimization variable \( o \). If the optimization function \( \mathcal{O}(v_1, \ldots, v_m) \) is ascending\(^2\) with respect to \( v_1, \ldots, v_m \) then it is sometimes beneficial to prefer the smallest values of \( v_1, \ldots, v_m \). Although promising, this direction has not been pursued and should be investigated in the future.

### 4.5 Generating an interpolant

McMillan [McM03] introduced an interpolating theorem prover. His technique made it possible to prove many types of formulas by using a SAT solver. It may be beneficial to extend this technique such that CSP would be used for instead of SAT. Consider an unsatisfiable Boolean CNF which can be split into two conjuncts \( \psi \land \varphi \). According to [Cra57] it is possible to find a formula, an interpolant, \( I \) such that:

- \( \text{vars}(I) \subseteq \text{vars}(\psi) \cap \text{vars}(\varphi) \)
- \( \psi \vdash I \)
- \( I \land \varphi \vdash \bot \)

Algorithm 4.4, which is implemented in HCSP, can generate an interpolant for CSP using a technique similar to McMillan’s. It is adopted from [McM05] where it is presented for Boolean-SAT, to the world of Signed-CNF SAT, without modifications. The algorithm and all the proofs in [McM05] are taken as is, where instead of Boolean-literals it uses signed-literals.

Algorithm 4.4 accepts \( G \) for the unsatisfiability of the CSP. Currently, the algorithm can work only on explanation-based proofs\(^3\). The proof \( G \) is a Directed Acyclic Graph (DAG) whose roots are constraints in \( \mathcal{C} \) and the leaf node is \( \bot \). For an explanation-based proof, if a root \( c_1 \) is a non-clausal constraint then every one of its outgoing edges is connected to singed-clause explanation, e.g., \( c_2 \). Every such explanation has only one incoming edge. Other than that, every inner node has two incoming edges which represent the binary-resolution that derives this node.

\(^2\) The notion of ascending objective function can be extended to monotone functions, but it was left out because it complicates the description.

\(^3\) Future work: it would be interesting to lift this restriction and work with the more general Combine bases proofs.
The generation of an interpolant in HCSP

```plaintext
Algorithm 4.4

1: function INTERPOLATE(ψ, φ, G)
2:     ψ' ← ψ, φ' ← φ, and G' ← G.
3:  for all r ∈ G s.t. r is a non-clausal constraint do
4:     X ← {e_i | (r, e_i) ∈ G'}  \[X gets the set of clauses adjacent to r]
5:     Ω ← \(\bigwedge_{e_i \in X} e_i\)
6:     ψ' ← ψ' with r replaced by Ω.
7:     φ' ← φ' with r replaced by Ω.
8:     G' ← G' \ {r} \ {(r, e_i) | (r, e_i) ∈ G'}
9:  end for
10: return INTERPOLATE-SIGNED-CLAUSE (ψ', φ', G')
11: end function
```

Theorem 4.5.1 (INTERPOLATE is correct) We assume that φ and φ are conjunctions of non-overlapping sets of constraints.

Given G, an explanation-based proof of \(ψ \land φ \models ⊥\), Algorithm 4.4 produces a valid interpolant for sub-problems \(ψ\) and φ.

Lemma 4.5.1 Consider G, the explanation-based proof for \(ψ \land φ \models ⊥\), a non-clausal constraint in G depicted as r, and \((r, e_1), (r, e_2), \ldots, (r, e_n)\) the outgoing edges of r in G. Define \(G'\) as a sub-graph of G resulting from the removal of r and its outgoing edges. Define \(ψ', φ'\) as rewrites of \(ψ, φ\) where the occurrences\(^4\) of r, if any, are replaced with \(e_1 \land e_2 \land \cdots \land e_n\). Then \(ψ' \land φ' \models ⊥\) and \(G'\) is a proof for this expression.

Proof To show that \(G'\) is a valid proof for \(ψ' \land φ' \models ⊥\) we traverse the proof from ⊥ backwards. With G this traversal would stop at a set of roots \(r_1, r_2, \ldots, r_k\) which are all variables of \(ψ \land φ\). Without loss of generality, assume that the root removed from \(G'\) was \(r_1 = r\). Traversing back on \(G'\) from ⊥ would then give \(r_2, \ldots, r_k\) and a subset of \(e_1, e_2, \ldots, e_n\) instead of \(r_1\), because \(e_1, \ldots, e_n\) are the only successors to \(r_1\).

Because \(r_2, \ldots, r_k\) are part of \(ψ \land φ\) and were not removed from \(ψ' \land φ'\) then they are also part of \(ψ' \land φ'\). Since \(ψ' \land φ'\) is created by replacing \(r_1\) with \(e_1 \land e_2 \land \cdots \land e_n\) then \(e_1, e_2, \ldots, e_n\) are also in \(ψ' \land φ'\).

Since traversing the application of rules backwards from ⊥ to the roots stops at variables of \(ψ' \land φ'\) then it means that applying the rules on the variables of \(ψ' \land φ'\) will reach the leaf node ⊥. This shows that \(G'\) is a valid proof for \(ψ' \land φ' \models ⊥\). □

Proof [Proof of Theorem 4.5.1] By induction on the number of non-clausal constraints in G. Clearly if all constraints are clauses then this algorithm works correctly. Assuming

\(^4\)Since \(ψ, φ\) are a partitioning of G, the constraint r is either in \(ψ\) or in φ.

75
that it works correctly for \( n \) non-clausal constraints in \( G \) we need to show that it works correctly also for \( n + 1 \) non-clausal constraints.

Consider \( G \) with \( n + 1 \) non-clausal constraints. The first iteration of the loop takes non-clausal constraint \( r \), and removes it from \( G' \) and replaces it with \( e_1 \land e_2 \land \cdots \land e_k \) in \( \psi \land \varphi \). According to Lemma 4.5.1 we know that the resulting \( G', \psi', \) and \( \varphi' \) maintain the invariant that \( G' \) is a proof for \( \psi' \land \varphi' \models \bot \).

According to the assumption that the algorithm works for \( n \) non-clausal constraints in \( G \) we know that it will produce a valid interpolant \( I \) for \( \psi' \) and \( \varphi' \) as set by the first iteration. This means that \( \psi' \models I \) and \( I \land \varphi' \models \bot \). We are left to show that \( I \) is also an interpolant for the original \( G, \psi, \varphi \).

If \( \psi \) had \( r \) then it can be written as \( \psi = (\bar{\psi} \land r) \) where \( \bar{\psi} \) represents \( \psi \) with \( r \) removed. In this case \( \psi' = (\bar{\psi} \land \Omega) \). To check the relation between \( \psi \) and \( \psi' \), look at the relation between \( r \) and \( \Omega \). Since all \( e_i \) are explanations to \( r \) we know that \( \forall e_i \in X. [r \models e_i] \), i.e., \( r \models \Omega \). This means that \( (\bar{\psi} \land r) \models (\bar{\psi} \land \Omega) \), i.e., \( \psi \models \psi' \) when \( r \) is in \( \psi \). If \( r \) is not in \( \psi \) then \( \psi' = \psi \) and trivially \( \psi \models \psi' \). It can be shown, similarly, that \( \varphi \models \varphi' \).

The following shows that \( I \) satisfies all three interpolation requirements for \( \psi \) and \( \varphi \). Since \( \psi \models \psi' \) and, according to the induction assumption, \( \psi' \models I \) then \( \psi \models I \). Since \( \varphi \models \varphi' \) then \( \varphi \land I \models \varphi' \land I \) and since, according to the induction assumption, \( \varphi' \land I \models \bot \) it turns out that \( \varphi \land I \models \bot \).

According to the induction assumption we know that \( I \) refers to variables shared by \( \psi' \) and \( \varphi' \). If \( \psi' = \bar{\psi} \land \Omega \) then \( \text{vars}(\psi') = \text{vars}(\bar{\psi}) \cup \text{vars}(e_1) \cup \cdots \cup \text{vars}(e_n) \). Since \( e_i \) are explanations to \( r \) then \( \text{vars}(e_i) \subseteq \text{vars}(r) \) and hence \( \text{vars}(\psi') \subseteq \text{vars}(\bar{\psi}) \cup \text{vars}(r) \). And since, according to the induction assumption, \( \text{vars}(I) \subseteq \text{var}(\psi') \) then \( \text{vars}(I) \subseteq \text{vars}(\bar{\psi}) \cup \text{vars}(r) \). According to the definition of \( \bar{\psi} \) this simplifies to \( \text{vars}(I) \subseteq \text{vars}(\psi) \). Using the same technique it is easy to show that \( \text{vars}(I) \subseteq \text{vars}(\varphi) \).

This means that \( I \) meets all three requirements for interpolants over \( n + 1 \) non-clausal constraints in \( G \). This proves by induction that the algorithm generates the correct \( I \) for any number of non-clausal constraints. \( \square \)
Appendix
Chapter A

Constraints which HCSP supports

\footnote{Constraint propagation and the difference between precise and imprecise propagation is defined in Sect. 1.3.1}
<table>
<thead>
<tr>
<th>constraint</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_0 = x_1$</td>
<td></td>
</tr>
<tr>
<td>$x_0 = -x_1$</td>
<td></td>
</tr>
<tr>
<td>$x_0 \circ \text{abs}(x_1)$</td>
<td></td>
</tr>
<tr>
<td>$x_0 = x_1 + x_2 + \ldots$</td>
<td>2-s complement wrap on overflow</td>
</tr>
<tr>
<td>$x_0 \leq x_1 + x_2 + \ldots$</td>
<td></td>
</tr>
<tr>
<td>$x_0 \leq x_1 \cdot x_2$</td>
<td></td>
</tr>
<tr>
<td>$x_0 \geq x_1 \cdot x_2,$</td>
<td></td>
</tr>
<tr>
<td>$x_0 = \min(x_1, x_2, \ldots)$</td>
<td></td>
</tr>
<tr>
<td>$x_0 = \max(x_1, x_2, \ldots)$</td>
<td></td>
</tr>
<tr>
<td>$(b_0 \land x_0 = x_2) \lor (\neg b_0 \land x_0 = x_3)$</td>
<td>In C style this is $x_0 = (b_0?x_2:x_3)$</td>
</tr>
<tr>
<td>$x \neq 0 \leftrightarrow y = z$</td>
<td></td>
</tr>
<tr>
<td>$x_0 \in \text{Set} \leftrightarrow x_1 &lt; x_2$</td>
<td></td>
</tr>
<tr>
<td>$x_0 - x_1 \geq a_0$</td>
<td></td>
</tr>
<tr>
<td>$(x_0 + a_0 \leq x_1 \lor x_1 + a_1 \geq x_0)$</td>
<td></td>
</tr>
<tr>
<td>$x_0 \neq x_1$</td>
<td></td>
</tr>
<tr>
<td>AllDifferent($x_0, x_1, \ldots$)</td>
<td></td>
</tr>
<tr>
<td>$[x_0, x_1 + a_0] \subseteq [a_1, a_2]$</td>
<td></td>
</tr>
<tr>
<td>$[x_0, x_1 + a_0] \not\subseteq [a_1, a_2]$</td>
<td></td>
</tr>
<tr>
<td>$a_0 \cdot x_0 + a_1 \cdot x_1 + \ldots \geq a_k$</td>
<td>Also named \textit{weighted-sum} in [BCR05]</td>
</tr>
<tr>
<td>$(x_1 \in \text{Set}_1 \lor x_2 \in \text{Set}_2 \lor \cdots)$</td>
<td>Signed-clause</td>
</tr>
<tr>
<td>$c_1(x_1, \ldots, x_n) \lor c_2(x_{n+1}, \ldots, x_m) \lor \cdots$</td>
<td>A disjunction of any of the above.</td>
</tr>
<tr>
<td>$x_0 = x_1 \cdot x_2$</td>
<td>Imprecise constraint propagation $^1$</td>
</tr>
<tr>
<td>$x_0 = x_1 / x_2$</td>
<td>Imprecise constraint propagation.</td>
</tr>
<tr>
<td>$x_0 = x_1 % x_2$</td>
<td>Imprecise constraint propagation.</td>
</tr>
<tr>
<td>$x_0 = x_1^{x_2}, \text{i.e.,} \ x_0 = \text{pow}(x_1, x_2)$</td>
<td>Imprecise constraint propagation.</td>
</tr>
</tbody>
</table>

Table A.1: Constraints supported by HCSP, where $x_i$ denote variables, $b_i$ Boolean variables, $a_i$ constants, and $\circ \in \{=, \leq, \geq\}$.  

---  

$^1$ Also named \textit{weighted-sum} in [BCR05].
Chapter B

A Proof-Producing CSP Solver
(A proof supplement)\(^1\)

B.1 Introduction to the Proof Supplement

This report is meant to be a supplemental document to [VS10a]. It contains a mostly unstructured collection of things that were omitted from [VS10a] mostly due to lack of space. It includes proofs for inference rules and algorithms, introduces more inference rules, and, finally, it presents missing algorithms and improvements to algorithms that were presented in [VS10a].

B.2 Inference rules

In this section we prove the soundness of the inference rules that were introduced in Table 3 of [VS10a], and prove that it is possible to match an inference rule for any possible constraint propagation. The latter established the completeness of CSP-ANALYZE-CONFLICT.

B.2.1 Soundness proofs for the inference rules

Here we prove the soundness of the inference rules from Table 3 in [VS10a]. Throughout this section we assume that all constraints and inference rules refer to integer variables. This assumption is used only for convenience and is not a fundamental part of the work, which can be easily extended to reals.

\(^1\)Published as a technical report in [VS10b].
Lemma B.2.1 For an integer constant $m \in \mathbb{Z}$ and integer variables $a, b \in \mathbb{Z}$

\[
(a \leq b) \\
(a \in (-\infty, m] \lor b \in [m+1, \infty)) \\
(LE(m)).
\]

Proof Combining the premise $a \leq b$ with the tautology $a \leq m \lor a > m$ yields $a \leq m \lor b > m$. Converting this to a signed clause gives the consequent of the rule $(a \in (-\infty, m] \lor b \in [m+1, \infty))$. □

Lemma B.2.2 Assuming $V \subseteq \{v_1, \ldots, v_k\}$ such that $|V| = 1 + |D|

\[
\frac{\text{All-diff}(v_1, \ldots, v_k)}{(\bigvee_{v \in V} v \notin D)} (AD(D,V))
\]

Proof Given the premise All-diff($v_1$, $\ldots$, $v_k$), for $|D| = 0$, $|V| = 1$ the consequent is a tautology. Otherwise, due to counting considerations, there is no feasible assignment of $|D|$ different values to $|V| = |D| + 1$ variables, or formally $\neg(\bigwedge_{v \in V} v \in D)$. After pushing the negation into the expression, this gives $(\bigvee_{v \in V} v \notin D)$. □

Lemma B.2.3 For an integer constant $m \in \mathbb{Z}$ and integer variables $a, b \in \mathbb{Z}$

\[
a \neq b \\
(a \neq m \lor b \neq m) \\
(NE(m)).
\]

Proof Constraining the tautology $a \neq m \lor a = m$ by the premise $a \neq b$ results with $a \neq m \lor (a = m \land a \neq b)$ which implies $a \neq m \lor m \neq b$. Converting this to a signed clause gives the consequent of the rule $(a \neq m \lor b \neq m)$. □

Lemma B.2.4 For an arbitrary set of values $D$

\[
a = b \\
(a \notin D \lor b \in D) \\
(Eq(D)).
\]

Proof Constraining the tautology $a \notin D \lor a \in D$ by the premise $a = b$ results with $a \notin D \lor (a \in D \land a = b)$ which implies the consequent of the rule $(a \notin D \lor b \in D)$. □

Lemma B.2.5 For integer constants $m, n \in \mathbb{Z}$ and integer variables $a, b, c \in \mathbb{Z}$

\[
a \leq b + c \\
(a \in (-\infty, m+n] \lor b \in [m+1, \infty) \lor c \in [n+1, \infty)) \\
(LE_+(m, n)).
\]
Proof  Adding the premise \( a \leq b + c \) to the right hand side of the tautology
\[
\neg(b \leq m \land c \leq n) \lor (b \leq m \land c \leq n)
\]
gives \( \neg(b \leq m \land c \leq n) \lor a \leq m + n \). Pushing negation all the way to the literals yields \( b > m \lor c > n \lor a \leq m + n \), which is equivalent to the consequent clause \( (a \in (-\infty, m + n] \lor b \in [m + 1, \infty) \lor c \in [n + 1, \infty)) \).

\[\text{Lemma B.2.6} \quad \text{For integer constants } l_b, u_b, l_c, u_c \in \mathbb{Z} \text{ and integer variables } a, b, c \in \mathbb{Z}\]
\[
(a \in [l_b + l_c, u_b + u_c] \lor b \not\in [l_b, u_b] \lor c \not\in [l_c, u_c]) \quad (EQ^a_l(l_b, u_b, l_c, u_c))
\]

Proof  The tautology \( \neg(l_b \leq b \leq u_b \land c \leq c \leq u_c) \lor (l_b \leq b \leq u_b \land l_c \leq c \leq u_c) \) can be rewritten as \( \neg(b \in [l_b, u_b] \land c \in [l_c, u_c]) \lor (l_b \leq b \leq u_b \land l_c \leq c \leq u_c) \), which, after combining with the premise \( a = b + c \), becomes
\[
\neg(b \in [l_b, u_b] \land c \in [l_c, u_c]) \lor (l_b + l_c \leq a \leq u_b + u_c).
\]
Writing comparisons as signed literals yields
\[
\neg(b \in [l_b, u_b] \land c \in [l_c, u_c]) \lor a \in [l_b + l_c, u_b + u_c].
\]
Pushing negation all the way to the literals, and rewriting as a clause yields the consequent:
\[
(b \not\in [l_b, u_b] \lor c \not\in [l_c, u_c]) \lor a \in [l_b + l_c, u_b + u_c].
\]

\[\text{Lemma B.2.7} \quad \text{For integer constants } l_a, l_b, m, n \in \mathbb{Z} \text{ and integer variables } a, b \in \mathbb{Z}\]
\[
\neg(\text{Overlap}(a, l_a, b, l_b)) \quad (\text{NO}(m,n))
\]

Proof  The semantics of the premise \( \text{NoOverlap}(a, l_a, b, l_b) \) is
\[
\phi = (a \geq b + l_b \lor b \geq a + l_a),
\]
which can be written as \( \phi = (b + l_b \leq a \lor a \leq b - l_a) \). Combining \( \phi \) with the tautology
\[
\neg(m \leq a \leq n + l_b - 1) \lor (m \leq a \leq n + l_b - 1)\]
yields
\[
\neg(m \leq a \leq n + l_b - 1) \lor b + l_b \leq n + l_b - 1 \lor m \leq b - l_a.
\]
After simple rewriting this becomes
\[
\neg(m \leq a \leq n + l_b - 1) \lor b \leq n - b \lor a \geq m + l_a),
\]
and then \( \neg(m \leq a \leq n + l_b - 1) \lor \neg(b > n - 1 \land b < m + l_a) \). Rewriting with literal notation gives us the consequent of the rule:
\[
(a \not\in [m, n + l_b - 1] \lor b \not\in [n, m + l_a - 1])
\]
B.2.2 Completeness of inference rules

For CSP-ANALYZE-CONFLICT to be complete, it is mandatory to be able to create an explanation for any type of constraint, including for explicit table relations. When an unsatisfiability proof is printed, the proof derives each explanation from a constraint using an inference rule. For the proof system to be complete, we must be able to match an inference rule for any implication. Like in the proof, CSP-ANALYZE-CONFLICT may also use inference rules to derive explanations.

In order to discuss inference rules and their consequent explanations, we need to focus on valid implications first. The following definition summarizes what we refer to as a propagator. These are weak requirements that are satisfied by any propagator that we are aware of.

**Definition B.2.1 (Constraint propagator assumption)** Every constraint \( r \in C \) is associated with a propagator by the same name. We assume that the constraint propagator \( r(v_1, \ldots, v_k) \) meets the following criteria:

- when a complete assignment to \( v_1, \ldots, v_k \) violates the constraint \( r(v_1, \ldots, v_k) \) the propagator will recognize it;
- if it modifies a domain from \( D_i \) to \( D'_i \) then \( D'_i \subseteq D_i \) and the values in \( D_i \setminus D'_i \) are not supported by \( r \) (in other words, for every assignment \( v_i \leftarrow x_i \) such that \( x_i \in D_i \setminus D'_i \), there is no assignment to \( v_1, \ldots, v_k \) from their respective domains that satisfies the constraint \( r \)); and
- it terminates in finite time.

Note that this assumption leaves the freedom to choose the strength of the propagator. Some known strategies are arc consistency, which is the strongest, bounds consistency, and pure backtrack-search algorithm, which is the weakest.

Recall that CSP-ANALYZE-CONFLICT requires an explanation of inferred literals, and hence we need to show that such an explanation can always be given. For this purpose we define a generic inference rule and then prove that it is applicable to all constraints.

**Definition B.2.2 (The generic inference rule)** Consider \( r(v_1, \ldots, v_k) \), an arbitrary constraint, which when combined with literals \( v_1 \in D_1, \ldots, v_k \in D_k \) implies literal \( v_i \in D'_i \). The generic, parameterized, implication rule \( G_{r,i}(A_1, \ldots, A_k) \), as defined below, generates an explanation for this implication

\[
\frac{r}{(v_1 \in A_1 \lor \cdots \lor v_k \in A_k)} (G_{r,i}(A_1, \ldots, A_k)),
\]
where for each \( j = 1, \ldots, i - 1, i + 1, \ldots k \), we define \( A_j = \overline{D}_j \) and where \( A_i = D'_i \cup \overline{D}_i \).

We now show that Definition B.2.2 gives a valid inference rule for an arbitrary constraint.

**Lemma B.2.8 (Existence of an inference rule)** Let \( u \) be a node in an implication graph that represents a literal that was implied by an arbitrary constraint \( r \). There exists an inference rule that its premise is \( r \) and its consequent is an explanation of \( u \).

**Proof** For any constraint \( r(v_1, \ldots, v_k) \), Definition B.2.2 provides the generic propagation rule \( G_{r,i}(A_1, \ldots, A_k) \), which generates the following explanation for an implication of \( v_i \in D'_i \):

\[
\begin{align*}
    c & = (v_1 \in \overline{D}_1 \lor \cdots \lor v_{i-1} \in \overline{D}_{i-1} \lor v_i \in (D'_i \cup \overline{D}_i) \lor v_{i+1} \in \overline{D}_{i+1} \lor \cdots \lor v_k \in \overline{D}_k),
\end{align*}
\]

which can also be written as:

\[
\begin{align*}
    c & = (v_1 \notin D_1 \lor \cdots \lor v_k \in D_k \lor v_i \in D'_i).
\end{align*}
\]

To prove that the clause \( c \) is indeed an explanation, recall, we have to show that, for \( l_1 = (v_1 \in D_1), \ldots, l_k = (v_k \in D_k) \):

1. \( r \rightarrow c \),
2. \( (l_1 \land \cdots \land l_k \land c) \rightarrow l \).

To prove the first property we will show that every possible assignment \( x_1, \ldots, x_k \) satisfies \( \neg c(x_1, \ldots, x_k) \rightarrow \neg r(x_1, \ldots, x_k) \). The clause \( c \) is falsified only if all its literals are \( \text{false} \), i.e., \( v_1 \in D_1 \land \cdots \land v_k \in D_k \land v_i \notin D'_i \). In other words, all values are taken from their respective domains \( D_j \) and \( x_i \in D_i \setminus D'_i \), which is one of the values removed by the propagator \( r \). According to Definition B.2.1, \( r \) will not remove \( x_i \) from \( D_i \) if \( x_i \in D_i \land \cdots \land x_k \in D_k \) satisfies \( r(x_1, \ldots, x_k) \), implying that \( \neg r(x_1, \ldots, x_k) \). The conclusion is that \( \neg r(x_1, \ldots, x_k) \) whenever \( \neg c(x_1, \ldots, x_k) \).

For the second property we analyze all assignments \( v_1 = x_1 \land \cdots \land v_k = x_k \) which satisfy the left side of the implication. Such an assignment has to satisfy \( x_1 \in D_1 \land \cdots \land x_k \in D_k \), in which case all literals \( v_j \notin D_j \) of \( c \) are falsified except for \( v_i \in D'_i \), i.e., \( l \). This shows that an assignment that satisfies the left-hand side of the implication also satisfies its right-hand side.
B.3 Algorithms

B.3.1 CSP-Analyze-Conflict algorithm, the proof

First, we repeat Algorithm 2.1 which appears in [VS10a]:

1: function CSP-Analyze-Conflict
2:    cl := Explain (conflict-node);
3:    pred := Predecessors (conflict-node);
4:    front := Relevant (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 (Relevant (front ∪ pred, cl));
12:    end while
13:       add-clause-to-database(cl);
14:       return clause-asserting-level(cl);
15: end function

To prove the correctness of Algorithm 2.1, we first prove the loop invariant that is mentioned briefly in [VS10a].

Lemma B.3.1 (CSP-Analyze-Conflict loop invariant) The following invariant holds just before line 6: The clause cl is inconsistent with the labels in front.

Proof Consider the conflicting constraint p, i.e., the constraint that labels the edges leading to conflict-node. On the first iteration, the literals of pred conflict the constraint p. Because cl is the explanation clause of p it also conflicts on the conjunction of pred. Because Relevant at line 4 keeps the nodes which relevant to cl, i.e., share the same variables, then the labels of front also conflict cl.

Assuming that the invariant holds on iteration n − 1, we will show that the invariant holds at the n-th iteration, if executed. For this proof we denote by cl and cl’ the values of cl before and after the update at line 9, respectively, and similarly we use front and front’. Except for the literal with v = var(lit(curr-node)), according to the definition of expl, all other literals are falsified by pred. cl’ has three types of literals: the first is from expl and do not refer to v, the second is from cl and do not refer to v, and the third refers to v and contains a conjunction of literals from the former two sources.
1. The literals from $cl$ are falsified by the conjunction of $lits(front)$ and hence also by the conjunction of $lits(front \cup pred)$.

2. For an implication going from literals $\{l_1, \ldots, l_k\} = lits(pred)$ to literal $l = \text{lit(curr-node)}$, recall, the second requirement from an explanation $c$ is $(l_1 \land \cdots \land l_k \land c) \rightarrow l$. In order for $c$ to be able to imply a literal $l$, as required, all literals of $cl$ must be falsified by $l_1 \land \cdots \land l_k$, except for the literal that constrains $v$. Because $l_1 \land \cdots \land l_k$ is the conjunction of $lits(pred)$, then literals $cl$ that do not constrain $v$, which are of the second type of literals, are also falsified by the conjunction of $lits(front \cup pred)$.

3. We want to prove that the literal of $cl'$ labeled with $v$ is falsified by the $lits(front')$. First we introduce the following naming convention: $\omega_v$ is the disjunction of all literals of clause $\omega$ which affect $v$, and similarly $N_v$ is the conjunction $\bigwedge_{l \in lits(N) \mid \text{var}(l) = v} l$. We also reuse the notation used in the definition of explanations where $l$ is the literal of $\text{curr-node}$ and $l_1, \ldots, l_k$ are the literals of $pred$.

The claim to be proven can now be reformulated as $(cl'_v \land lits(front')) = false$. To prove this, first consider the invariant from iteration $n-1$ which gives $(cl_v \land l) = false$, or as a clause:

$$(-cl_v \lor -l)$$

The definition of explanations states that $(l_1 \land \cdots \land l_k \land expl) \rightarrow l$, or as a clause:

$$(-l_1 \lor \cdots \lor -l_k \lor -\text{expl} \lor l)$$

Resolving the last two clauses with $v$ as pivot results with:

$$(-l_1 \lor \cdots \lor -l_k \lor -\text{expl} \lor -cl_v)$$

which is the same as

$$\neg((l_1 \land \cdots \land l_k \land expl \land cl_v))$$

which shows that $expl \land cl_v$ is falsified by $lits(pred)$, and hence $cl'_v = expl_v \land cl_v$ is also falsified by $lits(front \cup pred)$. The call to RELEVANT at line 11 removes only nodes which are irrelevant to the falsification of $cl'$, hence, it is left to show that $front$, produced by DISTINCT at line 11, also falsifies $cl'_v$. For this part of the proof we need some formalism first. We depict $n_1, \ldots, n_q$ as the nodes of $lits(front \cup pred)$ for which $\text{var}(\text{lit}(n_j)) = v$ according to the propagation order that created them. Using this formalism we can say that DISTINCT will remove $n_1, \ldots, n_{q-1}$ and keep only $n_q$, we need to show that this node is sufficient to falsify $cl'_v$. Because how propagators are allowed to work, literals of succeeding nodes must refer to decreasing domain sized, and hence
n_q \rightarrow n_j \text{ for each } j = 1, \ldots, q. \text{ This result means that all nodes } n_1, \ldots, n_{q-1} \text{ are redundant, such that if } (lit(n_1) \land \cdots \land lit(n_q) \land cl'_v) = false \text{ then it is also true that } (lit(n_q) \land cl'_v) = false. \text{ Because } cl'_v \text{ was falsified by the literals of } n_1, \ldots, n_q, \text{ as shown above, then it must be falsified by } n_q \text{ which is entered into } \text{front}'.

As we seen, all literals of the new clause cl' are falsified by the literals of front'. This shows that iteration n also satisfies the loop invariant. \qed

**Theorem B.3.1 (Algorithm 2.1 correctness)** CSP-Analyze-Conflict, i.e., Algorithm 2.1, is sound and complete:

- **soundness** – the returned clause cl is derived from the CSP φ such that φ \rightarrow cl, and it is either falsified at the target decision level or is an asserting clause that will cause propagation at clause-asserting-level(cl).

- **completeness** – the algorithm terminates and returns cl.

Also, CSP-Analyze-Conflict returns a target level which forces backtrack of at least one level.

Note that this theorem allows the resulting cl to be false even at the target of backtrack. This situation happens often in practice and can be easily eliminated by a small modification to the algorithm, as described in Section B.4.

**Proof**  Soundness – The requirement of φ \rightarrow cl, is trivial because cl is created through a set of proven inference rules from premises and through applications of the resolution rule. The requirement of cl to be an asserting clause is met through the STOP-CRITERION-MET which is true in two cases:

- **front** nodes have no predecessors, i.e., they refer to initial domains. According to the loop invariant (Lemma B.3.1) all literals of cl are falsified by the conjunction of the literals of front, meaning that cl is false even with the initial domains.

- The latest node of front is the only one from decision level d and the next latest node is from decision level d' < d, such that function clause-asserting-level(cl) will return this d'. Backtracking will undo all decisions and implications done from d' + 1 and later a target state at which, we need to show, cl will be an asserting clause. At the target state all nodes of front, except for the latest one, are still present, which according to Lemma B.3.1 must falsify all literals except for the latest one. This means that cl is either falsified or an asserting clause at the end of decision level d'.

88
Completeness – First we show that the preconditions of all auxiliary functions are fulfilled allowing them to successfully terminate. Assuming that CSP-Analyze-Conflict is passed a conflict-node then the first PREDECESSORS must succeed. The first EXPLAIN must succeed because it is possible to generate an explanation for the conflicting constraint (Lemma B.2.8). Inside the loop ¬STOP-CRITERION-MET makes sure that curr-node has predecessors making PREDECESSORS succeed. Like with the first EXPLAIN the one in the loop must also find an explanation. All other used algorithms do not have special assumptions or preconditions.

The algorithm must terminate because each iteration removes the latest node, and inserts earlier nodes of pred instead of it, coupled with the fact that the implication graph is a DAG, each iteration is guaranteed to have curr-node which is earlier than at the previous iteration. Because the implication graph is finite then the number of iteration must be finite.

Backtrack level – as shown above, the target level is \( d' \) which is smaller than \( d \) which itself the current decision level, or earlier. This means that the caller will be instructed to backtrack to at least one decision level back. □

B.4 Enhancements and optimizations

In this section we describe several enhancements to the definitions and to the algorithms that may either improve performance or minimize proof size. At the present time we do not have exact measurements of the effectiveness of these modifications. Nevertheless these enhancements have interesting properties, making them worth mentioning.

B.4.1 CSP-Analyze-Conflict node rejuvenation

A problem that the proof of CSP-Analyze-Conflict (Theorem B.3.1) showed is that a conflict clause may be conflicting immediately after backtrack, i.e., with no new decision. This problem is solved in HCSP by the following, trivial, modification to the algorithm.

At lines 4 and 11 DISTINCT is called. After this call we add a call to a new function REJUVENATE (\( cl.front \)), which finds the earliest nodes that still falsify \( cl \). This modification preserves the invariant of the loop because it moves only to such earlier nodes that still falsify \( cl \). The proof of Theorem B.3.1 with the amended algorithm is mostly unchanged since it relies on the invariant, which is not affected by the modification.

This rejuvenating can have a positive impact on conflict clause and proof sizes as it allows CSP-Analyze-Conflict to ignore all implications done between the original node and the rejuvenated node. However, it may also have a negative effect since by
ignoring an implication we may lose a good candidate for clause resolution that would erase the literal from $cl$ and reach a UIP. Instead, by ignoring a good candidate for resolution, we will have to apply resolution many times before a UIP is reached.

B.4.2 Augmented explanations

Let $(l_1, l), \ldots, (l_n, l)$ be the incoming edges of a node $u$ such that $lit(u) = l$. If $c$ is an explanation clause of $u$, recall, then $(l_1 \land \cdots \land l_n \land c) \rightarrow l$. We now propose to weaken this requirement.

**Definition B.4.1 (augmented explanations)** Let $u$ be a node in the implication graph such that $lit(u) = l$. Let $(l_1, l) \ldots (l_n, l)$ be the incoming edges of $u$, all of which are labeled with a constraint $r$. Let $l'$ be the literal in the clause $cl$ just before the resolution step in CSP-Analyze-Conflict, such that $\text{var}(l) = \text{var}(l')$. A signed clause $c'$ is an augmented explanation clause of a node $u$ and a clause $cl$ if it satisfies:

1. $r \rightarrow c'$,
2. $(l_1 \land \cdots \land l_n \land c') \rightarrow \neg l'$.

This definition lets us find $c'$ which is not an explanation clause according to the original definition, but which is sufficient for our purpose, since while it is still implied by the original constraint $r$, it also holds that

$$\text{Resolve}(cl, c', \text{var}(l)) \rightarrow \text{Resolve}(cl, c, \text{var}(l)).$$

In other words, we derive a stronger resolvent. This may lead to shorter proofs down the line.

**Example B.4.1** The literals $l_1 = (b = 3)$, $l_2 = (a \in [1, 5])$ together with the constraint $r = (a \leq b)$ imply $l = (a \in [1, 3])$. This implication is depicted in the following small diagram:

\[
\begin{array}{c}
\text{b} = 3 \xrightarrow{a \leq b} \text{a} \in [1, 3] \\
\text{a} \in [1, 5] \xrightarrow{a \leq b} \end{array}
\]

The only valid explanation clause is derived using $LE(3)$:

$$c = (a \in (-\infty, 3] \lor b \in [4, \infty)).$$

Indeed $(l_1 \land l_2 \land c) \rightarrow l$. 

90
Now suppose that \( cl = (a = 5 \lor c = 1) \) and hence \( l' = (a = 5) \). We need to find \( c' \) such that \((l_1 \land l_2 \land c') \rightarrow \neg l'\). Although \( c \) is an explanation clause, we can get a stronger such clause with \( LE(4) \):

\[
c' = (a \in (-\infty, 4] \lor b \in [5, \infty)),
\]

which is sufficient. Resolution of \( cl \) with \( c' \) results in \((b \in [5, \infty) \lor c = 1)\) whereas resolving with \( c \) would result in the weaker clause \((b \in [4, \infty) \lor c = 1)\).

What happens in this example is that \( c' \) does not have to eliminate the value 4 from the domain of \( a \) because it is allowed by \( \neg l' \). As a result the other literal, referring to \( b \), becomes stronger. \( \Box \)

### B.4.3 Lookahead explanations

The weaker definition of \textit{augmented explanations} gives us more freedom to choose a clause from a bigger set of possible clauses. Some freedom is also present in regular explanation clauses, especially when explaining a conflicting constraint. For example, the constraint \( a \leq b \) is conflicting for domains \( D_a = [10, 20] \) and \( D_b = [0, 8] \), for which both \( c = (a \in (-\infty, 8] \lor b \in [9, \infty)) \) and \( c = (a \in (-\infty, 9] \lor b \in [10, \infty)) \) are valid explanations.

We have discovered that using this freedom wisely can considerably shorten both proof size and run-time. If \textit{curr-node} is associated with variable \( v \), we simply prefer an explanation which has a stronger literal associated with the rightmost node of \textit{predecessors} which is not associated with \( v \). The effect of this preference is twofold:

1. At the next iteration of CSP-\textsc{Analyze-Conflict}, there is a bigger chance that this literal \( l' \) will be a pivot than any other literal of the explanation clause. Making this literal smaller may strengthen the next augmented explanation because it will weaken the right-hand side of the implication in the requirement of \textit{augmented explanations}:

\[
(l_1 \land \cdots \land l_n \land c') \rightarrow \neg l',
\]

which will give more freedom for constructing \( c' \).

2. By producing a stronger literal, there is a bigger chance that \textsc{Rejuvenate} will rejuvenate the node of \textit{var}(\( l' \)) to a previous decision level, saving the need for some resolutions and, eventually, facilitate the creation of a smaller conflict clause.
Bibliography


[GMN] IANP GENT, IAN MIGUEL, and PETER NIGHTINGALE. The alldifferent constraint: Exploiting strongly-connected components and other efficiency measures.


96


null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null


Chapter 4

...
Theorem 1.6 (n - 1)!

In order to find a solution to the CNF SAT problem, we can use a SAT solver. However, in practice, using a SAT solver can be computationally expensive, especially for large formulas.

A SAT solver is a program that can be used to determine whether a given propositional formula is satisfiable. If a formula is satisfiable, then the SAT solver will provide a model that satisfies the formula.

The most well-known SAT solvers are DPLL-based solvers, which use a series of heuristics to search for a satisfying assignment. One such heuristic is the choice of the variable to be assigned.

The choice of variable can significantly affect the efficiency of the SAT solver. Therefore, it is important to choose the right variable to assign.

In this section, we will discuss the decision variable selection strategies used in DPLL-based SAT solvers. We will also present a simple example to illustrate these strategies.

Consider the following CNF formula:

\[ (x_1 \lor x_2) \land (x_3 \lor x_4) \land \neg x_1 \land x_3 \]

The decision variable selection strategies used in DPLL-based SAT solvers can be divided into two main categories: random selection and conflict-driven selection.

Random Selection:

In random selection, a variable is chosen at random to be assigned. This strategy is simple and easy to implement, but it can lead to inefficient search.

Conflict-Driven Selection:

In conflict-driven selection, the SAT solver uses information from previous assignments to guide the choice of the next variable to be assigned. This strategy can lead to more efficient search.

In this section, we will discuss how to implement conflict-driven selection in a DPLL-based SAT solver.

For example, consider the following formula:

\[ (x_1 \lor x_2) \land (x_3 \lor x_4) \land \neg x_1 \land x_3 \]

If we assign \( x_1 \) to 1, we will encounter a conflict. Therefore, we need to choose a different variable to assign.

We can use the following strategy to choose the next variable to assign:

1. Choose a variable that has the most clauses.
2. If there is a tie, choose a variable that has the most literals.

In this case, \( x_3 \) has two clauses, while \( x_1 \) and \( x_2 \) have one clause each. Therefore, we will choose \( x_3 \) to be the next variable to assign.

Once we have chosen \( x_3 \) to be the next variable to assign, we will add the following clause to the formula:

\[ \neg x_3 \]

Now, the formula becomes:

\[ (x_1 \lor x_2) \land (x_3 \lor x_4) \land \neg x_1 \land \neg x_3 \land x_3 \]

We can now continue the search for a satisfying assignment.

In practice, DPLL-based SAT solvers use more sophisticated strategies to choose the next variable to assign. These strategies take into account not only the number of clauses and literals, but also other factors such as the structure of the formula and the current search state.

These strategies can significantly improve the efficiency of the SAT solver.

In conclusion, the choice of variable to assign is a crucial factor in the efficiency of a DPLL-based SAT solver. Random selection can lead to inefficient search, while conflict-driven selection can lead to more efficient search.

Therefore, it is important to choose the right variable to assign in order to find a solution to the CNF SAT problem as quickly as possible.
The propagation process of the solution to the constraint satisfaction problem.

Algorithm 1.1 MAC (Maintain Arc Consistency)

Let $D_v = \{1, 2, \ldots, n\}$ be the set of variables and $D_x = D_y = D_z = \{1, 2, 3, 4\}$ be the set of values.

$\text{If } x \neq y \neq z \neq x, D_x = D_y = D_z = \{1, 2\}.$

$x = 1$ and $y = 2$.

$D_v = \{1, 2, \ldots, n\}$ be the set of variables and $D_x = D_y = D_z = \{1, 2, 3, 4\}$ be the set of values.

$\text{If } x \neq y \neq z \neq x, D_x = D_y = D_z = \{1, 2\}.$

$x = 1$ and $y = 2$.

$D_v = \{1, 2, \ldots, n\}$ be the set of variables and $D_x = D_y = D_z = \{1, 2, 3, 4\}$ be the set of values.

$\text{If } x \neq y \neq z \neq x, D_x = D_y = D_z = \{1, 2\}.$

$x = 1$ and $y = 2$.

$D_v = \{1, 2, \ldots, n\}$ be the set of variables and $D_x = D_y = D_z = \{1, 2, 3, 4\}$ be the set of values.

$\text{If } x \neq y \neq z \neq x, D_x = D_y = D_z = \{1, 2\}.$

$x = 1$ and $y = 2$.

$D_v = \{1, 2, \ldots, n\}$ be the set of variables and $D_x = D_y = D_z = \{1, 2, 3, 4\}$ be the set of values.

$\text{If } x \neq y \neq z \neq x, D_x = D_y = D_z = \{1, 2\}.$

$x = 1$ and $y = 2$.
 внויון מוסכמים של 앞-آلילתום מייפיסות שלום, כלכליים נדרשים 60,000 שורות

רמיזה 3awy אל-آلילתום ויקוינוין שנות:

• [RL09] שפורט שוחזרה-b
• [MSKS10] שפורט שוחזרה-
• [IMR+06] חקל מפורט המאפרת בכלי מסוכן פאודר-בוליארג

הائف-آلילתום

ללא התמצאות על קד החפצים מぷ. מבן 3 שפורט אל-آلילתום ויקוינוין שנות:

.4(x=3)
.5(y=1)
.6.
.7.
.8.
.9.

יתאו את השיטה הבנה את הד prostitu שבינ אל-עומר עד התוכן הפריסים והניקים של

ומ싶ים נזר חאיר. בצאת החזון אא נזר y ∈ [1, 2, 3] שקיסים את האילואר. עד לפלים שמסיס להזיב ערך לא השות.

.3(x=2)
.4(y=3)
.5(x=3)
.6.
.7.
.8.
.9.

יתאו את השיטה הבנה את הד prostitu שבין אל-עומר עד התוכן הפריסים והניקים של

ומبيق נזר חאיר. בצאת החזון אא נזר y ∈ [1, 2, 3] שקיסים את האילואר. עד לפלים שמסיס להזיב ערך לא השות.

.3(x=2)
.4(y=3)
.5(x=3)
.6.
.7.
.8.
.9.

יתאו את השיטה הבנה את הדProstitע שבין אל-עומר עד התוכן הפריסים והניקים של

ומبيق נזר חאיר. בצאת החזון אא נזר y ∈ [1, 2, 3] שקיסים את האילואר. עד לפלים שמסיס להזיב ערך לא השות.
The constraint satisfaction problem (CSP) is a framework for expressing and solving problems defined as a set of entities (variables), each with a domain of possible values, and a set of constraints that must be satisfied. The goal is to find an assignment of values to the variables that satisfies all constraints.

In this context, we consider the problem of finding a solution to a system of equations. The equations are given as a list of tuples, where each tuple represents a constraint. For example, the tuple `(C, V, D)` represents a constraint that variable `V` must be in the domain `D` for set `C`.

To solve this problem, we can use backtracking search. Initially, we choose an unassigned variable and assign it a value from its domain. If this assignment leads to a conflict, we backtrack and try another value. If we find a solution where all variables are assigned values, we stop.

This approach is efficient for small problems, but for larger problems, more sophisticated algorithms such as arc consistency and local search may be required.
רשימה של אלגוריתמים

1.1 (The non-recursive MAC solving algorithm) אלגוריתם MAC-ה אל קורס

1.2 (The HCSP solving algorithm) אלגוריתם הפתרון של HCSP

2.1 (Conflict analysis) בוחנה סתירות

2.2 (Printing the proof) הדפסת ההוכחה

3.1 (ANALYZECONFLICT algorithm in HCSP) אלגוריתם ANALYZECONFLICT

3.2 (COMBINE infers a new constraint $c^*$ from $c_1, c_2$, which satisfies (3.8) and (3.9), the requirements listed in Sec. 3.3.) Combine אלגוריתם Combine

4.1 (The HCSP constraint propagation algorithm) אלגוריתם הפתרון האלגוריתם של אלגוריתם הפתרון האלגוריתם של HCSP

4.2 (Value ordering heuristic in HCSP) האלגוריתם מסתמכים לע blir יוזה המטרה ב- HCSP-

4.3 (Optimizing an objective variable $o$ in HCSP) האלגוריתם מסתמכים לע blir יוזה המטרה ב- HCSP-

4.4 (The generation of an interpolant in HCSP) האלגוריתם מסתמכים לע blir יוזה המטרה ב- HCSP-
<table>
<thead>
<tr>
<th>רישומת אורותím</th>
<th>1.1</th>
<th>(Example with a graph-coloring CSP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
<td>HCSP -זנינק ביצות העתקה בגרף ביניים אלכסים</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>(The solving loop in HCSP)</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>(An implication graph corresponding to the running example.)</td>
</tr>
<tr>
<td>35</td>
<td></td>
<td>(Part of a conflict graph with relaxation)</td>
</tr>
<tr>
<td>63</td>
<td></td>
<td>(Comparison of number of instances solved within the given time limit with different algorithms)</td>
</tr>
<tr>
<td>64</td>
<td></td>
<td>(Comparing the number of backtracks for successful runs (log-scale).)</td>
</tr>
</tbody>
</table>
A Proof-Producing CSP Solver (A proof supplement)

Introduction to the Proof Supplement

Inference rules

Soundness proofs for the inference rules

Completeness of inference rules

Algorithms

CSP-Analyze-Conflict algorithm, the proof

Enhancements and optimizations

CSP-Analyze-Conflict node rejuvenation

Augmented explanations

Lookahead explanations

ביבליוגרפיה

ה
Learning general constraints in CSP (long version)

HCSP

HCSP Domains

HCSP Constraints

Decomposing the CSP into basic HCSP constraints

Constraint watches

Solving optimization problems

Introduction to optimization

Generating an interpolant

Constraints which HCSP supports
The Constraint Satisfaction Problem (CSP) 1.1
A few words about complexity 1.1.1
The CSP solver 1.2
The CSP solving algorithm 1.3
Propagating constraints 1.3.1
Decision Making 1.3.2
Conflict Analysis 1.3.3
Research contribution 1.4

A Proof-Producing CSP Solver (Published in AAAI-10) 2
Introduction 2.1


HCSP

http://tx.technion.ac.il/~mveksler/HCSP/ .
פתורון אילוציםنظم נבנוף למידה מעלה
אילוצים כללים

ינבר על מחקרה

לשם מדלי חלקי של ה드리ישת קרבנות התרぬ
דוקטור פילוסופיה

ميיכל ווקסלר

ווהש כלכית חברתית — מרכז טכנולוגיה לישראל
תמונה התחדד
2014
פתורון איילונים עם מגנוול זמן מדחה emits
איילונים כלליים

מיכאל ווקֶלר