

# Decomposition and Preprocessing of Ternary Constraint Networks

Pierre Talbot 

University of Luxembourg, Luxembourg

---

## Abstract

---

Constraint programming is a general and exact method based on constraint propagation and backtracking search. We provide a function decomposing a constraint network into a *ternary constraint network* (TCN) with a reduced number of operators. TCNs are not new and have been used since the inception of constraint programming, notably in constraint logic programming systems. This work aims to specify formally the decomposition function of discrete constraint network into TCN and its preprocessing. We aim to be self-contained and descriptive enough to serve as the basis of an implementation. Our primary usage of TCN is to obtain a regular data layout of constraints to efficiently execute propagators on graphics processing unit (GPU) hardware.

**2012 ACM Subject Classification** Theory of computation → Constraint and logic programming

**Keywords and phrases** Ternary constraint network, constraint programming

## 1 Introduction

Constraint programming is a general and exact method based on constraint propagation and backtracking search [7]. We provide a function decomposing a constraint network into a *ternary constraint network* (TCN) with a reduced number of operators (Section 3). TCN are not new and have been used since the inception of constraint programming, notably in constraint logic programming systems [16, 5, 3]. Our contribution is to specify the decomposition function of discrete constraint network into TCN (Section 3). Furthermore, as the size of the decomposition is up to 3 orders of magnitude larger than the initial constraint network, we apply several known preprocessing techniques to reduce its size (Section 4). Over the 100 instances of the MiniZinc 2024 challenge [13], the median increase is 4.8x in the number of variables and 4.5x in the number of constraints, although 12 instances remain more than 100x larger. Our paper aims to be self-contained and descriptive enough to serve as the basis of an implementation. We implemented the decomposition and preprocessing of MiniZinc constraint networks in our constraint solver Turbo [15].

Our primary usage of TCN is to obtain a regular data layout of constraints to efficiently execute propagators on graphics processing unit (GPU) hardware. Another use-case of TCN is for education. We have used this formalism to teach constraint programming and program solvers that can be used on MiniZinc instances [8]. TCNs are useful to describe propagation and focus on the fundamental of constraint programming.

## 2 Constraint Programming

In the following, we consider constraint programming over integer variables only. Let  $X$  be a finite set of variables and  $C$  be a finite set of constraints. For each constraint  $c \in C$ , let  $scp(c) \subseteq X$  be the set of free variables of  $c$ , called its *scope*—for instance,  $scp(x < y) = \{x, y\}$ . Without loss of generality, we represent the domain of variables using intervals. Let  $I = \{[\ell, u] \mid \ell \in \mathbb{Z} \cup \{-\infty\}, u \in \mathbb{Z} \cup \{+\infty\}, \ell \leq u\} \cup \{\perp\}$  be the set of intervals ordered by inclusion with a special element  $\perp$  representing the empty interval. We define  $lb([\ell, u]) \triangleq \ell$  and  $ub([\ell, u]) \triangleq u$  to extract the lower and upper bounds. A *constraint network* is a pair  $P = \langle d, C \rangle$  such that  $d \in X \rightarrow I$  is the *domain function*. We denote  $D$  the set of all domain functions  $X \rightarrow I$  ordered pointwise ( $d \leq d' \Leftrightarrow \forall x \in X, d(x) \subseteq d'(x)$ ). An *assignment* is a map  $asn : X \rightarrow \mathbb{Z}$ , and we denote the set of

all assignments by **Asn**. The set of solutions of a constraint is given by  $rel(c) \subseteq \mathbf{Asn}$ . The set of solutions of a constraint network is:

$$sol(d, C) \triangleq \{asn \in \mathbf{Asn} \mid \forall c \in C, \, asn \in rel(c) \wedge \forall x \in X, \, asn(x) \in d(x)\}$$

An interval propagator is a function  $p_c : \mathbf{D} \rightarrow \mathbf{D}$  where  $c \in C$  is a constraint. Let  $d \in \mathbf{D}$ , then a propagator is reductive ( $p_c(d) \leq d$ ), monotone ( $d \leq d' \Rightarrow p_c(d) \leq p_c(d')$ ) and sound ( $sol(d, \{c\}) \subseteq sol(p_c(d), \{\})$ ). It is also complete on singleton intervals: whenever  $\forall x \in scp(c), \exists v \in \mathbb{Z}, d(x) = [v, v]$ , then  $sol(d, \{c\}) \supseteq sol(p_c(d), \{\})$ .

Constraint propagation consists in finding the greatest fixpoint of a set of propagators  $\{p_1, \dots, p_n\}$  over a domain  $d \in X \rightarrow I$ . As long as the propagators are executed fairly, their order of execution does not matter and the same greatest fixpoint is always eventually reached [1]. This fact has been used to design various *propagation algorithms* to accelerate the computation of the fixpoint [11, 14]. As constraint propagation is sound but incomplete in general, it must be interleaved with a search procedure. Let  $split \in \mathbf{D} \rightarrow \mathcal{P}(\mathbf{D})$  be a strictly reductive ( $\forall d \in \mathbf{D}, \forall d' \in split(d), d' < d$ ) and sound and complete ( $\forall d \in \mathbf{D}, \bigcup\{sol(d', C) \mid d' \in split(d)\} = sol(d, C)$ ) branching procedure. We introduce next a standard optimization algorithm based on the *propagate-and-search* constraint solving algorithm. The following algorithm finds a solution  $best \in \mathbf{D}$  which minimizes the value of the variable  $z \in X$ .

```
function minimize( $d, C, best, z$ )
     $d(z) \leftarrow d(z) \cap [-\infty, lb(best(z)) - 1]$ 
    propagate( $d, C$ )
    if isasn( $d$ ) then
         $best \leftarrow d$ 
    else if  $\neg isbot(d)$  then
         $\forall d' \in split(d), \text{minimize}(d', C, best, z)$ 
    end if
end function
```

with

- $propagate(d, \{c_1, \dots, c_n\})$  computes the greatest fixpoint of  $p_{c_1} \circ \dots \circ p_{c_n}$  below  $d$ .
- $isasn(d) \triangleq \forall x \in X, \exists v \in \mathbb{Z}, d(x) = [v, v]$
- $isbot(d) \triangleq \exists x \in X, d(x) = \perp$

Here and thereafter, we always pass parameters by reference. The function *isasn* tests if  $d$  maps only to interval singletons (assignment) and *isbot* tests whether a variable has an empty domain, in which case we must backtrack. It is well-known that the algorithm *minimize* is a sound and complete solving procedure, see e.g. [7, 14]. The result holds even in the presence of infinite intervals as long as they become finite after a finite number of propagation steps.

## Representation of Propagators

It is possible to construct an infinite number of constraints, take for example the sequence  $\langle x_1 = x_2, x_1 = x_2 \odot_1 x_3, x_1 = x_2 \odot_1 x_3 \odot_2 x_4, \dots \rangle$  for a sequence of arithmetic operators  $\langle \odot_1, \odot_2, \dots \rangle$ . Therefore, we cannot implement a different propagator function for each possible constraint. Historically, solvers have limited their constraint languages in order to avoid implementing too many different propagators [16, 5, 3]. More complex constraints must be rewritten into supported constraints automatically or by the user. The *indexicals* approach is a particularly concise way of specifying propagators by mean of a few primitive constructions [4, 17].

However, the decomposition of constraints into primitive ones increase the number of auxiliary variables and propagators. It has been shown to be detrimental to the solver performance [12, 6].

Therefore, the key idea is to implement a propagator as an interpreter over the abstract syntax tree (AST) of the constraint. The propagator recursively traverses the AST to evaluate each node and compute the domain of each subexpression of the constraint. It is called a *view-based propagator* in recent work [12, 6], but a similar technique was already used in the HC-4 consistency algorithm [2].

Modern constraint solvers such as Choco and OR-Tools implement view-based propagation using inheritance to represent the AST, and subtype polymorphism to evaluate the tree. To avoid the overhead caused by dynamic subtyping, parametric polymorphism has also been used [6, 12], but it has the inconvenient that the solver must be recompiled for each new constraint problem tackled.

### 3 Ternary Constraint Network

In this section, we rewrite any constraint network into a *ternary constraint network* (TCN), that is, a constraint network with only constraints of arity 3 such as  $x = y + z$  and  $b = (x \leq y)$  with  $\{x, y, z, b\} \subseteq X$ .

► **Definition 1.** A ternary constraint network  $\langle d, C \rangle$  is a constraint network such that each  $c \in C$  is of the form  $x = y \odot z$  where  $x, y, z \in X$  are variables and  $\odot$  is a binary operator.

In this paper, the set of supported operators is  $\odot \in OP = \{+, *, /, mod, min, max, =, \leq\}$ <sup>1</sup>. The constraint language considered is sufficient to support all instances of the 2024 MiniZinc challenge. Unary constraints of the form  $x = k$ ,  $x \leq k$  and  $x \geq k$  where  $x \in X$  and  $k \in \mathbb{Z}$  are directly represented as domains of the variables. The definition of the set  $OP$  could be different. For instance, the constraint  $x = min(y, z)$  can be rewritten into  $-x = max(-y, -z)$  which, when decomposed into ternary constraints, gives  $\exists x', \exists y', \exists z', x' = max(y', z') \wedge x' = zero - x \wedge y' = zero - y \wedge z' = zero - z$ , with  $d(zero) = [0, 0]$ . It allows  $OP$  to be smaller, but it introduces three new variables and constraints. The *min* and *max* are particularly important since they also encode disjunction and conjunction, and it is preferable to keep them both. The lack of subtraction is justified by the relational semantics of constraints as we can rewrite  $x = y - z$  into  $y = x + z$  without loss of precision.

We now give a complete decomposition of a constraint network into a TCN, along with the proof that a TCN has exactly the same set of solutions than the initial constraint network. Note that the global constraints are decomposed in primitive constraints before this decomposition and using the MiniZinc compiler in the experiments.

Let  $\langle d \in X \rightarrow I, C \rangle$  be a constraint network. We introduce the function  $tcn(d, C) = \langle d', C' \rangle$  rewriting a constraint network into a ternary constraint network. The constraint language considered for  $C$  is sufficient to support all instances of the 2024 MiniZinc challenge. The function  $tcn$  only adds new variables and therefore we have  $dom(d) \subseteq dom(d')$  where  $dom$  is the domain of the function  $d$ . For clarity, we denote by  $(\| b ? X : Y \|)$  the function returning  $X$  if the expression  $b$  is true and  $Y$  otherwise. We also use the *let expression let*  $x = E$  **in**  $E'$  which binds the result of the evaluation of the expression  $E$  to the variable  $x$ , and returns the evaluation of the expression  $E'$  where  $E'$  can use  $x$ . It extends naturally to the case where  $E$  returns a tuple of values, e.g. *let*  $x, y = E$  **in**  $E'$ .

---

<sup>1</sup> We follow MiniZinc semantics by using truncated integer division and Euclidean modulus.

## Decomposition and Preprocessing of Ternary Constraint Networks

We first define several functions to extend a constraint network with a new variable  $x \notin \text{dom}(d)$ , and to update a variable already in  $d$ :

```

 $\text{extend}(d, x, [\ell, u]) = (d \cup \{x \mapsto [\ell, u]\}, x)$ 
 $\text{extend}(d, x) = \text{extend}(d, x, [-\infty, \infty])$ 
 $\text{extend}(d) = \text{extend}(d, \text{fresh}(d))$ 
 $\text{extend}_{\mathbb{B}}(d) = \text{extend}(d, \text{fresh}(d), [0, 1])$ 
 $\text{extend}_{\text{co}}(d, k) =$ 
  let  $c = \text{__CONSTANT\_}(\kappa < 0 ? \text{mk\_}k : k)$  in
   $(c \in \text{dom}(d) ? (d, c) : \text{extend}(d, c, [k, k]))$ 
 $\text{update}(d, x, [\ell, u]) =$ 
   $\{y \mapsto (x = y ? [\ell, u] \cap d(y) : d(y)) \mid y \in \text{dom}(d)\}$ 

```

The function  $\text{fresh}(d)$  returns a new variable's name  $x$  such that  $x \notin \text{dom}(d)$ . We use the function  $\text{extend}_{\mathbb{B}}$  to create a new Boolean variable. To avoid creating different variables with the same constant value, we create a unique name for each constant. For instance,  $\text{__CONSTANT\_5}$  is the name of the variable for the constant 5, and  $\text{__CONSTANT\_m1}$  for the constant  $-1$ . We suppose that no variable's name in the initial constraint network is of this form. The function  $\text{update}(d, x, [\ell, u])$  intersects the domain  $[\ell_x, u_x]$  of a variable  $x$  with an interval  $[\ell, u]$ , where interval intersection is defined as  $[\ell_x, u_x] \cap [\ell, u] \triangleq [\max\{\ell_x, \ell\}, \min\{u_x, u\}]$ .

The function  $\text{tcn}(d, C) = \langle d', C' \rangle$  rewrites each constraint to be ternary:

```

 $\text{tcn}(d, \{\}) = \langle d, \{\} \rangle$ 
 $\text{tcn}(d, \{c_1, c_2, \dots, c_n\}) =$ 
  let  $d, T = \text{tcn}(d, \{c_2, \dots, c_n\})$  in
  let  $d, U, x = \text{tc}(d, c_1)$  in
   $\langle \text{update}(d, x, [1, 1]), T \cup U \rangle$ 

```

Each constraint  $c \in C$  is rewritten into a set  $T$  of TCN constraints, possibly with new variables in  $d$ , using the function  $\text{tc}$ . The variable  $x$  is a Boolean variable reifying the constraint  $c$ . As we are in the top-level conjunction, we must set this variable to 1 to activate the constraint which is done by updating its domain.

We introduce the recursive function  $\text{tc}(d, t) = (d', T, x)$  which rewrites a term or constraint  $t$  into a set  $T$  of TCN constraints. The result of the expression  $t$  (or its reification status if it is a constraint) is stored in the variable  $x$ . We first consider the base cases (variables and constants) and unary constraints (arithmetic negation  $-$ , set membership  $\in$  and absolute value function  $\text{abs}$ ). Variables are simply returned as any variable occurring in a constraint must already be in  $d$ . A new variable is created for each distinct constant  $k$  using  $\text{extend}_{\text{co}}$  defined above. Unary constraints are rewritten using equivalent ternary constraints. For the set membership, we need the function  $\text{itvs}(S)$  to turn a set  $S$  into a set of intervals, for instance  $\text{itvs}(\{1, 2, 3, 5\}) = \{[1, 3], [5, 5]\}$ . The rewriting strategy is always the same: we first rewrite the parameters of the function or predicate, and then assemble the results to rewrite the current expression.

$$\begin{aligned}
tc(d, x) &= (d, \{\}, x) \\
tc(d, -t) &= tc(d, 0 - t) \\
tc(d, t \in S) &= \\
&\quad \text{let } d, T, x = tc(d, t) \text{ in} \\
&\quad \text{let } d = update(d, x, [\min S, \max S]) \text{ in} \\
&\quad \text{let } d, U, y = \\
&\quad \quad tc(d, \bigvee_{[\ell, u] \in itvs(S)} (x \geq \ell \wedge x \leq u)) \text{ in} \\
&\quad \quad (d, T \cup U, y) \\
tc(d, k) &= \\
&\quad \text{let } d, x = extend_{co}(d, k) \text{ in} \\
&\quad (d, \{\}, x) \\
tc(d, abs(t)) &= \\
&\quad \text{let } d, T, x = tc(d, t) \text{ in} \\
&\quad \text{let } d, U, y = tc(d, 0 - x) \text{ in} \\
&\quad \text{let } d, V, z = tc(d, max(x, y)) \text{ in} \\
&\quad \text{let } d = update(d, z, [0, \infty]) \\
&\quad tc(d, T \cup U \cup V, z)
\end{aligned}$$

The rewriting of binary operators  $\odot \in OP$  is factored into a single function. When the operator is Boolean ( $\leq, =$ ), the result is stored into a Boolean variable, otherwise in an integer variable. The other operators can be rewritten into expressions containing operators in  $OP$ . In particular, as subtraction is the inverse of addition, we have  $x = y - z \Leftrightarrow y = x + z$ . It is not the case of multiplication and division over integers, which is why we keep both operators. We rewrite the operator  $\neq$  by negating equality. As we could be in a reified context, we cannot simply add the TCN constraint  $zero = (x = y)$ , which is why we delegate the rewriting to logical equivalence.

$$\begin{aligned}
tc(d, t_1 \odot t_2) &= \\
&\quad \text{let } d, x = \\
&\quad \quad (\odot \in \{\leq, =\} ? extend_{\mathbb{B}}(d) : extend(d)) \text{ in} \\
&\quad \text{let } d, T, y = tc(d, t_1) \text{ in} \\
&\quad \text{let } d, U, z = tc(d, t_2) \text{ in} \\
&\quad (d, T \cup U \cup \{x = y \odot z\}, x) \\
tc(d, t_1 \neq t_2) &= \\
&\quad \text{let } d, zero = extend_{co}(d, 0) \text{ in} \\
&\quad tc(d, zero \Leftrightarrow t_1 = t_2) \\
tc(d, t_1 - t_2) &= \\
&\quad \text{let } d, x = extend(d) \text{ in} \\
&\quad \text{let } d, T, y = tc(d, t_1) \text{ in} \\
&\quad \text{let } d, U, z = tc(d, t_2) \text{ in} \\
&\quad (d, T \cup U \cup \{y = x + z\}, x) \\
tc(d, t_1 \geq t_2) &= tc(d, t_2 \leq t_1) \\
tc(d, t_1 > t_2) &= tc(d, t_2 \leq t_1 - 1) \\
tc(d, t_1 < t_2) &= tc(d, t_1 \leq t_2 - 1)
\end{aligned}$$

The last step is to compile logical connectors into TCN constraints. Logical formulas can appear in arithmetic expressions, e.g.  $(x \neq y \wedge x \neq z) + (y \neq z) \geq 1$ , and integer variables in logical formulas, e.g.  $((x + w) \vee y) = z$ . In a Boolean context, we interpret the value of an interval  $[\ell, u]$  to be *true* iff  $0 \notin [\ell, u]$ , and *false* iff  $\ell = u = 0$ . However, if the constraint is reified, we must map *true* to 1 and not to any value. Because we compile  $\vee$  to the *min* function, and  $\wedge$  to the *max* function, we must ensure the result of logical connectors lies in the interval  $[0, 1]$ . The function *booleanize* maps the domain of an expression occurring in a Boolean context to a Boolean variable. As an optimization, we only map  $x$  to a Boolean variable if it is not already a Boolean variable, i.e. its domain is within the interval  $[0, 1]$ . This function is then used to rewrite all logical connectors. Note that we cannot directly rewrite  $t_1 \text{ xor } t_2$  as  $t_1 \neq t_2$ , otherwise expressions such as  $[1, 1] \text{ xor } [2, 2]$  would be *true*, which according to our semantics is not correct as both  $[1, 1]$  and  $[2, 2]$  should map to *true*.

```

booleanize(d, t) =
  let d, T, x = tc(d, t) in
    if  $\neg(d(x) \leq [0, 1])$  then
      let d, U, b = tc(d, x  $\neq$  0) in
        (d, T  $\cup$  U, b)
    else (d, T, x)

tc(d,  $\neg t$ ) = tc(d, t = 0)

tc(d,  $t_1 \Rightarrow t_2$ ) = tc(d,  $\neg t_1 \vee t_2$ )

tc(d,  $t_1 \wedge t_2$ ) =
  let d, b = extend $\mathbb{B}$ (d) in
  let d, T, b1 = booleanize(d, t1) in
  let d, U, b2 = booleanize(d, t2) in
    (d, T  $\cup$  U  $\cup$  {b = min(b1, b2)}, b)

tc(d,  $t_1 \vee t_2$ ) =
  let d, b = extend $\mathbb{B}$ (d) in
  let d, T, b1 = booleanize(d, t1) in
  let d, U, b2 = booleanize(d, t2) in
    (d, T  $\cup$  U  $\cup$  {b = max(b1, b2)}, b)

tc(d,  $t_1 \Leftrightarrow t_2$ ) =
  let d, b = extend $\mathbb{B}$ (d) in
  let d, T, b1 = booleanize(d, t1) in
  let d, U, b2 = booleanize(d, t2) in
    (d, T  $\cup$  U  $\cup$  {b = (b1 = b2)}, b)

tc(d,  $t_1 \text{ xor } t_2$ ) =
  let d, T, b1 = booleanize(d, t1) in
  let d, U, b2 = booleanize(d, t2) in
  let d, V, b = tc(d, b1  $\neq$  b2) in
    (d, T  $\cup$  U  $\cup$  V, b)

```

We now prove that every constraint network  $\langle d, C \rangle$  is equivalent to its rewriting  $tcn(d, C)$ , i.e., it has exactly the same set of solutions. Because we introduce new variables, we must restrict the solutions to those defined on the initial variables. Let  $asn : X \rightarrow \mathbb{Z}$  be an assignment and  $Y$  be a subset of  $X$ , its restriction  $asn|_Y$  is defined as  $\{y \mapsto asn(y) \mid y \in Y\}$ . We extend the restriction to

set of assignments  $A$  as follows:  $A|_Y = \{asn|_Y \mid asn \in A\}$ . We prove the following statements by structural induction on the formula, showing that at each step the set of solutions is preserved.

► **Proposition 2** (Soundness and completeness). *Let  $\langle d, C \rangle$  be a constraint network defined over the set of variables  $X$ , then  $\text{sol}(d, C) = \text{sol}(\text{tcn}(d, C))|_X$ .*

**Proof.** The function  $\text{tcn}$  applies recursively  $\text{tc}$  on each constraint. Therefore, we first prove the soundness and completeness of  $\text{tc}$ . Let  $d_1, T, x = \text{tc}(d, t)$  and  $Y = X \cup \{x\}$ , the induction hypothesis is:

$$\text{sol}(d_1, T)|_Y = \text{sol}(d_2, \{x = t\}) \text{ with } d_2 = d \cup \{x \mapsto [-\infty, \infty]\}$$

If  $t$  is a top-level constraint, then  $\text{tcn}$  sets the domain of  $x$  to 1, thus we fall back on the equality  $\text{sol}(d_1, T \cup \{x = 1\})|_Y = \text{sol}(d, \{c\})$  which holds since for any constraint  $c$ , we have  $c \Leftrightarrow (x \Leftrightarrow c \wedge x = 1)$ . Further, because  $\text{sol}(d, \{c_1\}) \cap \text{sol}(d, \{c_2\}) = \text{sol}(d, \{c_1, c_2\})$ , the equality holds for any constraint network  $\langle d, C \rangle$ .

We must prove the induction hypothesis holds for the function  $\text{tc}$ . When appropriate, we prove the induction hypothesis by showing a logical equivalence  $x = t \Leftrightarrow x = t'$ , in which case we must have, by the induction hypothesis on  $t$  and  $t'$ , the equality  $\text{sol}(d_1, T)|_Y = \text{sol}(d_2, T')|_Y$  where  $d_1, T, x = \text{tc}(d, t)$  and  $d_2, T', x = \text{tc}(d, t')$ .

- Variable: the domain  $d_1$  is unchanged, hence  $d_1 = d_2$ . The set of constraints is empty and  $x = x$  is a tautology. Therefore,  $\text{sol}(d_1, T)|_Y = \text{sol}(d_1, \{\})|_Y = \text{sol}(d_2, \{x = x\})|_Y$ .
- Constant: by definition we have  $x \in \text{dom}(d_2)$  and  $d_1(x) = [k, k]$ . For any constraint network  $\langle d, C \rangle$  with  $x \in \text{dom}(d)$ , we have the equality  $\text{sol}(\text{update}(d, x, [k, k]), C) = \text{sol}(d, C \cup \{x = k\})$ .
- Negation: by logical equivalence  $x = -t \Leftrightarrow x = 0 - t$ .
- Membership: by logical equivalence of  $t \in S \Leftrightarrow \bigvee_{v \in S} t = v$  and  $t = v \vee t = v + 1 \vee \dots \vee t = v + n \Leftrightarrow t \geq v \wedge t \leq v + n$ . Furthermore, the update of  $d(x)$  with  $[\min S, \max S]$  is implied by the membership constraint and therefore does not modify the set of solutions.
- Absolute value function: by logical equivalence  $x = \text{abs}(t) \Leftrightarrow x = \text{max}(t, -t)$ . Furthermore, the update of  $d(x)$  with  $[0, \infty]$  does not modify the set of solutions as it is implied by the tautology  $\text{max}(t, -t) \geq 0$ .
- TCN ternary constraint: by logical equivalence  $x = t_1 \odot t_2 \Leftrightarrow x = y \odot z \wedge y = t_1 \wedge z = t_2$ .
- Subtraction: by logical equivalence  $x = t_1 - t_2 \Leftrightarrow t_1 = x + t_2$ .
- Inequalities: by logical equivalences  $x = t_1 \geq t_2 \Leftrightarrow x = t_2 \leq t_1$ ,  $x = t_1 > t_2 \Leftrightarrow x = t_2 \leq t_1 - 1$ ,  $x = t_1 < t_2 \Leftrightarrow x = t_1 \leq t_2 - 1$ , and  $x = t_1 \neq t_2 \Leftrightarrow x = (0 \Leftrightarrow (t_1 = t_2))$ .
- Logical connectors: we first notice that logical operators are functions  $\square \in \mathbb{B}^n \rightarrow \mathbb{B}$  where  $n$  is the arity of the logical operator  $\square$ . The conversion from an integer value to a Boolean value is done implicitly. However, this conversion must be encoded explicitly when decomposing the constraints because we use arithmetic operators  $\odot \in \mathbb{Z}^n \rightarrow \mathbb{Z}$  to encode the logical operators, which does not have the same domains than their logical equivalent  $\square$ . We restrict the parameter to Boolean value by converting them in *booleanize*. The conversion of  $y$  to  $b_1$  is given by  $b_1 = 0 \Leftrightarrow y = 0$  and  $b_1 = 1 \Leftrightarrow y \neq 0$ , and similarly  $z$  to  $b_2$ . Then we have the following equivalences:
  - Logical negation:  $x = \neg t_1 \Leftrightarrow (x = (b_1 = 0))$ .
  - Conjunction:  $(x = t_1 \wedge t_2) \Leftrightarrow (x = \text{max}(b_1, b_2))$ , and  $\forall a, b \in \mathbb{B}, \text{max}(a, b) \in \mathbb{B}$ .
  - Disjunction:  $(x = t_1 \vee t_2) \Leftrightarrow (x = \text{min}(b_1, b_2))$ , and  $\forall a, b \in \mathbb{B}, \text{min}(a, b) \in \mathbb{B}$ .
  - Implication:  $(x = t_1 \Rightarrow t_2) \Leftrightarrow (x = \neg t_1 \vee t_2)$ .
  - Equivalence:  $(x = t_1 \Leftrightarrow t_2) \Leftrightarrow (x = (b_1 = b_2))$ .
  - Exclusive disjunction:  $(x = t_1 \text{ xor } t_2) \Leftrightarrow (x = (b_1 \neq b_2))$ .



► **Proposition 3** (Uniqueness). *Further, we have a bijection between both set of solutions:  $|sol(d, C)| = |sol(tcn(d, C))|$ .*

**Proof.** Let  $s_1, s_2$  be two solutions in  $sol(tcn(d, C))$  and  $s$  be a solution in  $sol(d, C)$ . Suppose that  $s_1|_X = s$  and  $s_2|_X = s$ . It means that  $s_1$  and  $s_2$  differ on the value of at least an auxiliary variable  $x$ , or are the same. We show by induction that the variables in  $dom(s)$  fully define the auxiliary variables, and therefore  $s_1$  and  $s_2$  must be equal. Our induction hypothesis is that the variable returned by  $tc$  is fully defined.

- Variable: no auxiliary variable is created.
- Constant: the auxiliary variable is created with a single value.
- Ternary constraint  $x = y \odot z$ : by induction hypothesis,  $y$  and  $z$  are fully defined. Since all  $\odot$  are functions, it is necessary that the auxiliary variable  $x$  is fully defined whenever  $y$  and  $z$  are. Note that division is fully defined over the intervals as dividing by zero maps to the empty interval.
- Subtraction  $y = x + z$ : similarly,  $x$  must be fully defined whenever  $y$  and  $z$  are since the inverse of subtraction is a function (addition).
- Logical operators  $b = b_1 \square b_2$ : new Boolean variables are only created for the conjunction, disjunction and equivalence. By induction hypothesis,  $b_1$  and  $b_2$  are fully defined. The operators min, max and  $=$  are functions, hence  $b$  must be fully defined as well.



## 4 Preprocessing of Ternary Constraint Network

We follow standard preprocessing techniques that we specialize to ternary constraint networks [10, 9]. The goal of preprocessing is essentially to remove variables and constraints. Before presenting our preprocessing algorithm, we define a structure to keep track of equivalent variables.

We aim at finding a partition  $E$  of  $X$  such that each component  $Y$  of  $E$  represents a set of equivalent variables. More precisely, all pairs of variables  $x, y \in Y$  are connected by an equality constraint  $x = y$ . We write  $[x]_E \in E$  the equivalence class of  $x$  in  $E$ . We suppose variables are totally ordered (e.g. by an indexing) and we choose  $\min Y$  to be the representative variable of the equivalence class  $Y$ .

The equivalence classes are discovered by various preprocessing techniques. Initially, we suppose the variables are all distinct which is given by the partition  $init(X) \triangleq \{\{x\} \mid x \in X\}$ . We add a variable equality  $x = y$  by removing both equivalence classes  $[x]_E$  and  $[y]_E$  from  $E$  and adding back their union into the partition.

$$\begin{aligned} merge(E, x, y) &\triangleq \\ &\text{let } XY = \{[x]_E\} \cup \{[y]_E\} \text{ in } (E \setminus XY) \cup \{\bigcup XY\} \end{aligned}$$

The interval domain of a variable  $x \in X$  in an equivalence class  $Y$  is the intersection of all variables' domains in  $Y$ , defined as  $d_E(x) \triangleq \bigcap_{y \in [x]_E} d(y)$ . During preprocessing, we read the domain of a variable using  $d_E$  instead of  $d$ . In practice, we implement the partition efficiently using a union-find data structure.

We apply seven preprocessing functions:

- *Propagation* to reduce the domains of the variables.
- *Algebraic simplification* to eliminate constraints for which the solutions can be expressed directly in the domains of the variables. It also detects equivalence between variables and refine the partition  $E$ .
- *Common subexpression elimination* detects ternary constraints with the same right-hand side. For instance, consider  $a = y + z$  and  $b = y + z$ , the procedure eliminates one of the two constraints and merge the equivalence classes of  $a$  and  $b$ .

- *Merge domains* of the variables in the equivalence classes. It is especially useful for propagation which is not aware of the equivalence classes.
- *Entailed constraint elimination* to remove the constraints that are entailed by  $d$ . For instance, if  $d(x) = [1, 2]$  and  $d(y) = [2, 3]$ , then the ternary constraint  $1 = (x \leq y)$  is always true regardless of the evolution of  $d$ .
- *Variable renaming* to use a unique variable per equivalence class.
- *Useless variable elimination* to remove variables not in the scope of any constraint. We keep the variables with an empty domain to be able to detect unsatisfiability.

The preprocessing steps are combined in the computation of a greatest fixpoint over the triple  $\langle d, C, E \rangle$ . We stop once  $d$  and  $E$  do not change anymore, and we ignore the modifications on  $C$ . To be able to define a greatest fixpoint, we must define a partial order on partitions. Let  $E, E'$  be two partitions such that  $\bigcup E = \bigcup E'$ , then we define the following partial order:  $E \leq E' \Leftrightarrow \forall Y \in E, \exists Z \in E', Y \supseteq Z$ . The intuition is that we only merge equivalence classes and never divide an existing one while preprocessing. We ignore the change on  $C$  because, by inspecting the places where we rewrite constraints, the rewritten constraint cannot trigger further rewriting without a change in  $d$  or  $E$ . Therefore, if  $d$  and  $E$  do not change, all constraints that can be rewritten must have been rewritten already.

The detection of entailed constraints and useless variables, as well as variable renaming, are extracted outside of the fixpoint loop. Indeed, regardless of the modifications on  $d$  and  $C$ , an entailed constraint will stay entailed, and a variable not occurring in the scope of any constraint will remain useless. Formally, preprocessing is defined by the following two algorithms.

$$\begin{aligned} preprocess(d, C) &\triangleq \\ &\text{let } E = init(dom(d)) \text{ in} \\ &\quad (\text{initialize equivalence classes}) \\ &\text{let } \langle d, C, E \rangle = \text{gfp}_{\langle d, C, E \rangle} \text{ preprocess} \\ &\quad (\text{see overloaded def. below}) \\ &\text{let } C = C \setminus \{c \in C \mid \gamma(d) \subseteq rel(c)\} \text{ in} \\ &\quad (\text{eliminate entailed constraints}) \\ &\text{let } R = \{x \mapsto \min [x]_E \mid x \in dom(X)\} \text{ in} \\ &\quad (\text{create a substitution}) \\ &\text{let } C = \{c[R] \mid c \in C\} \text{ in} \\ &\quad (\text{rename variables by their representative elements}) \\ &\text{let } d = \{x \mapsto d(x) \mid x \in \bigcup_{c \in C} scp(c) \vee d(x) = \perp\} \text{ in} \\ &\quad (\text{eliminate useless variables}) \\ &\langle d, C \rangle \end{aligned}$$

$$\begin{aligned} preprocess(d, C, E) &\triangleq \\ &\text{let } d = \text{gfp}_d p_{c_1} \circ \dots \circ p_{c_n} \text{ in} \\ &\quad (\text{root propagation}) \\ &\text{let } \langle d, C, E \rangle = as(d, C, E) \text{ in} \\ &\quad (\text{algebraic simplification}) \\ &\text{let } \langle d, C, E \rangle = \text{gfp}_{\langle d, C, E \rangle} icse \text{ in} \\ &\quad (\text{common subexpression elimination}) \\ &\text{let } d = \{x \mapsto \text{dom}(E, d, x) \mid x \in X\} \text{ in} \\ &\quad (\text{merge domains of equivalent variables}) \\ &(\langle d, C \rangle, E) \end{aligned}$$

Note that in practice, we must save the partition  $E$  and eliminated variables in order to print the solutions with the original variables of the model, but this poses no particular challenge.

We give an example of the algorithm on the constraint network  $\langle d, \{x = y + z, w = y + z, x = (y = z)\} \rangle$  with  $d(x) = [0, 1]$ ,  $d(w) = [1, 2]$  and all other variables with domain  $[-\infty, \infty]$ . Propagation is unable to remove any value from the domains and *as* does not detect any equality constraint since  $x$  could be equal to 0. Then, *icse* detects the equality  $x = w$ , modifies the partition to  $\{\{x, w\}, \{y\}, \{z\}\}$  and removes the constraint  $x = (y = z)$ . The propagation is still inefficient, but this time *as* is able to detect the equality  $y = z$  since  $d_E(x) = d(x) \cap d(w) = [1, 1]$ , and the partition is updated to  $\{\{x, w\}, \{y, z\}\}$ . At the next iteration, *as* rewrites the constraint  $w = y + z$  into  $w = y * 2$ . It enables the propagation step to reduce the domain of  $y$  to  $\perp$ , effectively detecting the problem unsatisfiable. At that point, the constraint network is  $\langle d, \{w = y * 2\} \rangle$  and the fixpoint of *preprocess* is reached. The entailment checking step removes the constraint  $w = y * 2$  as  $\gamma(d) = \{\}$ . In this situation, the problem has been completely solved, and detected unsatisfiable, by preprocessing.

### Common Subexpression Elimination

We eliminate the TCN subexpressions that are identical by iterating over each pair of constraints in  $C$ .

```

 $icse(d, \{\}, E) = \langle d, \{\}, E \rangle$ 
                                (base cases  $|C| \leq 1$ )
 $icse(d, \{c\}, E) = \langle d, \{c\}, E \rangle$ 
 $icse(d, \{x_1 = (y_1 \odot_1 z_1), c_2, \dots, c_n\}, E) =$ 
    let  $\langle d, C, E \rangle = icse(d, \{c_2, \dots, c_n\}, E)$  in
    if  $\exists(x_i = (y_i \odot_i z_i)) \in C$ ,  $\odot_1 = \odot_i$ 
         $\wedge ((y_1 = y_i \wedge z_1 = z_i) \vee (\odot_1 \in \mathbf{C} \wedge y_1 = z_i \wedge z_1 = y_i))$ 
    then
        (detecting subexpression equality)
         $\langle d, C, merge(E, x_1, x_i) \rangle$ 
            ( $c_1$  is removed and an equality is added)
    else
         $\langle d, C \cup \{x_1 = (y_1 \odot_1 z_1)\}, E \rangle$ 

```

Note that we implement this algorithm efficiently in  $\mathcal{O}(|C|)$  by using a hash map between  $y \odot z$  and  $x$  for each constraint  $x = (y \odot z)$ . To account for commutativity, the hashing function must give the same result for  $y \odot z$  and  $z \odot y$ , which can be done by simply multiplying the indexes of the variables and the operator. The equality function of the hash map must compare the elements taking into account commutativity as shown in the algorithm above.

### Algebraic Simplification

We present a few rewriting rules that are not taken into account when performing interval propagation, especially because propagation does not deal well with multiple occurrences of the same variable in a constraint. The function *as* is essentially a pattern matching algorithm over the constraints. We write  $\mathbf{C} = \{+, *, \min, \max, =\}$  the set of commutative operators. To simplify the notation, we rewrite

each constraint in a normal form as follows:

```

 $nf(E, d, x = y \odot z) =$ 
let  $y, z = (\odot \in \mathbf{C} \wedge y \leq z ? (y, z) : (z, y))$  in
    (normalize commutative operators)
let  $y, z = (\odot \in \mathbf{C} \wedge d_E(y) = [k, k] ? (z, y) : (y, z))$  in
    (constant on the right of  $\odot$ )
let  $x, y, z = \min [x]_E, \min [y]_E, \min [z]_E$  in
    (variable substitution)
 $(x = y \odot z)$ 

```

Furthermore, an integer **k** in bold font denotes a variable  $x$  such that  $d(x) = [k, k]$ .

```

as(d, {}, E) = ⟨d, {}, E⟩

as(d, {c1, c2, ..., cn}, E) =
let d, 2 = extendco(d, 2) in
let d, C, E = as(d, {c2, ..., cn}) in
match nf(c1) with
| x = x + y → ⟨update(d, y, [0, 0]), C, E⟩
| x = y + 0 → ⟨d, C, merge(E, x, y)⟩
| x = y + y → ⟨d, C ∪ {x = y * 2}, E⟩
| x = x * k →
    (⟨k = 1 ? ⟨d, C, E⟩ : ⟨update(d, x, [0, 0]), C, E⟩⟩)
| k = x * x →
    (⟨∃n ∈ N, n * n = k
     ? ⟨update(d, x, [−√k, √k]), C ∪ {c1}, E⟩
     : ⟨update(d, x, ⊥), C, E⟩⟩)
| x = y * 1 → ⟨d, C, merge(E, x, y)⟩
| x = x * x → ⟨update(d, x, [0, 1]), C, E⟩
| x = 1/x → ⟨update(d, x, [−1, 1]), C ∪ {c1}, E⟩
| x = 0/x → ⟨update(d, x, ⊥), C, E⟩
| k = x/x →
    (⟨k = 1
     ? ⟨d, C ∪ {c1}, E⟩
     : ⟨update(d, x, ⊥), C, E⟩⟩)
| x = y/1 → ⟨d, C, merge(E, x, y)⟩
| x = x/x → ⟨update(d, x, [1, 1]), C, E⟩
| x = x mod x → ⟨update(d, x, [0, 0]), C, E⟩
| x = x mod k →
    ⟨update(d, x, [0, abs(k) − 1]), C, E⟩
| x = k mod x → ⟨update(d, x, ⊥), C, E⟩
| 0 = x mod x → ⟨d, C, E⟩
| x = min(y, y) → ⟨d, C, merge(E, x, y)⟩
| x = max(y, y) → ⟨d, C, merge(E, x, y)⟩
| x = min(x, y) → ⟨d, C ∪ {1 = (x ≤ y)}, E⟩
| x = max(x, y) → ⟨d, C ∪ {1 = (y ≤ x)}, E⟩
| 1 = (x = y) → ⟨d, C, merge(E, x, y)⟩
| x = (y = y) → ⟨update(d, x, [1, 1]), C, E⟩
| x = (x = k) → ⟨(⟨k = 0 ? update(d, x, ⊥)
    (⟨k = 1 ? update(d, x, [1, 1])
     : update(d, x, [0, 0])⟩), C, E⟩)
| x = (y ≤ y) → ⟨update(d, x, [1, 1]), C, E⟩
| x = (x ≤ k) → ⟨(⟨k = 0 ? update(d, x, ⊥)
    (⟨k > 0 ? update(d, x, [1, 1])
     : update(d, x, [0, 0])⟩), C, E⟩)
| x = (k ≤ x) → ⟨(⟨k = 1 ? d
    (⟨k < 1 ? update(d, x, [1, 1])
     : update(d, x, [0, 0])⟩), C, E⟩)

```

	Variables				Constraints			
	average	median	stddev	max	average	median	stddev	max
FlatZinc	9.42x	1.86x	18.63x	111.62x	24.94x	2.95x	67.91x	486.87x
TCN	53.61x	7.97x	151.61x	1133.22x	76.68x	6.21x	265.88x	1837.19x
Preprocessed	22.06x	4.76x	50.48x	344.62x	36.39x	4.33x	115.18x	746.09x

■ **Table 1** Increase in size of constraint networks relatively to Choco constraint networks over 89 instances.

There are a few rules we left out because they are already implicitly encoded by constraint propagation over intervals. It is the case of the absorbing element 0 for multiplication and division. Note that there is no need to evaluate constants as this is already taken into account by propagation and the removal of entailed constraints.

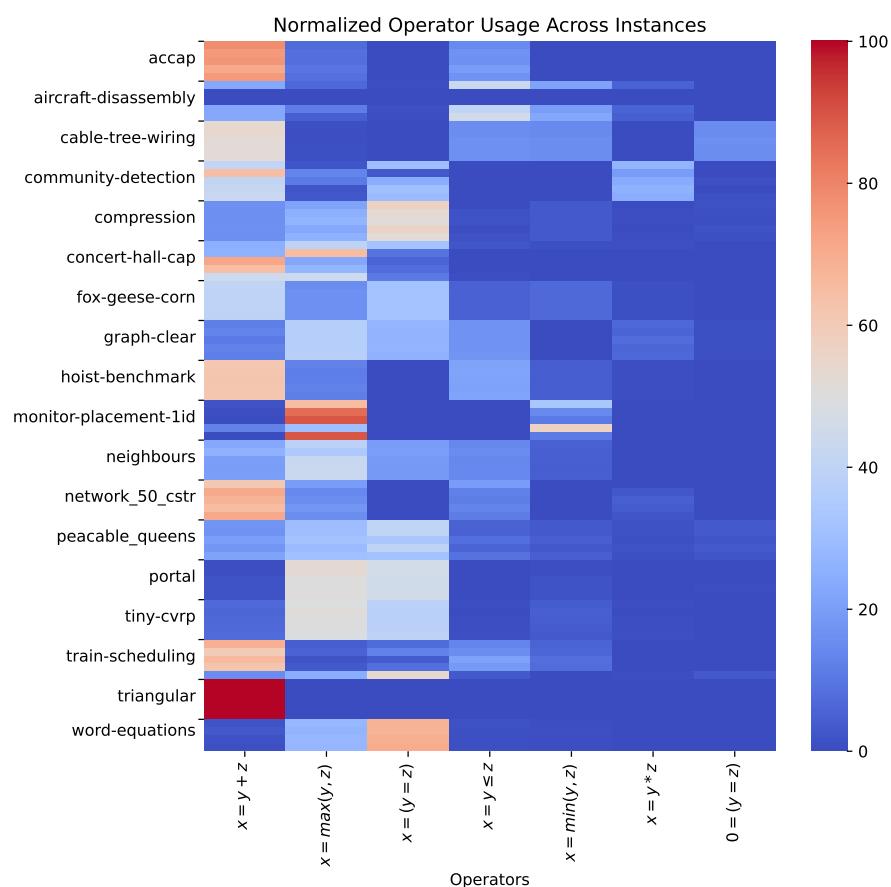
## 5 Analysis of Ternary Constraint Networks

As we do not support any global constraint, the MiniZinc to FlatZinc conversion is already quite costly. Table 1 shows the average, median, standard deviation and maximum increase in the numbers of variables and constraints for fully decomposed MiniZinc model (without global constraints), after decomposition into TCN (Section 3) and after preprocessing (Section 4). On 63/89 instances, the FlatZinc decomposition is less than an order of magnitude larger than the Choco constraint network in both the number of variables and constraints. After applying the TCN decomposition, we further increase by 5 times the number of variables and 3 times the number of constraints on average. Fortunately, the preprocessing step reduces this increase to 2.5 times for variables and 1.5 times for constraints on average, which is a reasonable increase considering the constraint network only contains ternary constraints. For a more precise overview, we provide a scatter plot of the variables and constraints increases for all instances in Figure 2. The average preprocessing time is 24.22s with a standard deviation of 96.91s. The median time is 0.91s and only 11 instances take more than 10 seconds to be preprocessed.

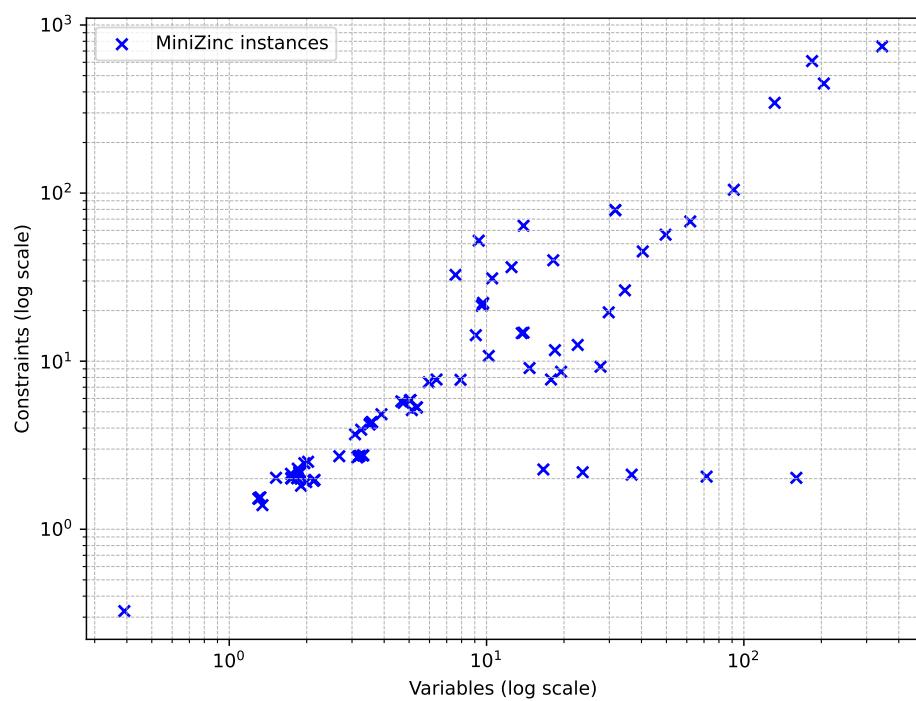
On Figure 1, we analyze the usage of the operators across instances. Clearly, thread divergence is not be the main concern anymore since the instances are not using a wide variety of operators. In particular, there is no instance with modulo and division operations (or they are simplified after preprocessing). Because of the preprocessing, all non-reified equality constraints  $1 = (x = y)$  are deleted. The constraint  $x = y \leq z$  only exists in a reified context and  $0 = y \leq z$  (modelling  $y > z$ ) and  $1 = y \leq z$  have completely disappeared. It is a consequence of the FlatZinc and TCN decomposition,  $y \leq z$  is rewritten  $y - z \leq 0$  by the FlatZinc decomposition, which is further rewritten  $y = x + z \wedge x \leq 0$  leading to  $\leq$  and  $>$  being directly managed in the domain of the variables. The most used TNF constraints are addition, maximum which is used to encode disjunction, and reified equality. Interestingly, although present in 5 problem classes, the decomposition of all-different into  $\mathcal{O}(n^2)$  constraints of the form  $0 = (x = y)$  does not seem to be a bottleneck.

## References

- 1 Krzysztof R. Apt. The essence of constraint propagation. *Theoretical computer science*, 221(1-2):179–210, 1999. URL: <http://www.sciencedirect.com/science/article/pii/S0304397599000328>, doi:10.1016/S0304-3975(99)00032-8.
- 2 Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-François Puget. Revising hull and box consistency. In *Proceedings of the 1999 International Conference on Logic Programming*, page 230–244, USA, 1999. Massachusetts Institute of Technology.



**Figure 1** Heatmap of the operators used in preprocessed ternary constraint networks over 89 instances grouped by problem classes.



■ **Figure 2** Increase in size of preprocessed ternary constraint networks relatively to Choco constraint networks (95 instances).

- 3 Frédéric Benhamou and William J. Older. Applying interval arithmetic to real, integer, and boolean constraints. *The Journal of Logic Programming*, 32(1):1 – 24, 1997. URL: <http://www.sciencedirect.com/science/article/pii/S0743106696001422>, doi:10.1016/S0743-1066(96)00142-2.
- 4 Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. *Programming Languages: Implementations, Logics, and Programs*, 1997.
- 5 Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *The Journal of Logic Programming*, 27(3):185 – 226, 1996. URL: <http://www.sciencedirect.com/science/article/pii/0743106695001212>, doi:10.1016/0743-1066(95)00121-2.
- 6 Marco Correia and Pedro Barahona. View-based propagation of decomposable constraints. *Constraints*, 18(4):579–608, October 2013. URL: <http://link.springer.com/10.1007/s10601-013-9140-8>, doi:10.1007/s10601-013-9140-8.
- 7 Christophe Lecoutre. *Constraint networks: techniques and algorithms*. ISTE/John Wiley, Hoboken, NJ, 2009.
- 8 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming—CP 2007*, pages 529–543. Springer, 2007. URL: [http://link.springer.com/chapter/10.1007/978-3-540-74970-7\\_38](http://link.springer.com/chapter/10.1007/978-3-540-74970-7_38).
- 9 Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in Savile Row. *Artificial Intelligence*, 251:35–61, October 2017. URL: <https://linkinghub.elsevier.com/retrieve/pii/S0004370217300747>, doi:10.1016/j.artint.2017.07.001.
- 10 Andrea Rendl. *Effective compilation of constraint models*. PhD Thesis, University of St Andrews, 2010.
- 11 Christian Schulte and Peter J. Stuckey. Efficient Constraint Propagation Engines. *ACM Trans. Program. Lang. Syst.*, 31(1):2:1–2:43, December 2008. URL: <http://doi.acm.org/10.1145/1452044.1452046>, doi:10.1145/1452044.1452046.
- 12 Christian Schulte and Guido Tack. View-based propagator derivation. *Constraints*, 18(1):75–107, January 2013. URL: <http://link.springer.com/10.1007/s10601-012-9133-z>, doi:10.1007/s10601-012-9133-z.
- 13 Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. The MiniZinc challenge 2008–2013. *AI Magazine*, 35(2):55–60, 2014. URL: <https://www.aaai.org/ojs/index.php/aimagazine/article/view/2539>.
- 14 Guido Tack. *Constraint Propagation – Models, Techniques, Implementation*. PhD thesis, Saarland University, 2009.
- 15 Pierre Talbot, Frédéric Pinel, and Pascal Bouvry. A Variant of Concurrent Constraint Programming on GPU. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 3830–3839, Jun. 2022. doi:10.1609/aaai.v36i4.20298.
- 16 Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.
- 17 Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(FD). *The Journal of Logic Programming*, 37(1–3):139–164, 1998. URL: <http://www.sciencedirect.com/science/article/pii/S0743106698100067>, doi:[http://dx.doi.org/10.1016/S0743-1066\(98\)10006-7](http://dx.doi.org/10.1016/S0743-1066(98)10006-7).