Box Constraint Collections for Adhoc Constraints *

K.C.K. Cheng[1], J.H.M. Lee[1], and P.J. Stuckey[2]

[1] Dept of Comp. Sci. & Eng.
The Chinese University of Hong Kong, Hong Kong SAR
{ckcheng,jlee}@cse.cuhk.edu.hk
[2] Dept. of Comp. Sci. & Soft. Eng.
University of Melbourne, Australia
pjs@cs.mu.oz.au

Abstract. In this paper, we propose a new language-independent representation of adhoc constraints, called a box constraint collection. Using constructive disjunction, this representation achieves domain consistency. We develop an algorithm to automatically generate a box constraint collection for a given adhoc constraint. The result is guaranteed to be complete and correct, and achieve domain consistency. The constructive disjunction propagator for the box constraint collection can be efficiently implemented using indexicals. We give correctness and completeness result for our compilation scheme, and outline optimization techniques. Experiments show that our representation is simple, compact, and propagates efficiently.

1 Introduction

Constraint programming is a promising technique for solving many difficult combinatorial problems. Since real-life constraints can be difficult to describe in symbolic expressions, or provide very weak propagation from their symbolic representation, they are sometimes represented in the form of the sets of solutions or sets of nogoods. This adhoc representation provides strong propagation through generalized arc consistency techniques. However, the adhoc representation is expensive in terms of memory and computation, when the adhoc constraint is large.

There is interest in determining less expensive methods for building propagators for adhoc constraints. The first step in this direction was the automatic generation of propagation rules pioneered by Apt and Monfroy [4]. They represent an adhoc constraint as a set of simple rules of the form $x_1 = v_1 \wedge \ldots \wedge x_n = v_n \rightarrow y \neq a$ such that rule consistency, which is weaker than domain consistency, is achieved. These rules can be extended to $x_1 \in S_1 \wedge \ldots \wedge x_n \in S_n \rightarrow y \neq a$, such that domain consistency is achieved. They propose two algorithms to generate all non-redundant rules for a given adhoc constraint.

Apt and Monfroy's work is extended by Abdennadher and Rigotti [2], who express the propagation rules in CHRs [10] so that user-defined predicates are allowed. They

* We thank the anonymous referees for their constructive comments. The work described in this paper was substantially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region (Project no. CUHK4183/00E).

develop the PROPMINER algorithm, which generates all non-redundant propagation rules based on the set of user-defined predicates. Constraint handling rules, while expressive, are less efficient than other approaches to implementing constraint solvers.

Indexicals are powerful, and efficient language to define constraint propagation. Dao *et al.* [8] propose a framework and two algorithms to learn indexical operators (a subset of the indexical operators available in GNU Prolog [9]) that achieve bounds-consistency for adhoc constraints. They require that the indexicals must not delete a solution of the original constraint, and at the same time they try to minimize the cases that a nogood is wrongly classified as a solution. Under this formulation, the output indexicals are correct (i.e. they will not remove a solution), but may be incomplete (i.e. they may not detect all nogoods). However, they show that indexicals with good pruning power can often be discovered. Barták [5] gives an efficient filtering algorithm as the basis of the implementation of a binary tabled constraint by clustering the tuples into boxes, but does not discuss how to find the boxes.

In this paper, we propose a new language-independent representation for adhoc constraints, the *box constraint collection*. The idea is to break up an adhoc constraint into pieces and cover these pieces using *box constraints* as tiles. With the aid of constructive disjunction and a suitable choice of forms of constraint to use in the collection, our new representation achieves domain consistency. We can compile this representation using the indexical language provided by SICStus Prolog, to provide efficient propagators for adhoc constraints.

We describe an algorithm, bccFinder, that automatically generates a box constraint collection for an adhoc constraint. The output representation is guaranteed to be complete, correct, and achieve domain consistency. We also suggest a compilation scheme which generates efficient indexicals for box constraint collections, and outline optimization techniques. Experiments confirm the compactness of our representation and efficiency in propagation.

2 Propagation Based Constraint Solving

In this section we give our terminology for constraint satisfaction problems, and propagation based constraint solving.

An *integer valuation* θ is a mapping of variables to integer values, written $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$. We extend the valuation θ to map expressions and constraints involving the variables in the natural way. Let $vars$ be the function that returns the set of (free) variables appearing in a constraint or valuation.

A *domain* D is a complete mapping from a fixed (countable) set of variables $\mathcal{V}$ to finite sets of integers. A *false domain* D is a domain with $D(x) = \emptyset$ for some x. A domain D_1 is *stronger* than a domain D_2, written $D_1 \sqsubseteq D_2$, if $D_1(x) \subseteq D_2(x)$ for all variables x.

In an abuse of notation, we define a valuation θ to be an element of a (non-false) domain D, written $\theta \in D$, if $\theta(x_i) \in D(x_i)$ for all $x_i \in vars(\theta)$.

We are also interested in the notion of an *initial domain*, denoted by D_{init}. The initial domain gives the initial values possible for each variable.

A *constraint* c over variables $x_1, \ldots, x_n$, written as $c(x_1, \ldots, x_n)$, restricts the values that each variable x_i can take simultaneously. An *adhoc* constraint $c(x_1, \ldots, x_n)$ is defined *extensionally* as a set of valuations θ over the variables $x_1, \ldots, x_n$. We say $\theta \in c$ is a *solution* of c. For any valuation θ on variables $x_1, \ldots, x_n$, with $\theta \notin c$, we call θ a *nogood* of c.

Often we define constraints *intensionally* using some well understood mathematical syntax. For an intensionally defined constraint c we have that $\theta \in c$ iff $vars(\theta) = vars(c) \wedge \mathcal{Z} \models_\theta c$, where $\mathcal{Z}$ is the integers. For example the constraint $x_1 = x_2 + 1$ where $D_{init}(x_1) = D_{init}(x_2) = \{1, 2, 3\}$ defines the solution set $\{\{x_1 \mapsto 2, x_2 \mapsto 1\}, \{x_1 \mapsto 3, x_2 \mapsto 2\}\}$.

Two constraints c_1 and c_2 are *equivalent* to each other, denoted by $c_1 \equiv c_2$, if they define the same set of solutions.

A *constraint satisfaction problem* (CSP) [15] consists of a set of constraints $\{c_1, \ldots, c_k\}$ over a set of variables $\{x_1, \ldots, x_n\}$, where each variable x_i can only take values from its domain $D_{init}(x_i)$, a set of integers. Solving a CSP requires finding a value for each variable from its domain so that no constraint is violated, *i.e.* all constraints are satisfied.

We adopt the notion of *propagation solver* from Schulte and Stuckey [14]. A *propagator* f is a monotonically decreasing function from domains to domains. A *propagation solver* for a set of propagators F and current domain D, $solv(F, D)$, repeatedly applies all the propagators in F starting from domain D until there is no further change in resulting domain. We say two sets of propagators F_1 and F_2 are *equivalent* if $solv(F_1, D) = solv(F_2, D)$ for all $D \sqsubseteq D_{init}$.

Define the *generalized arc consistent propagator* (or equivalently the *domain consistent* [14] propagator) for a constraint c as

$$dom(c)(D)(x) = \{\theta(x) \mid \theta \in D \text{ and } \theta \in c (\text{that is } \theta \text{ is a solution of } c)\}$$

3 Box Constraint Collections

Formally, an *adhoc constraint* c over variables $x_1, \ldots, x_n$ is a set of valuations in D_{init} representing the solutions of c. Adhoc constraints are usually implemented as tabled constraints by listing all the solutions or nogoods, incurring space and time overhead.

Example 1. The adhoc constraint c_{adhoc} over x and y for $D_{init}(x) = D_{init}(y) = \{1, 2, 3, 4, 5\}$ shown in Fig. 1(a) can be represented by the set of solutions { $(1,3)$, $(2,2)$, $(2,3)$, $(3,1)$, $(3,2)$, $(3,4)$, $(3,5)$, $(5,3)$ } or the set of nogoods { $(1,1)$, $(1,2)$, $(1,4)$, $(1,5)$, $(2,1)$, $(2,4)$, $(2,5)$, $(4,1)$, $(4,2)$, $(4,3)$, $(4,4)$, $(4,5)$, $(5,1)$, $(5,2)$, $(5,4)$, $(5,5)$ }.

Often we represent a constraint in an adhoc manner because it is difficult (or unwieldy) to describe it using a symbolic expression. However, it may be easier to find symbolic expressions if we examine part of the solution space. Therefore, we propose representing an adhoc constraint c_{adhoc} with a set of simple constraints in DNF. The idea is similar to the use of Karnaugh-Veitch-diagrams [13] for finding prime implicants.

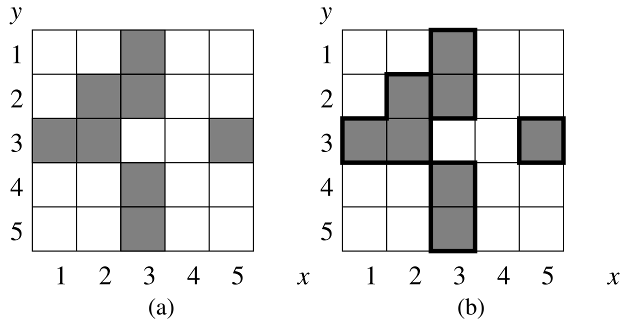

Fig. 1. (a) An adhoc constraint c_{adhoc}, and (b) broken into a box constraint collection

The core idea is to use a disjunction of constraints as "tiles" to cover the solution space of an adhoc constraint. By carefully choosing the shapes of the tiles we can achieve domain consistency using constructive disjunction. Triangles and rectangular boxes are good tile shapes for filling grids.

A *box* $B = \prod_{j=1}^{n} \left[l_j^B..u_j^B\right]$ is an n-dimensional hyper-cube, where $\left[l_j^B..u_j^B\right]$ is a *interval* of integers l_j^B and u_j^B. If $c(x_1, \ldots, x_n)$ is a constraint on variables $x_1, \ldots, x_n$, then $\bigwedge_{j=1}^{n} l_j^B \leq x_j \leq u_j^B \ \wedge \ c(x_1, \ldots, x_n)$ is a *box constraint*, which we write as $B \Rightarrow c$. We restrict the form of constraints c to two *templates*. Either c is $true$ and then $B \Rightarrow c$ is simply the box B, or c is of the form $\sum_{j=1}^{n} a_j x_j \leq a_0$, then we call $B \Rightarrow c$ a *triangle*. A *box constraint collection* (BCC) is simply a disjunction of box constraints.

We represent an adhoc constraint c_{adhoc} over variables $x_1, \ldots, x_n$ as a collection of m box constraints

$$c_{adhoc}(x_1, \ldots, x_n) \equiv \bigvee_{i=1}^{m} B_i \Rightarrow c_i(x_1, \ldots, x_n). \tag{1}$$

Example 2. A box constraint collection representation of the adhoc constraint c_{adhoc} shown in Fig. 1(a) is

$$\begin{array}{c}[3..3] \times [4..5] \Rightarrow true \vee [1..2] \times [2..3] \Rightarrow x + y \geq 4 \\ \vee\ [5..5] \times [3..3] \Rightarrow true \vee [3..3] \times [1..2] \Rightarrow true\end{array}$$

The box constraint $[1..2] \times [2..3] \Rightarrow x + y \geq 4$ represents the conjunction $1 \leq x \leq 2 \wedge 2 \leq y \leq 3 \wedge x + y \geq 4$. The BCC representation for c_{adhoc} is shown in Fig. 1(b).

Representing a constraint using a box constraint collection is more compact than a set of solutions. However, disjunctive constraints do not usually propagate as effectively as other representations. But disjunctions of box constraints can be propagated effectively, achieving generalized arc consistency.

Lemma 1. *If each constraint c_i in (1) is implemented by generalized arc consistent propagator $dom(c_i)$, then using constructive disjunction [16] on this representation achieves generalized arc consistency for c_{adhoc}.*

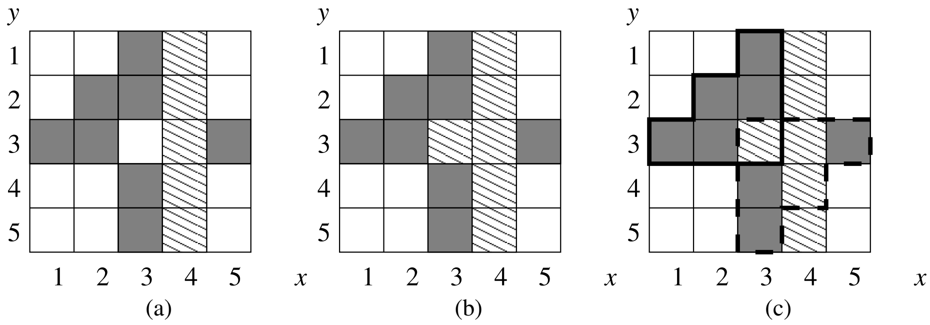

Fig. 2. Freeing the representation by adding "dont care" tuples to c_{adhoc} for (a) $x = 4$ and (b) $x = 3 \wedge y = 3$, and (c) the resulting box constraint collection c_{tri}.

4 Separable Nogoods

We can improve the description of an adhoc constraint by a box constraint collection by determining parts of the constraint which can be represented separately without losing generalized arc consistency of the resulting set of propagators.

Example 3. Consider c_{adhoc} defined in Fig. 1(a). Since $x \neq 4$ is implied by c_{adhoc} we can extract this as a separate constraint, we are then free to model the remainder of c_{adhoc} by filling in some boxes in the $x = 4$ column and this will not change the propagation behavior. Fig. 2(a) shows c_{adhoc} with "dont care" annotations in the $x = 4$ column.

Similarly the remaining nogood (3,3) is such that unless x (y) is assigned to 3, it will not remove the value 3 from the domain of y (x). In this situation, we can represent this nogood with an extra constraint $\neg(x = 3 \wedge y = 3)$ without changing the propagation behavior. Fig. 2(b) shows c_{adhoc} with "dont care" annotation at $x = 3 \wedge y = 3$.

Note that now we can represent c_{adhoc} by the conjunction of constraints $x \neq 4$, $\neg(x = 3 \wedge y = 3)$ and c_{tri} defined as the box constraint collection

$$[1..3] \times [1..3] \Rightarrow x + y \geq 4 \quad \vee \quad [3..5] \times [3..5] \Rightarrow x + y \leq 8$$

We obtain the same propagation behavior. The representation is smaller in terms of the number of box constraints and propagates more efficiently.

These two observations for separability of nogoods in the above example can be formalized as follows.

Lemma 2. *Let c be an adhoc constraint such that $c \rightarrow x \neq d$ for some $x \in vars(c)$ and $d \in D_{init}(x)$. Let S be a set of solutions for $vars(c)$ where $x = d$. Then $\{dom(x \neq d), dom(c \cup S)\}$ and $\{dom(c)\}$ are equivalent.*

Lemma 3. *Let c be an adhoc constraint on variables $(x_1, \ldots, x_n)$ with nogood $\theta \notin C$ such that there are no other nogoods $\theta' \notin c$ and $1 \leq i \leq n$ where $\theta(x_i) = \theta'(x_i)$. Then $\{dom((x_1, \ldots, x_n) \neq (\theta(x_1), \ldots, \theta(x_n))), dom(c \cup \{\theta\})\}$ and $\{dom(c)\}$ are equivalent.*

5 Building Box Constraint Collections

In this section, we describe a greedy algorithm, bccFinder, which computes a compact box constraint collection for a given adhoc n-ary constraint c_{adhoc} with solutions $solutions$ and nogoods $nogoods$. Before we find the set of box constraints, we remove the set of separable nogoods from c_{adhoc}, by adding extra constraints as discussed in Section 4. This leaves a description of the constraint involving three kinds of tuples: solutions, nogoods, and "dont cares" which may be included or not since they will be removed by other constraints. Then, we repeatedly find box constraints for the remaining uncovered solutions. A valuation θ is *covered* by the constraint c if $\theta \in c$; otherwise, it is *uncovered*. Fig. 3 shows the pseudo-code of bccFinder.

Since we would like to reduce the number of box constraints in the collection, we want each box constraint $B \Rightarrow c$ to cover as many uncovered solutions as possible. Although finding the optimal collection is in practice infeasible, we can find a relatively large box B by greedily growing one, until we cannot find any corresponding c, where c is an instantiation of one of our templates ct. For the code shown, ct is always of the form of $\sum_{j=1}^{n} a_j x_j \leq a_0$ since such constraints are straightforward to find, and have generalized arc consistency propagators which are efficiently computable [14].

To find $B \Rightarrow c$, we randomly pick an uncovered solution and put it into the (unit) box B and initialize C, the constraints on the coefficients a_j, to $true$. As a result, each a_j is unconstrained. Then, we iteratively try to enlarge B in each dimension j. We first reduce the lower bound l_j^B until either the lower bound of x_j is reached, or no enlargement is possible. Then we try to increase the upper bound u_j^B.

Let B' be the enlarged B. The procedure update is called so that for each valuation $\theta \in B' - B$ of the form $\theta \equiv \{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$ we either (a) add the constraint $\sum_{j=1}^{n} a_j d_j \leq a_0$ if $\theta \in solutions$ to ensure θ is included in the box constraint, or (b) add the constraint $\sum_{j=1}^{n} a_j d_j > a_0$ if $\theta \in nogoods$. This update procedure is an exact version of an algorithm by Anthony and Frisch [3] for constraint induction.

If the constraints are satisfiable, there exist values for a_j and we continue expanding the box. If the constraints are unsatisfiable, we first remove all the constraints added in the last expansion and try expanding in a different direction. Eventually every expansion leads to failure (or we have covered the entire space). At this stage we simply choose a value for each a_j that satisfies the current constraints. In our implementation, we solve for a_j's with the SICStus Prolog `clp(Q)` constraint-solving library [1].

We have created a single box constraint. We add this to our collection, and move all the solutions covered by this box constraint into the "dont care" category. This continues until there are no solutions remaining (which are not "dont care"). We then simplify the resulting collection if possible, by replacing $\sum_{j=1}^{n} a_j x_j \leq a_0$ by $true$ if $B \rightarrow \sum_{j=1}^{n} a_j x_j \leq a_0$ and removing box constraints which are subsumed by other box constraints.

A box constraint collection with only boxes ($B \Rightarrow true$) can be found similarly, except that B stops expanding along a particular dimension if B' contains at least one nogood.

```
bccFinder(n,solutions,nogoods)
  c_B := false
  c_S := constraints for separable nogoods
  separable := nogoods of c_S
  nogoods := nogoods − separable
  while (∃θ ∈ solutions)
    B := a unit box equal to θ
    C := true
    for j := 1 to n
      while (l_j^B > min(D_init(x_j)))
        B' := B with l_j^{B'} = l_j^B − 1
        C' := update(C, B', B, solutions, nogoods)
        if (C' is not satisfiable) break
        B := B'
        C := C'
        solutions := solutions − B
      endwhile
      while (u_j^B < max(D_init(x_j)))
        B' := B with u_j^{B'} = u_j^B + 1
        C' := update(C, B', B, solutions, nogoods)
        if (C' is not satisfiable) break
        B := B'
        C := C'
        solutions := solutions − B
      endwhile
    endfor
    let φ be a solution of C
    c_B := c_B ∨ (B ⇒ Σ_{j=1}^n φ(a_j)x_j ≤ φ(a_0))
  endwhile
  simplify c_B
  return c_B ∧ c_S
```

```
update(C,B',B,solutions,nogoods)
  for each θ ∈ B' − B
    if θ ∈ solutions
      C := C ∧ Σ_{j=1}^n a_j θ(x_j) ≤ a_0
    elseif θ ∈ nogoods
      C := C ∧ Σ_{j=1}^n a_j θ(x_j) > a_0
    endif
  endfor
  if C is satisfiable
    return C
  else return false
```

Fig. 3. Pseudo-code of bccFinder

The box constraint collection being returned is always equivalent to the given adhoc constraint, because when bccFinder terminates, all solutions will be covered, while all nogoods will remain uncovered.

The bccFinder algorithm always terminates because each while loop removes at least one valuation (θ) from $solutions$.

Although in worst case `clp(Q)` takes exponential time to solve for the coefficients of ct, our experiments confirm that our bccFinder algorithm is capable of returning a box constraint collection for an adhoc constraint in a reasonable amount of time.

There are many possible improvements to the simple algorithm shown here. For example we should not examine an expansion where all the valuations in $B' - B$ are in $nogoods$, and we should find large rectangular boxes first before starting the box expansion.

Table 1. The (partial) indexical grammar and its semantics in SICStus Prolog

Rule	Semantics
$r \rightarrow \texttt{dom}(y)$	y_σ
$r \rightarrow t_1..t_2$	$\{i \in \mathcal{Z} : t_{1_\sigma} \leq i \leq t_{2_\sigma}\}$
$r \rightarrow \{t_1, \ldots, t_n\}$	$\{t_{1_\sigma}, \ldots, t_{n_\sigma}\}$
$r \rightarrow r_1 \texttt{ /\textbackslash } r_2$	$r_{1_\sigma} \cap r_{2_\sigma}$
$r \rightarrow r_1 \texttt{ \textbackslash/ } r_2$	$r_{1_\sigma} \cup r_{2_\sigma}$
$r \rightarrow r_1 \texttt{ ? } r_2$	$\emptyset$ if $r_{1_\sigma} = \emptyset$; r_{2_σ} otherwise
$t \rightarrow$ integer	t
$t \rightarrow \texttt{inf}$	$-\infty$
$t \rightarrow \texttt{sup}$	$+\infty$
$t \rightarrow \texttt{min}(y)$	minimum value of y_σ
$t \rightarrow \texttt{max}(y)$	maximum value of y_σ
$t \rightarrow t_1 + t_2$	$t_{1_\sigma} + t_{2_\sigma}$
$t \rightarrow t_1 - t_2$	$t_{1_\sigma} - t_{2_\sigma}$

6 Compilation of box constraint collection

In this section, we will explain how a box constraint collection can be compiled into indexicals. The constraint system FD [7, 16] is based on domain constraints and functional rules called *indexicals*. Indexicals provide an efficient approach to implementing propagators for constraints.

A *domain constraint* is an expression $x \in I$, where I is a finite set of integers. A *store* σ is a set of domain constraints. The expression x_σ denotes the intersection $I_1 \cap \ldots \cap I_n$ for all constraints $x \in I_k$ in σ, where $1 \leq k \leq n$. If σ does not contain a constraint $x \in I$, x_σ is the set $\mathcal{Z}$ of integers. A variable x is *determined in* σ if x_σ is a singleton set.

An *indexical* has the form $x \texttt{ in } r$, where r is a *range* generated by r in Table 1. The *value* of $x \texttt{ in } r$ in σ is $x \in r_\sigma$, where r_σ is the value of r in σ, a set of integers. A range may consist of other ranges or terms. A *term* t is generated by t in Table 1. The value of t in σ, t_σ, is an integer. Table 1 summarizes how the values of r_σ and t_σ are computed.

6.1 Basic Compilation

We illustrate the compilation process with the following example.

Example 4. The representation of c_{tri} from Example 3 is a disjunction of two box constraints

$$[1..3] \times [1..3] \Rightarrow x + y \geq 4 \qquad (2)$$

$$\vee\ [3..5] \times [3..5] \Rightarrow x + y \leq 8 \qquad (3)$$

The indexicals[3] for (2) and (3) are respectively

[3] The syntax of SICStus Prolog, shown in teletype font, requires variables to be in upper case. Upper and lower case variables of the same name should be understood interchangeably.

```
X in ((4-max(Y))..3)        X in (3..(8-min(Y)))
Y in ((4-max(X))..3)        Y in (3..(8-min(X)))
```

These maintain generalized arc consistency [14].

We can create an indexical for `X` for the box constraint collection by combining these indexical rules as follows:

```
Y13 in {0} \/ (dom(Y) /\ (1..3))
Y35 in {6} \/ (dom(Y) /\ (3..5))
X in ((dom(Y) /\ (1..3)) ? ((4-max(Y13))..3)) \/
     ((dom(Y) /\ (3..5)) ? (3..(8-min(Y35))))
```

`Y13` records the maximum value of `Y` in the interval $[1..3]$. The additional value 0 is added to the domain of `Y13` to ensure it is always non-empty (and thus does not cause failure). We call this additional value a *dummy value* and the constraint between `Y` and `Y13` a *confinement constraint*. Similarly `Y35` records the minimum value of `Y` in the interval $[3..5]$. The rule for `X` joins the constraints together, using the `Y13` or `Y35` to give the appropriate value of `Y` for the box of interest.

We can automatically map the indexical expressions for constraints $c_i(x_1, \ldots, x_n)$ to create indexical expression for a disjunction of box constraints $\vee_{i=1}^{m} B_i \Rightarrow c_i(x_1, \ldots, x_n)$ such that if each indexical for $c_i(x_1, \ldots, x_n)$ maintains generalized arc consistency, then so does this indexical.

Let $B_i = [a_{i1}..b_{i1}] \times \cdots \times [a_{in}..b_{in}]$ then define the indexicals

$$\texttt{Max}_{ij} \ \texttt{in} \ \{a_{ij} - 1\} \ \texttt{\textbackslash/} \ \texttt{(dom(X}_j\texttt{)} \ \texttt{/\textbackslash} \ \texttt{(}a_{ij}..b_{ij}\texttt{))}$$
$$\texttt{Min}_{ij} \ \texttt{in} \ \{b_{ij} + 1\} \ \texttt{\textbackslash/} \ \texttt{(dom(X}_j\texttt{)} \ \texttt{/\textbackslash} \ \texttt{(}a_{ij}..b_{ij}\texttt{))}$$

$\texttt{Min}_{ij}$ and $\texttt{Max}_{ij}$ are called the *confinement variables* of $\texttt{X}_j$ over B_i. The indexical expression for $\texttt{X}_k$ for a single box constraint $B_i \Rightarrow c_i(x_1, \ldots, x_n)$ is then

$$\texttt{(dom(X}_1\texttt{)} \ \texttt{/\textbackslash} \ \texttt{(}a_{i1}..b_{i1}\texttt{))} \ \texttt{?} \ \cdots \ \texttt{?} \ \texttt{(dom(X}_n\texttt{)} \ \texttt{/\textbackslash} \ \texttt{(}a_{in}..b_{in}\texttt{))} \ \texttt{?} \ \texttt{(}r'_{ik} \ \texttt{/\textbackslash} \ \texttt{(}a_{ik}..b_{ik}\texttt{))}$$

where r'_{ik} is the indexical r_{ik} for $\texttt{X}_k$ and constraint $c_i(x_1, \ldots, x_n)$ with $\texttt{max(X}_j\texttt{)}$ replaced by $\texttt{max(Max}_{ij}\texttt{)}$, $\texttt{min(X}_j\texttt{)}$ replaced by $\texttt{min(Min}_{ij}\texttt{)}$ and $\texttt{dom(X}_j\texttt{)}$ replaced by $\texttt{dom(X}_j\texttt{)} \ \texttt{/\textbackslash} \ \texttt{(}a_{ij}..b_{ij}\texttt{)}$. We call each $\texttt{dom(X}_j\texttt{)} \ \texttt{/\textbackslash} \ \texttt{(}a_{ij}..b_{ij}\texttt{)}$ a *guard* for r'_{ik}.

The indexical expression for $\texttt{X}_k$ for the disjunction of box constraints $\vee_{i=1}^{m} B_i \Rightarrow c_i(x_1, \ldots, x_n)$ is obtained by unioning the expressions for each box constraint for $\texttt{X}_k$.

Theorem 1. *The indexical for box constraint collection*

$$c \equiv \bigvee_{i=1}^{m} (\bigwedge_{j=1}^{n} a_{ij} \leq x_j \leq b_{ij} \ \wedge \ c_i(x_1, \ldots, x_n))$$

achieves generalized arc consistency if each indexical for c_i achieves generalized arc consistency.

This guarantees that, by choosing the constraints c_i carefully, the box constraint collection of an adhoc constraint achieves generalized arc consistency.

Adding terms $t \rightarrow min(r)$ and $t \rightarrow max(r)$ to the indexical language would allow the expression of constructive disjunction of triangles without confinement variables. We conjecture that this would speed up the propagation markedly.

6.2 Optimizing Compilation

The basic compilation generates correct but inefficient indexicals, because there are many redundant operations. We can improve the computation of confinement variable domains, as illustrated by the following example.

Example 5. The confinement indexical

```
Y13 in {0} \/ (dom(Y) /\ (1..3)).
```

is invoked whenever the domain of `Y` is modified, and performs an expensive intersection operation. If we instead initialize the domain of `Y13` to `{0} \/ (1..3)` then we can replace this intersection. So we replace the single indexical by

```
Y13 in {0} \/ (1..3)
Y13 in {0} \/ dom(Y).
```

Furthermore once `dom(Y)` and `1..3` are disjoint, the domain of `Y13` cannot change. We can add (using SICStus Prologs extended indexicals) a check that removes the second indexical if `Y13 in {0}`.

Since set operations are expensive, a guard `dom(X) /\ (L..U) ?` r should be removed or replaced with a more efficient indexical operation whenever possible. We can remove the guard if `L..U` is the initial domain of `X`, or r becomes empty for any values in `dom(X)` outside `L..U`. In both situations the guard is redundant.

Example 6. Consider the indexical for `X` in Example 4. If `Y13` takes its dummy value 0, then `((4 - max(Y13))..3)` is the empty domain. Similarly for the other disjunct. Hence the following indexical is equivalent

```
X in ((4 - max(Y13))..3) \/ (3..(8-min(Y35)))
```

By suitably choosing the dummy values, all guards for indexicals `Inf..b` and `a..Sup` can be removed, where `a` and `b` are constants and `Inf` and `Sup` are terms involving `min(Y)` and `max(Y)` of other variables `Y`.

For the remaining guards, we can replace `dom(X) /\ (L..U)` with `min(X)..U` if `L` is the lower bound of the initial domain of `X`, because if its domain intersects `L..U`, the minimum value in its domain must be smaller than `U`. Similarly, we can replace a guard with `L..max(X)` if `U` is the upper bound of its initial domain.

Also, we can remove `L..U` from r `/\ (L..U)` if r is always inside the range.

Other optimization techniques include combining indexicals, removing confinement variables and rearranging the execution order of indexicals. However, due to space limitations, they will not be discussed further.

7 Experiments

In this section, we compare the efficiency of two BCC representations (box and triangle) and another approach to representing adhoc constraints in SICStus Prolog. We

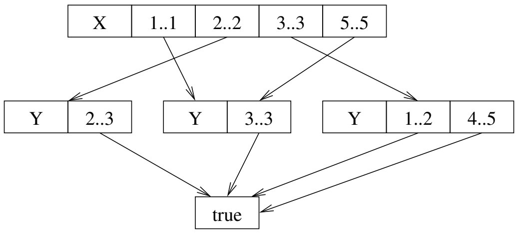

Fig. 4. A DAG representation of c_{adhoc} from Fig. 1.

implement the bccFinder algorithm and conduct the experiments using SICStus Prolog 3.9.1 on a Sun Blade 1000 with 2GB of memory (our largest benchmark consumes around only 20MB).

SICStus Prolog introduced in release 3.9.0 a new constraint, `case`, for encoding arbitrary n-ary adhoc constraints. To use the `case` constraint, users must first obtain a directed acyclic graph (DAG) from the list of solutions of the constraint. In the DAG, each node n is either the special leaf node $true$ or includes a variable x_n and a disjoint set of ranges $l_{nj}..u_{nj}$ each with a pointer to the next node n_j. A tuple θ satisfies the relation defined by the graph rooted by node n if n is the leaf node $true$, or there exists j such that $l_{nj} \leq \theta(x_n) \leq u_{nj}$ and θ satisfies the relation defined by graph rooted at n_j.[4]

The `case` constraint is a built-in global constraint equipped with an efficient filtering algorithm [6] to traverse the DAG for maintaining generalized arc consistency. In other words, the `case` technology consists of two parts: the DAG representation and the filtering algorithm. It is thus appropriate to compare the space and time tradeoffs of the BCC and the DAG (expanded into a tree[5]) representations when both are compiled into indexicals. We give also the results of using the `case` constraint for reference purposes. We envisage the possibility of an efficient filtering algorithm for maintaining generalized arc consistency of a BCC.

Example 7. A `case` constraint defining c_{adhoc} is given by the DAG show in Fig. 4. The indexical representation of the tree of the DAG is

```
X in ((dom(Y) /\ (2..3)) ? (2..2)) \/
     ((dom(Y) /\ (3..3)) ? ((1..1) \/ (5..5))) \/
     ((dom(Y) /\ ((1..2)\/(4..5))) ? (3..3)),
Y in ((dom(X) /\ (1..1)) ? (3..3)) \/
     ((dom(X) /\ (2..2)) ? (2..3)) \/
     ((dom(X) /\ (3..3)) ? ((1..2) \/ (4..5))) \/
     ((dom(X) /\ (5..5)) ? (3..3)).
```

We compare the propagation efficiency among box (indexicals for boxes only), $tri{-}box$ (indexicals for triangles and boxes), cas (DAG in the `case` constraint), and

[4] Actually the definition is slightly different but effectively equivalent.

[5] The filtering algorithm treats the DAG like a tree. The DAG representation is simply more compact.

Table 2. Performance comparisons on random 3-dimensional convex hull constraints

N	cas/dag		box		$tri{-}box$			$W=10$				$W=20$			
	B	gen	B	gen	T	B	gen	cas	dag	box	$tri{-}box$	cas	dag	box	$tri{-}box$
6025	448	1.14	294	27.25	52	18	15.70	37.85	71.12	62.73	35.22	45.05	117.43	111.26	67.28
4754	324	0.70	205	19.74	41	18	13.64	23.07	49.27	41.78	24.89	29.35	81.78	75.18	50.38
7086	385	1.80	287	42.71	57	23	15.62	37.52	60.31	59.98	36.67	37.87	94.20	109.72	75.64
7302	347	1.67	278	57.87	50	18	24.47	18.40	58.97	64.94	34.23	25.69	94.87	113.07	71.48
5598	339	1.18	262	29.20	47	24	14.16	35.76	50.17	56.68	28.97	37.08	88.31	98.60	61.56

Table 3. Performance comparisons on ternary non-linear inequality constraints

N	cas/dag		box		$tri{-}box$			$W=10$				$W=20$			
	B	gen	B	gen	T	B	gen	cas	dag	box	$tri{-}box$	cas	dag	box	$tri{-}box$
24591	225	8.48	224	186.81	4	32	101.74	7.99	21.93	25.15	9.70	12.76	36.65	43.67	16.54
20987	489	8.66	309	142.39	7	59	81.37	10.60	41.83	31.20	18.07	16.46	69.05	50.79	31.91
19671	471	8.76	215	122.36	8	47	90.83	11.56	39.86	24.34	17.34	21.46	65.05	44.35	31.92
17886	699	8.30	271	109.50	4	87	65.12	11.51	56.31	27.66	21.77	21.78	94.43	46.47	37.12
21938	499	7.32	238	134.07	15	10	94.38	10.50	44.12	24.94	10.19	17.04	73.34	42.87	21.51

dag (DAG in indexicals). The first three experiments simply test raw propagation. For each variable x in the constraint, we repeat M times picking a subset $S \subseteq D_{init}(x)$ where $|S| = W$, and adding the constraints $x \neq v$ for each $v \in S$. These constraint additions are then removed, and the next set S is selected.

We restrict our attention to benchmarks with structure, such as convex hull and non-linear inequality constraints, since BCC is designed for real-life constraints with meaning and thus reasonable patterns. Our experiment on random constraints show that BCC performs worse than `case` constraints, as expected.

In the first experiment, the adhoc constraint in each problem instance is defined by the convex hull generated by 15 random points chosen from the Cartesian product space of the variable domain $1..30$. Table 2 gives the results. N is the number of solutions, B and T are the number of boxes and triangles respectively, and gen is the generation time (in seconds). For cas and dag, we consider each path from root to leaf in the DAG as a box. We use the same DAG for both cas and dag, so that they share the same B and gen. The columns cas, dag, box and $tri{-}box$ report the execution time (in seconds) of the propagation test when $M = 5000$, and $W = 10$ or $W = 20$.

The second experiment deals with non-linear inequalities of the form $ax^3 + by^3 + cz^3 + dxyz + ex + fy + gz \leq h$, where the integer coefficients a to h are generated randomly from the $[-9..9]$. The initial domain for each variable is $1..30$. Results are summarized in Table 3.

We observe from the two experiments that both box and $tri{-}box$, in particular $tri{-}box$, use many fewer tiles than cas/dag for covering the same set of solutions. The representation of $tri{-}box$ is much more compact so that it is always faster than dag and box. The built-in filtering algorithm allows cas to be almost two times more efficient than $tri{-}box$ in some cases, despite the size disadvantage in representation.

Table 4. Performance comparisons on structured 3-dimensional polyhedron constraints

N	cas/dag		box		$tri\text{-}box$			$W=10$				$W=20$			
	B	gen	B	gen	T	B	gen	cas	dag	box	$tri\text{-}box$	cas	dag	box	$tri\text{-}box$
4225	437	1.15	437	39.17	8	0	68.24	34.26	57.39	73.92	4.22	38.44	84.51	112.68	8.56
4858	468	1.30	440	38.85	22	6	48.14	30.53	70.46	87.19	17.44	37.91	113.54	152.76	35.79
4526	240	1.16	30	10.59	28	2	27.26	25.17	30.38	11.69	17.47	30.52	50.48	22.29	34.50
2755	380	0.46	30	5.65	29	5	10.32	33.07	58.63	11.70	24.38	40.23	94.66	24.67	51.53
956	348	0.11	53	1.92	65	12	2.36	23.49	50.80	13.31	44.59	32.51	79.04	26.32	95.87

To study further the speed comparison, we conduct the third experiment on structured polyhedrons including dipyramids and sheared rectangular boxes, which are generated manually to investigate how the number of triangles and boxes affect the efficiency of BCC. Results are presented in Table 4.

We found that $tri\text{-}box$ fares very well against dag, since the representation advantage of $tri\text{-}box$ becomes more apparent in polyhedrons with good structures. The size of the cas/dag representation remains in the same order of magnitude as in the previous two experiments. The last three benchmarks exhibit even more regular structure so that they can be covered well also with boxes. In those cases, box is the fastest since the indexicals for implementing boxes lack the overhead of the confinement variables. This dependency on the number of boxes and triangles is more a result of the indexical implementation, but not the representation scheme.

Our last experiment is an application of our method to model induction [12], the outcome of which is a CSP consisting of only adhoc constraints. We study four different formulations (or models) of the Langford's problem (listed as "prob24" in CSPLib [11]): M_1, $M_1{}^*$, $M_1 \cap i(f^{-1}, M_2)$ and $M_1 \cap i(f^{-1}, M_2)^*$. Model M_1 is a hand-crafted model originally with symbolic constraints, but we turn the symbolic constraints into table form. Model $M_1{}^*$ is the same as M_1 except solutions of constraints of the same signature are intersected to form one constraint, which we call *constraint merging*. Model $M_1 \cap i(f^{-1}, M_2)$ is a model constructed from model M_1 plus constraints generated from model induction, while model $M_1 \cap i(f^{-1}, M_2)^*$ is model $M_1 \cap i(f^{-1}, M_2)$ with constraint merging. For more details see [12].

Table 5 summaries the results. The column $inst$ contain the problem instances (in terms of problem size) and $model$ contains the CSP models. Besides giving the number of unary and binary constraints in the u and b columns respectively, we give also the number of *distinct* unary and binary constraints that are learned in columns u_d and b_d. Two constraints are distinct if they have different sets of nogoods. N is the total number of solutions for the problem, while columns cas, dag, box and $tri\text{-}box$ give the CPU time in seconds to search for all solutions. Variables are chosen using the first-fail heuristic. It is *important* to note that the search trees of the same problem using different constraint representations are the same since generalized arc consistency is enforced in all cases.

In this application, $tri\text{-}box$ is significantly better than even cas for models M_1, $M_1{}^*$, and $M_1 \cap i(f^{-1}, M_2)$. It is because all constraints in the hand-crafted model M_1 are disequality constraints of the form $x \neq y+k$ for different k. Such constraints have a

Table 5. Performance comparisons on different models of Langford's Problem

$inst$	$model$	u	b	u_d	b_d	N	cas	dag	box	$tri\text{-}box$
(3,9)	M_1	0	369	0	10	6	4.56	4.88	7.86	0.51
	M_1*	0	351	0	10	6	4.25	4.51	7.21	0.48
	$M_1 \cap i(f^{-1}, M_2)$	9	720	9	361	6	5.85	9.63	12.39	4.34
	$M_1 \cap i(f^{-1}, M_2)$*	9	351	9	342	6	2.75	6.33	6.97	4.02
(3,10)	M_1	0	455	0	11	10	19.09	20.92	34.12	1.97
	M_1*	0	435	0	11	10	18.06	19.40	31.41	1.86
	$M_1 \cap i(f^{-1}, M_2)$	10	890	10	446	10	24.49	42.08	54.63	16.62
	$M_1 \cap i(f^{-1}, M_2)$*	10	435	10	425	10	11.43	28.19	31.46	15.52
(3,11)	M_1	0	550	0	12	0	101.78	110.87	182.27	9.59
	M_1*	0	528	0	12	0	95.82	102.92	168.50	9.01
	$M_1 \cap i(f^{-1}, M_2)$	11	1078	11	540	0	144.05	241.05	312.26	92.00
	$M_1 \cap i(f^{-1}, M_2)$*	11	528	11	517	0	60.56	151.29	167.48	82.38

high percentage of connected solutions, allowing the covering of the solutions by only a few triangles and boxes. In M_1^* this connectedness and structure are destroyed by constraint merging which removes some solutions from the constraint. In $M_1 \cap i(f^{-1}, M_2)$, constraints generated from model induction are not as structured, but model M_1 is still the backbone. The representation advantage of $tri\text{-}box$ degrades for $M_1 \cap i(f^{-1}, M_2)^*$ since the original constraints in M_1 are merged with the unstructured constraints from model induction.

8 Concluding Remarks

We have proposed a new language-independent representation, the box constraint collection, for adhoc constraints. With constructive disjunction, our new representation achieves generalized arc consistency, if all constraints inside the collection do. We have developed a greedy algorithm, bccFinder, to compute the box constraint collection of an adhoc constraint. It creates simple and compact representations of adhoc constraints, in a reasonable amount of time. We have shown how to implement box constraint collections as indexicals, and illustrated there efficient propagation on a number of examples. They are significantly more efficient than the DAG representation implemented by indexicals (dag).

Box constraint collections can be implemented in other ways than using indexicals. For binary constraints represented as boxes only (no triangles), Barták [5] gives an efficient arc consistency algorithm. For the more general case, an implementation similar to the `case` constraint seems quite plausible.

We conjecture that the difference in performance between cas and dag is mainly because the cas implementation propagates on all variables simultaneously, while the indexical representation runs each indexical separately. Worse, in the indexical representations (dag, box, and $tri\text{-}box$), when the indexical reduces the domain of x_i because of a change in x_j, then all the indexicals are re-executed since x_i has changed. But this re-execution can never find new information. The re-execution does not occur using `case` (according to our limited understanding). An internal implementation of the BCC constraint (like the `case` constraint) could avoid these overheads, and should lead to similar speedups (dag/cas) over the indexical representation.

We restrict our experiments to binary and ternary adhoc constraints. The BCC idea works for n-ary constraints in general, where $n > 0$. The bccFinder algorithm, however, needs improvement to be practical on higher dimensional constraints. This is an interesting topic for further study.

References

1. *Sicstus Prolog 3.9.1 manual*.
2. S. Abdennadher and C. Rigotti. Automatic generation of propagation rules for finite domains. In *Principles and Practice of Constraint Programming*, pages 18–34, 2000.
3. S. Anthony and A.M. Frisch. Generating numerical literals during refinement. In *ILP97*, pages 61–76, 1997.
4. K.R. Apt and E. Monfroy. Automatic generation of constraint propagation algorithms for small finite domains. In *Principles and Practice of Constraint Programming*, pages 58–72, 1999.
5. R. Barták. Filtering algorithms for tabular constraints. In *Proceedings of Colloqium on Implementation of Constraint and Logic Programming Systems (CICLOPS 2001)*, pages 168–182, 2001.
6. N. Beldiceanu. Global constraints as graph properties on structured networks of elementary constraints of the same type. Technical Report T2000-01, SICS, 2000.
7. B. Carlson and M. Carlsson. Compiling and executing disjunctions of finite domain constraints. In *Int. Conf. on Logic Programming*, pages 117–132, 1995.
8. T.B.H. Dao, A. Lallouet, A. Legtchenko, and L. Martin. Indexical-based solver learning. In *Principles and Practice of Constraint Programming*, pages 541–555, September 2002.
9. D. Diaz and P. Codognet. Design and implementation of the GNU Prolog system. *Journal of Functional and Logic Programming*, 2001(6), 2001.
10. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
11. I. Gent and T. Walsh. CSPLib: A benchmark library for constraints. In *Principles and Practice of Constraint Programming*, pages 480–481, 1999. Available at http:/www-users.cs.york.ac.uk/ tw/csplib/.
12. Y.C. Law and J.H.M. Lee. Model induction: a new source of csp model redundancy. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-2002)*, pages 57–71, 2002.
13. M. M. Mano and C. R. Kime. *Logic and Computer Design Fundamentals*. Prentice Hall, second edition, 1999.
14. C. Schulte and P.J. Stuckey. When do bounds and domain propagation lead to the same search space. In *3rd Int. Conf. on Principles and Practice of Declarative Programming*, pages 115–126, 2001.
15. E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.
16. P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(FD). *Journal of Logic Programming*, 37(1–3):139–164, 1998.