Dynamic Variable Elimination During Propagation Solving

Christian Schulte
ICT, KTH – Royal Institute of Technology
Stockholm, Sweden
cschulte@kth.se

Peter J. Stuckey
National ICT Australia, Victoria Laboratory
University of Melbourne, Australia
pjs@cs.mu.oz.au

Abstract

Constraint propagation solvers interleave propagation (removing impossible values from variables domains) with search. Propagation is performed by executing propagators (removing values) implementing constraints (defining impossible values). In order to specify constraint problems with a propagation solver often many new intermediate variables need to be introduced. Each variable plays a role in calculating the value of some expression. But as search proceeds not all of these expressions will be of interest any longer, but the propagators implementing them will remain active. In this paper we show how we can analyse the propagation graph of the solver in linear time to determine intermediate variables that can be removed without effecting the result. Experiments show that applying this analysis can reduce the space and time requirements for constraint propagation on example problems.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Constraint and logic languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Constraints

General Terms Languages, Design, Experimentation

Keywords Constraint (logic) programming, constraint propagation, redundancy, analysis

1. Introduction

Finite domain propagation interleaved with search is a powerful approach to solving combinatorial problems. Here, finite domain refers to a finite set of values, typically a finite subset of the integers. The set of possible values for each variable in the problem is stored in its *domain*. A constraint is implemented by a *propagator* which considers the domains of the variables in the constraint, and removes values from the domain of a variable which could not take part in any solution of the constraint. Propagators are repeatedly applied until no new values can be removed. Then a choice is made, typically fixing a variable to one of the values in its domain, and with this new information the propagation process is repeated.

Since "most" of the work of the finite domain propagation is done near the leaves of the search tree, finite domain solvers take some pains to simplify the problem they are tackling as search proceeds. This involves: removing fixed variables from propagators involving them, and eliminating redundant propagators (propagators that will no longer remove any value from any variable). But there is more that can be done. In this paper we examine how to eliminate variables and propagators during runtime that will not cause any more useful propagation.

Example 1.1 (Disjunction) Consider placing squares so that they do not overlap. Given two squares of dimensions $w_1 \times h_1$ and $w_2 \times h_2$, placed with their left corner at (x_1, y_1) and (x_2, y_2) respectively, then the non-overlap constraint is expressed as:

$$\begin{array}{c} x_1 + w_1 \leq x_2 \vee x_2 + w_2 \leq x_1 \vee \\ y_1 + h_1 \leq y_2 \vee y_2 + h_2 \leq y_1 \end{array}$$

In a typical constraint programming system this would be expressed as a conjunction of reified constraints

$$\begin{array}{l} \exists b_1.\exists b_2.\exists b_3.\exists b_4. \\ \quad b_1 \Leftrightarrow x_1 + w_1 \leq x_2 \wedge b_2 \Leftrightarrow x_2 + w_2 \leq x_1 \wedge \\ \quad b_3 \Leftrightarrow y_1 + h_1 \leq y_2 \wedge b_4 \Leftrightarrow y_2 + h_2 \leq y_1 \wedge \\ \quad \mathrm{or}(b_1, b_2, b_3, b_4) \end{array}$$

where e.g. the reified constraint $b_1 \Leftrightarrow x_1 + w_1 \leq x_2$ indicates that b_1 is true (1) if the constraint $x_1 + w_1 \leq x_2$ holds, and false (0) if the negation $x_1 + w_1 > x_2$ holds.

Imagine that some stage later in the search we have that $h_1 = 3$, $y_1 \in [0 .. 4]$ and $y_2 \in [8 .. 10]$. Then the propagator $b_3 \Leftrightarrow y_1 + h_1 \leq y_2$ sets $b_3 = 1$ since the right hand side constraint must hold. The propagator then removes itself (it is now redundant and cannot add more information). Now the propagator $\mathrm{or}(b_1, b_2, b_3, b_4)$ detects it is redundant and removes itself. Assuming $h_2 \geq 0$ then the propagator $b_4 \Leftrightarrow y_2 + h_2 \leq y_1$ will set $b_4 = 0$ and remove itself. The other propagators will remain even though they cannot cause failure or ever change the domains of any original variable x_1, y_1, x_2 or y_2. □

Modeling in finite domain propagation systems will rarely involve building models where a variable occurs only once, except when it is an output variable, and clearly we do not wish to eliminate these. But once constraints are found to be redundant it is quite possible that such systems arise during search. One reason that these cases occur reasonably frequently is that, in order to simplify the construction of a finite domain solver, constraints are broken down into components. This introduces new variables that occur only twice: once for defining the value of an intermediate term, and once to constrain it.

Example 1.2 (Decomposition) The constraint $(x_1 - x_2)^2 + (y_1 - y_2)^2 \geq d$ requires the two points (x_1, y_1) and (x_2, y_2) to be at least distance $\sqrt{d}$ apart. This constraint is not directly available in most

PPDP'08, July 15–17, 2008, Valencia, Spain.

finite domain solvers, instead it is decomposed into the system:

$$\begin{aligned}
&\exists dxx.\exists dx.\exists dyy.\exists dy.\\
&\quad dx = x_1 - x_2 \wedge dxx = dx \times dx \wedge\\
&\quad dy = y_1 - y_2 \wedge dyy = dy \times dy \wedge\\
&\quad dxx + dyy \geq d
\end{aligned}$$

Imagine at some time the domains of the original variables are $x_1 \in [0 \,..\, 10]$, $x_2 \in [15 \,..\, 30]$, $y_1 \in [0 \,..\, 30]$, $y_2 \in [0 \,..\, 30]$, and $d = 25$. Then propagation will calculate the domains of the newly introduced variables as $dx \in [-30 \,..\, -5]$, $dxx \in [25 \,..\, 900]$, $dy \in [-30 \,..\, 30]$, $dyy \in [0 \,..\, 900]$. The inequality $dxx + dyy \geq d$ is redundant, and indeed the original constraint is redundant. But every further change in the original variables will cause the propagators for the new intermediate variables to be reexecuted for no purpose. □

Global constraints can also be implemented by decomposition, and in this case we can possibly build large chains of introduced variables that may not be useful for the lifetime of the global constraint.

Example 1.3 (Lexicographic order) The constraint for the strict lexicographic order $(x_1, \ldots, x_n) < (y_1, \ldots, y_n)$ can be encoded as $lt(1, x, y)$ where:

$$\begin{array}{ll}
lt(i, x, y) = \text{false} & i > n\\
lt(i, x, y) = (x_i < y_i) \vee (x_i = y_i \wedge lt(i+1, x, y) & \text{otherwise}
\end{array}$$

This is decomposed further to $l_i \Leftrightarrow x_i < y_i$, $e_i \Leftrightarrow x_i = y_i$, $t_i \Leftrightarrow e_i \wedge lt(i+1, q, y)$ and $lt(i, x, y) \Leftrightarrow l_i \vee t_i$.

Suppose l_j becomes true (because we detect $x_j < y_j$ holds). Then by propagation $lt(j, x, y)$ will be set true and the propagator for $lt(j, x, y) \Leftrightarrow l_k \vee t_j$ will be removed as redundant. Now t_j only occurs in one propagator and can be eliminated, which then allows the elimination of e_j and $lt(j+1, x, y)$. In fact all propagators for constraints with indices greater than j can be removed.

We can do the same if we determine that l_j and e_j are both false. □

There are also special cases of variable elimination that arise from optimization problems.

Example 1.4 (Linear inequalities) In a typical optimization problem, there is a variable representing the objective function, defined by some constraint. For example, to minimize $\sum_{i=1}^{n} a_i x_i$ we would define $y = \sum_{i=1}^{n} a_i x_i$ and minimize y. Now y only occurs once in the propagation engine, but when a new solution is found with value $y = d$ a new constraint is imposed globally that $y \leq d - 1$. In forward computation no constraints will be placed on y and indeed as soon as we have that all possible values of $\sum_{i=1}^{n} a_i x_i$ are less than d the propagator is removable since no other constraints will modify y (in forward computation).

In effect, we wish to treat the constraint like $\sum_{i=1}^{n} a_i x_i \leq d-1$, although it is implemented using the variable y. □

We present a runtime analysis to find such eliminable variables, and remove the useless propagators attached to them. We will see that this can lead to improved performance in terms of time and space.

But there is a further use of the analysis. The analysis will visit all the variables in the problem that may actually affect further computation. If there are variables that are not traversed from the search variables (variables that will be assigned during search), then they will not be modified further by the search. Unless they have no propagators remaining on them this indicates a modeling error. We return to this point in Section 4.

The paper is organized as follows. In the next section we define propagation-based constraint solving, and how it performs redundancy elimination. In Section 3 we define existential redundancy of propagators, and explain how it can be detected for common propagators. In Section 4 we define an algorithm for runtime analysis of the propagation graph that detects variables and propagators that can be safely removed without affecting future computation. In Section 5 we discuss some of the issues that arise in implementing the analysis in practice. In Section 6 we give experiments showing the overhead and effectiveness of the analysis on example benchmarks. In Section 7 we discuss related work, and finally conclude.

2. Propagation-based Constraint Solving

This section defines our terminology for the basic components of a constraint propagation engine. In this paper we restrict ourselves to finite domain integer constraint solving. Almost all the discussion applies to other forms of finite domain constraint solving such as for sets and multisets.

2.1 Propagation

Domains. A *domain* D is a complete mapping from a fixed (finite) set of variables $\mathcal{V}$ to finite sets of integers. A *false domain* D is a domain with $D(x) = \emptyset$ for some $x \in \mathcal{V}$. A variable $x \in \mathcal{V}$ is *fixed* by a domain D, if $|D(x)| = 1$. The *intersection* of domains D_1 and D_2, denoted $D_1 \sqcap D_2$, is defined by the domain $D(x) = D_1(x) \cap D_2(x)$ for all $x \in \mathcal{V}$.

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 $x \in \mathcal{V}$. A domain D_1 is stronger than (equal to) a domain D_2 w.r.t. variables V, denoted $D_1 \sqsubseteq_V D_2$ (resp. $D_1 =_V D_2$), if $D_1(x) \subseteq D_2(x)$ (resp. $D_1(x) = D_2(x)$) for all $x \in V$. We use the notation $-\{x\}$ to denote the variable set $\mathcal{V} - \{x\}$.

A range is a contiguous set of integers, we use *range* notation $[l \,..\, u]$ to denote the range $\{d \in \mathbb{Z} \mid l \leq d \leq u\}$ when l and u are integers. A domain is a *range domain* if $D(x)$ is a range for all $x \in \mathcal{V}$. Let $D' = \text{range}(D)$ be the smallest range domain containing D, that is, the unique domain $D'(x) = [\inf D(x) \,..\, \sup D(x)]$ for all $x \in \mathcal{V}$.

We shall be interested in the notion of a *starting domain*, which we denote D_{start}. The starting domain gives the initial values possible for each variable. It allows us to restrict attention to domains D such that $D \sqsubseteq D_{\text{start}}$.

Valuations and constraints. 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 variables appearing in a valuation. We define a valuation θ to be an element of a domain D, written $\theta \in D$, if $\theta(x_i) \in D(x_i)$ for all $x_i \in \text{vars}(\theta)$.

The *infimum* and *supremum* of an expression e with respect to a domain D are defined as $\inf_D e = \inf\{\theta(e) \mid \theta \in D\}$ and $\sup_D e = \sup\{\theta(e) \mid \theta \in D\}$.

We can map a valuation θ to a domain D_θ as follows

$$D_\theta(x) = \begin{cases} \{\theta(x)\} & x \in \text{vars}(\theta) \\ D_{\text{start}}(x) & \text{otherwise} \end{cases}$$

A *constraint* c over variables $x_1, \ldots, x_n$ is a set of valuations θ such that $\text{vars}(\theta) = \{x_1, \ldots, x_n\}$. We also define $\text{vars}(c) = \{x_1, \ldots, x_n\}$.

Propagators. We will *implement* a constraint c by a set of propagators $\text{prop}(c)$ that map domains to domains. A *propagator* f is a monotonically decreasing function from domains to domains: $f(D) \sqsubseteq D$, and $f(D_1) \sqsubseteq f(D_2)$ whenever $D_1 \sqsubseteq D_2$.

A propagator f is *correct* for a constraint c iff for all domains D

$$\{\theta \mid \theta \in D\} \cap c = \{\theta \mid \theta \in f(D)\} \cap c$$

This is a very weak restriction, for example the identity propagator is correct for all constraints c.

A set of propagators F is *checking* for a constraint c, iff for all valuations θ where $\text{vars}(\theta) = \text{vars}(c)$ the following holds: $f(D_\theta) =_{\text{vars}(\theta)} D_\theta$ for all $f \in F$, iff $\theta \in c$. That is, for any domain D_θ corresponding to a valuation on $\text{vars}(c)$, $f(D_\theta)$ is a fixpoint iff θ is a solution of c. We assume that $\text{prop}(c)$ is a set of propagators that is correct and checking for c.

The variables, $\text{vars}(f)$, of a propagator f are defined as $\{x \in \mathcal{V} \mid \exists D \sqsubseteq D_{\text{start}}, f(D)(x) \neq D(x)\} \cup \{x \in \mathcal{V} \mid \exists D_1, D_2 \sqsubseteq D_{\text{start}}, D_1 =_{-\{x\}} D_2, f(D_1) \neq_{-\{x\}} f(D_2)\}$ The set includes the variables that can change as a result of applying f, and the variables that can modify the result of f.

To simplify presentation, we will use $\text{props}(x)$ to denote the set of propagators f with $x \in \text{vars}(f)$.

Example 2.1 (Propagators) For the constraint $c \equiv x_1 \leq x_2 + 1$ the function f_1 defined by

$$\begin{array}{rcll} f_1(D)(x_1) & = & \{d \in D(x_1) \mid d \leq \sup_D x_2 + 1\} & \\ f_1(D)(x) & = & D(x) & x \neq x_1 \end{array}$$

is a correct propagator for c.

Its variables are x_1 whose domain can be modified by f_1 (the first part of the definition above) and x_2 which can cause the modification of the domain of x_1 (the second part of the definition above). So $\text{vars}(f_1) = \{x_1, x_2\}$.

Let $D_1(x_1) = \{1, 5, 8\}$ and $D_1(x_2) = \{1, 5\}$, then $f(D_1) = D_2$ where $D_2(x_1) = D_2(x_2) = \{1, 5\}$.

The propagator f_2 defined as

$$\begin{array}{rcll} f_2(D)(x_2) & = & \{d \in D(x_2) \mid d \geq \inf_D x_1 - 1\} & \\ f_2(D)(x) & = & D(x) & x \neq x_2 \end{array}$$

is another correct propagator for c. Again, $\text{vars}(f_2) = \{x_1, x_2\}$.

The set $\{f_1, f_2\}$ is checking for c. The domain $D_{\theta_1}(x_1) = \{2\}$ and $D_{\theta_1}(x_2) = \{2\}$ corresponding to the solution $\theta_1 = \{x_1 \mapsto 2, x_2 \mapsto 2\}$ of c is a fixpoint of both propagators. The non-solution domain $D_{\theta_2}(x_1) = \{2\}$ and $D_{\theta_2}(x_2) = \{0\}$ corresponding to the valuation $\theta_2 = \{x_1 \mapsto 2, x_2 \mapsto 0\}$ is not a fixpoint (of either propagator). □

Propagation Solving. A *propagation solver* $\text{solv}(F, D)$ for a set of propagators F and a domain D finds the greatest mutual fixpoint of all the propagators $f \in F$. In other words, $\text{solv}(F, D)$ returns a new domain defined by

$$\begin{array}{rcl} \text{solv}(F, D) & = & \text{gfp}(\lambda d.\, \text{iter}(F, d))(D) \\ \text{iter}(F, D) & = & \bigsqcap_{f \in F} f(D) \end{array}$$

where gfp denotes the greatest fixpoint w.r.t. $\sqsubseteq$ lifted to functions.

A constraint propagation system evaluates $\text{solv}(F, D)$ during backtracking search. We assume an execution model for solving a constraint problem with a set of constraints C and a starting domain D_{start} as follows. We execute the procedure $\mathsf{search}(\emptyset, F, D_{\text{start}}, SV)$ implementing depth-first search given in Figure 1 for an initial set of propagators $F = \cup_{c \in C} \text{prop}(c)$ on a set of search variables SV. It either returns false or a domain D representing a solution (or solutions) of C.

Note that the propagators are partitioned into two sets, the old propagators F_o and the new propagators F_n. The *incremental* propagation solver $\mathsf{isolv}(F_o, F_n, D)$ takes advantage of the fact that D is guaranteed to be a fixpoint of the old propagators F_o.

In this simple version of isolv a propagator f' is added back into the queue Q of propagators to be executed if one of its variables' domains has changed, and the choice of next propagator to execute given by choose is left unspecified. For more detailed discussion on how isolv is defined in practice see (Schulte and Stuckey 2004, 2008).

```
search($F_o, F_n, D, SV$)
  $D$ := isolv($F_o, F_n, D$)                % propagation
  if ($D$ is a false domain)
    return false
  if ($\exists x \in SV.|D(x)| > 1$)
    choose $\{c_1, \ldots, c_m\}$ where
      $C \wedge D \models c_1 \vee \cdots \vee c_m$     % search strategy
    for $i \in [1 .. m]$
      $D'$ := search($F_o \cup F_n, \text{prop}(c_i), D, SV$)
      if ($D' \neq$ false)
        return $D'$
    return false
  return $D$
```

Figure 1. Search procedure

```
isolv($F_o, F_n, D$)
  $F$ := $F_o \cup F_n$; $Q$ := $F_n$
  while ($Q \neq \emptyset$)
    $f$ := choose($Q$)                              % next prop to apply
    $Q$ := $Q - \{f\}$
    $D'$ := $f(D)$
    $V$ := $\{x \in \mathcal{V} \mid D(x) \neq D'(x)\}$          % modified vars
    $Q'$ := $\{f' \in F \mid \text{vars}(f') \cap V \neq \emptyset\}$ % props to reconsider
    $Q$ := $Q \cup Q'$
    $D$ := $D'$
  return $D$
```

Figure 2. Incremental propagation solver

Domain and Bounds Propagators. A consistency notion C gives a condition on domains with respect to constraints. A set of propagators F maintains C *consistency* for a constraint c, if for domain D where $f(D) = D, f \in F$ is always C consistent for c. Many propagators in practice are designed to maintain some form of consistency: usually domain or bounds. But note that many more do not.

The by far most successful consistency technique is *arc consistency* (Mackworth 1977), which ensures that for each binary constraint, every value in the domain of the first variable, has a supporting value in the domain of the second variable that satisfied the constraint. Arc consistency can be naturally extended to constraints of more than two variables to give *domain consistency* (also known as generalized arc consistency). A domain D is *domain consistent* for a constraint c if D is the least domain containing all solutions $\theta \in D$ of c. That is, there does not exist a domain $D' \sqsubset D$ such that $\theta \in D \wedge \theta \in c \rightarrow \theta \in D'$.

Define the *domain propagator* $\text{dom}(c)$, for a constraint c as

$$\begin{array}{rcll} \text{dom}(c)(D)(x) & = & \{\theta(x) \mid \theta \in D \wedge \theta \in c\} & x \in \text{vars}(c) \\ \text{dom}(c)(D)(x) & = & D(x) & \text{otherwise} \end{array}$$

The basis of bounds consistency is to relax the consistency requirement to apply only to the lower and upper bounds of the domain of each variable x. There are a number of different notions of bounds consistency (Choi et al. 2006), we give the two most common here.

A domain D is $\text{bounds}(\mathbb{Z})$ *consistent* for a constraint c, $\text{vars}(c) = \{x_1, \ldots, x_n\}$, if for each variable x_i, $1 \leq i \leq n$

and for each $d_i \in \{\inf_D x_i, \sup_D x_i\}$ there exist *integers* d_j with $\inf_D x_j \le d_j \le \sup_D x_j$, $1 \le j \le n, j \ne i$ such that $\theta = \{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$ is an *integer solution* of c.

A domain D is $\mathrm{bounds}(\mathbb{R})$ *consistent* for a constraint c , $\mathrm{vars}(c) = \{x_1, \ldots, x_n\}$, if for each variable $x_i, 1 \le i \le n$ and for each $d_i \in \{\inf_D x_i, \sup_D x_i\}$ there exist *real numbers* d_j with $\inf_D x_j \le d_j \le \sup_D x_j$, $1 \le j \le n, j \ne i$ such that $\theta = \{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$ is a *real solution* of c.

A $\mathrm{bounds}(\mathbb{Z})$ *propagator*, $\mathrm{zbnd}(c)$ for a constraint c ensures that $\mathrm{zbnd}(c)(D)$ is $\mathrm{bounds}(\mathbb{Z})$ consistent with c, while a $\mathrm{bounds}(\mathbb{R})$ *propagator*, $\mathrm{rbnd}(c)$ ensures $\mathrm{bounds}(\mathbb{R})$ consistency.

2.2 Redundancy Elimination

Most propagation-based solvers also determine which propagators are redundant and no longer can change the domains of variables. Redundant propagators can be removed from the solver.

A propagator f is *redundant* for domain D if $f(D') = D', \forall D' \sqsubseteq D$. Clearly $\mathrm{solv}(F \cup \{f\}, D') = \mathrm{solv}(F, D')$ if f is redundant for domain $D \sqsupseteq D'$. Redundancy is also known as entailment or subsumption.

Example 2.2 (Redundancy) Consider the domain propagator for the constraint $\mathrm{or}(b_1, b_2, b_3, b_4)$. If any of the Boolean variables b_i is set to 1 then the constraint holds and the propagator can be removed.

Consider the $\mathrm{bounds}(\mathbb{R})$ propagator for the constraint $x + y \ge z$. If at any stage $\inf_D x + \inf_D y \ge \sup_D z$ then the constraint is redundant. For example, when $D(x) = [25 \mathrel{..} 300]$, $D(y) = [0 \mathrel{..} 300]$, and $D(z) = [-100 \mathrel{..} 25]$, the propagator can be removed. □

It is also possible that a constraint may become independent of some of its variables. This means that any further change in the domain of that variable is irrelevant to the constraint.

A propagator f is *independent* of x for domain D if $f(D') \sqcap D'_x = f(D'_x)$ for all $D'_x \sqsubseteq D' \sqsubseteq D$ where $D' =_{-\{x\}} D'_x$ and D'_x is not a false domain. Note that if $|D(x)| = 1$ then x is automatically independent for all propagators f since $D' = D'_x$ as they only differ in x and are not false domains.

We can modify the variables of a propagator to remove those for which it becomes independent.

Example 2.3 (Independence) Consider the domain propagator f for the constraint $x = \min\{y_1, y_2, y_3\}$ and the domain $D(x) = [0 \mathrel{..} 10]$, $D(y_1) = [2 \mathrel{..} 10]$, $D(y_2) = [10 \mathrel{..} 25]$, and $D(y_3) = [0 \mathrel{..} 15]$. Then y_2 is never the sole minimum of the set on the right hand side.

Hence, changes in the domain of y_2 will not effect other variables through this propagator, and changes to the domains of the other variables will never effect y_2. Hence the propagator for this constraint can be effectively replaced by the domain propagator for $x = \min\{y_1, y_3\}$ without changing any future computation. □

Note that if propagator f is redundant for D then it is independent of all variables x for domain D. We can modify our propagation engine to take into account redundancy and independence by modifying the variables: remove those for which the propagator is independent. Let $\mathrm{independent}(f, D)$ be the set of variables in $\mathrm{vars}(f)$ for which f is independent for domain D. Then we can modify the incremental propagation solver of Figure 2 by adding the following line just before the calculation of V.

$$\mathrm{vars}(f) := \mathrm{vars}(f) - \mathrm{independent}(f, D')$$

In practice, the propagator f will be replaced by the propagation engine by a new one, which has a smaller set of variables. If $\mathrm{vars}(f)$ becomes empty then the propagator can be removed altogether (it must be redundant). In addition, the search procedure usually checks that at the end no propagators remain that are not redundant. In this way it has a proof that the solution (or solutions) described by the answer domain D actually satisfies all the constraints.

3. Variable elimination

A constraint problem is represented by an existentially quantified conjunction of primitive constraints. The primitive constraints are those that can be directly implemented by propagators.

3.1 Existential redundancy

A constraint c is *existentially redundant* for y at domain D if $D \models \exists y.c$. That is, for all valuations $\theta \in D$ of variables $\mathrm{vars}(c) - \{y\}$ there exists $d \in D(y)$ where $\theta \cup \{y \mapsto d\} \in c$.

Since propagators may be weaker than constraints, we can view the condition on propagators as: A propagator f is *existentially redundant* for y at domain D if $\mathrm{solv}(\{f\}, D') =_{-\{y\}} D'$ for each $D' \sqsubseteq D$ where $D(y) = D'(y)$. That is for all future domains D' which do not change $D(y)$, applying f (repeatedly) will not change any variable domain except that of y.

If c is existentially redundant for y at D then any correct propagator f for c is existentially redundant for y at D.

Lemma 3.1 *Suppose c is existentially redundant for y at D, and f is a correct propagator for c. Then f is existentially redundant for y at D.*

Proof: Let $\mathrm{vars}(c) = \{y\} \cup V$. We have that $D \models \exists y.c$. Hence, for every valuation $\theta \in D$, θ is a solution of $\exists y.c$ and can be extended to a solution $\theta_y \in D$ where $\theta(x) = \theta_y(x)$ for all $x \in V$.

Consider an arbitrary valuation $\theta \in D'$ where $D' \sqsubseteq D$. Then $\theta_y \in D'$ since no other propagator changes the domain of y and θ_y is a solution of c so the value of $\theta_y(y)$ could not be removed by f as f is correct for c. Since $\theta_y \in D'$ and it is a solution of c we have that $\theta_y \in f(D')$. Since this holds for arbitrary θ we have that $\mathrm{solv}(\{f\}, D') =_{-\{y\}} D'$. □

Existential redundancy allows a very simple form of optimization of propagation. We can remove the propagator f and variable y from the propagation engine if f is existentially redundant for y, and y occurs in no other propagator, without affecting future computation. The key lemma for variable elimination is thus the following:

Lemma 3.2 (Key Lemma) *Let f be existentially redundant for y and domain D and $y \notin \mathrm{vars}(f')$ for all $f' \in F$. Then*

$$\mathrm{solv}(F \cup \{f\}, D') =_{-\{y\}} \mathrm{solv}(F, D')$$

for all $D' \sqsubseteq D$ if $D'(y) = D(y)$.

Proof: Examine $\mathrm{solv}(F \cup \{f\}, D') = D_n$. This is a sequence of applications of propagators from $F \cup \{f\}$ resulting in new domains, until a fixpoint is reached. Let the sequence be denoted:

$$D' = D_0 \ (f_0) \ D_1 \ (f_1) \ \cdots D_i \ (f_i) \ D_{i+1} \cdots D_n$$

Consider the sequence

$$D' = D'_0 \ (f'_0) \ D'_1 \ (f'_1) \ \cdots D'_m \ (f) \ D'_{m+1} \ (f) \ \cdots \ (f) \ D'_n$$

of this form where $[f'_i | i \in 0..m] = [f_i | i \in 0..n, f_i \ne f]$ where all the applications of propagator f have been moved to the end. Clearly $D'_i(y) = D'(y) = D(y)$ for $i \in 0..m$ since no $f' \in F$ involves y. Then since f is existentially redundant at D we have that $D'_n =_{-\{y\}} D'_m$ since f cannot modify the domains of any other variables.

Since none of the propagators $f' \in F$ make use of the domain of y and $D_{i+1} = f(D_i) =_{-\{y\}} D_i$ whenever $f_i = f$, we have that $D'_m =_{-\{y\}} D_n$. Since $f'(D_n) = D_n, \forall f' \in F$ by definition, we also have that $f'(D'_m) = D'_m$. Hence, $D'_m = \text{solv}(F, D')$. □

3.2 Detecting existential redundancy

Any original model that allows variable elimination is obviously a poor model. So we will not expect to see it occurring in original models. There is some possibility of this occurring though, if the model itself has been generated automatically.

But in any usual model each variable occurs at least twice, except perhaps some variables that are used only to create output, and these of course we do not wish to eliminate. So how do we find variables to eliminate?

As evidenced by the examples in the introduction, existential redundancy arises in two situations. It can arise by the elimination of propagators. Or, it can arise from the removal of independent variables leading to variables that occur exactly once in the remaining propagators.

The remaining requirement is that we can detect a propagator as existentially redundant. But how practical or frequent is this? The definition given in the previous section is clearly too expensive to check. Thankfully, existential redundancy is often easy to check.

All binary domain propagators are always existentially redundant for both variables involved.

Lemma 3.3 *Let f be a domain propagator for a binary constraint c where $\text{vars}(f) = \{x_1, x_2\}$ (so f enforces arc consistency). Then f is existentially redundant for x_1 and x_2 for domains D where $f(D) = D$.*

Proof: By definition, $f(D)$ is domain consistent with c and hence $\forall d_1 \in D(x_1), \exists d_2 \in D(x_2)$ where $\{x_1 \mapsto d_1, x_2 \mapsto d_2\}$ satisfies c. Hence $D \models \exists x_2.c$. By Lemma 3.1 f is existentially redundant for x_2 at D. The same applies for x_1. □

Many propagators for functional constraints can be checked for existential redundancy reasonably easily. A Boolean total function constraint, such as $y \Leftrightarrow (\vee_{i=1}^n x_i)$, $y \Leftrightarrow (\wedge_{i=1}^n x_i)$, $y \Leftrightarrow (\oplus_{i=1}^n x_i)$ (xor), $y \Leftrightarrow (x_1 \rightarrow x_2)$, and $y \Leftrightarrow \neg x$, is existentially redundant if there is a full domain on the function variable y.

Lemma 3.4 *Let f be a propagator for the total functional constraint $y = e(\bar{x})$, where y is Boolean then f is existentially redundant for y at D if $D(y) = \{0, 1\}$.*

A $\text{bounds}(\mathbb{Z})$ or $\text{bounds}(\mathbb{R})$ propagator for a functional constraint $y = e(\bar{x})$, where e is a total function, for example $y = a_0 + \sum_{i=1}^n a_i x_i$ and $y = \max_{i=1}^n x_i$, can often easily be checked for existential redundancy.

Lemma 3.5 *Let f be a* $\text{bounds}(\mathbb{Z})$ *or* $\text{bounds}(\mathbb{R})$ *propagator for the functional constraint $y = e(\bar{x})$ where e is a total function and $D(y) \supseteq [\inf_D e(\bar{x}) \,..\, \sup_D e(\bar{x})]$, then f is existentially redundant for y at D.*

Note that the result above holds trivially from Lemma 3.1. The usefulness of the above lemma is that bounds propagators typically calculate the value of the expressions $\inf_D e(\bar{x})$ and $\sup_D e(\bar{x})$, or some weakening of them, in order to execute the propagator. For example the $\text{bounds}(\mathbb{R})$ propagator for $y = 3x_1 + 10x_2 + 19x_3$ will calculate $3 \inf_D x_1 + 10 \inf_D x_2 + 19 \inf_D x_3$ as well as $3 \sup_D x_1 + 10 \sup_D x_2 + 19 \sup_D x_3$ during propagation. Hence checking the existential redundancy is straightforward.

```
analyse($D$,$OV$,$SV$)
  for ($v \in \mathcal{V}$)
    interested[$v$] := no
    visited[$v$] := false
  for ($v \in OV \cup SV$)
    interested[$v$] := yes
  for ($v \in SV$)
    traverse($v$,$-$,$D$)

 traverse($v$,$g$,$D$)
  if (visited[$v$])
    return interested[$v$]
  if (interested[$v$] = no)
    interested[$v$] := maybe
  visited[$v$] := true
  for ($f \in \text{props}(v) - \{g\}$)
    if ($f$ is existentially redundant for $y \neq v$ at $D$)
      if (traverse($y$,$f$,$D$) $\neq$ no)
        interested[$v$] := true
        for ($v' \in \text{vars}(f) - \{v, y\}$)
          traverse($v', f, D$)
      else % not interested in $y$
        delete $f$ (remove $f$ from $\text{prop}(v')$ for $v' \in \text{vars}(f)$)
    else
      interested[$v$] := true
      for ($v' \in \text{vars}(f) - \{v\}$)
        traverse($v', f, D$)
  if (interested[$v$] = maybe)
    interested[$v$] := no
  return interested[$v$]
```

Figure 3. Dynamic analysis for eliminable variables

Propagators do not necessarily have to be "equational" to be existentially redundant. The domain propagator for $\sum_{i=1}^n a_i x_i \geq d$ is existentially redundant for x_j at D if

$$\sup_D a_j x_j \geq d - \sum_{i=1, i \neq j}^{n} \inf_D a_i x_i.$$

Since the propagator determines $\sum_{i=1}^n \inf_D a_i x_i$ in order to determine redundancy it is straightforward to extend it to check for existential redundancy.

Many propagators are unlikely to be existentially redundant unless they are almost redundant, that is, they will almost never propagate further. For example $\text{alldifferent}(x_1, \ldots, x_n)$ is existentially redundant for x_j if $D(x_i) \cap D(x_j) = \emptyset, 1 \leq i \neq j \leq n$ and some further conditions hold. At this point the alldifferent is almost redundant itself.

4. Dynamic analysis for variable elimination

We now give a simple linear time analysis algorithm for finding variables to eliminate. The analysis analyse shown in Figure 3 takes a current domain D, the set of output variables OV, and the set of search variables SV. The output variables cannot be eliminated as they are needed in the computed answer. The search variables cannot be eliminated as search will add new constraints on them.

The algorithm marks all output and search variables as being interesting, and then traverses each search variable in turn. The traverse function visits a variable v to determine if it can be eliminated and visits other variables reachable from this variable.

It returns if we are interested in the variable (that is, the variable cannot be eliminated). The traversal first checks that the variable has not already been visited. If the variable has already been visited the previous result is returned.

If the variable has not already been visited, its status is set to maybe meaning we are still determining whether it is interesting.

All propagators f attached to v are considered, except the propagator g by which v has been reached. If f is existentially redundant for some $y \neq v$ and we are not interested in y then the propagator f can be removed. Otherwise, all variables reachable through f are traversed.

If all propagators for v are removed then traverse will return that we are uninterested in v. The point of the maybe recording is to ensure correctness for loops. If we find a loop returning to v while determining the interest of v, then the answer will be yes.

Theorem 4.1 *Let F_0 be a set of propagators, D a domain, SV a set of variables, and OV a set of variables. Suppose after executing analyse(D, OV, SV) for F_0 we have that $F \subset F_0$ remain. Then* $\mathrm{solv}(F, D') =_{OV} \mathrm{solv}(F_0, D')$ *for all $D' \sqsubseteq D$ where $D' =_{-OV \cup SV} D$.*

Proof: Let $F' = F_0 - F$. The proof is by induction on elimination of propagators $f \in F'$. If traverse eliminates a propagator f then f must be existentially redundant for some y at D, such that all other occurrences of y are in eliminated propagators. Note that $y \notin OV \cup SV$.

We can order the propagators F' say $f_1, \ldots, f_n$ such that f_i is existentially redundant for y_i at D and y_i appears in no propagator in $F \cup \{f_{i+1}, \ldots f_n\}$. Note that this reverses the order of traversal of the propagators. Then using Lemma 3.2 we can show that $\mathrm{solv}(F \cup \{f_i, f_{i+1}, \ldots f_n\}, D') =_{-\{y_i\}} \mathrm{solv}(F \cup \{f_{i+1}, \ldots f_n\}, D')$ for all $D' \sqsubseteq D$ where $D'(y) = D(y)$.

By induction $\mathrm{solv}(F_0, D') =_{-\{y_1, \ldots, y_n\}} \mathrm{solv}(F', D')$ for all $D' \sqsubseteq D$ where $D(y_i) = D'(y_i), 1 \leq i \leq n$. Since $OV \cup SV \subseteq \mathcal{V} - \{y_1, \ldots, y_n\}$ the result holds. □

In order for the algorithm to be efficient, we do not wish to spend too much time checking if f is existentially redundant for some $y \neq v$ at D. For binary propagators, there is only one candidate variable. For most n-ary propagators this is still simple as there is only likely to be one variable that can be detected as existentially redundant, the variable being "equationally defined" by the propagator, e.g. y in $y = a_0 + \sum_{i=1}^{n} a_i x_i$

But some constraints can be detected as existentially redundant for multiple variables, for example propagators for $c \equiv \sum_{i=1}^{n} a_i x_i = d$ where $a_i \in \{-1, 1\}, 1 \leq i \leq n$. For a particular domain D it could be existentially redundant for any variable x_i, since this constraint can be read as equationally defining each variable, e.g. $x_j = d + \sum_{i=1, i \neq j}^{n} a_i x_i$ when $a_j = -1$, and $x_j = -d - \sum_{i=1, i \neq j}^{n} a_i x_i$ when $a_j = 1$.

Luckily for this constraint c and any domain D there is only a maximum of two candidates for which it can be existentially redundant at any domain D, and the remaining variables are fixed, and hence are independent of the propagator for c. Hence either only one variable is possibly existentially redundant, or the constraint is binary. In either case the traversal algorithm only has to visit one possible variable y (different from v).

Lemma 4.2 *Suppose the bounds propagator f for $\sum_{k=1}^{n} a_k x_k = d$ where $a_k \in \{-1, 1\}$ for $1 \leq k \leq n$ is existentially redundant for x_i and x_j at domain D. Then the remaining variables are fixed in D.*

Proof: Let

$$\begin{array}{rcl} L & = & \inf_D \sum_{k=1}^{n} a_k x_k - d \\ U & = & \sup_D \sum_{k=1}^{n} a_k x_k - d \end{array}$$

Assume for simplicity that a_i and a_j are positive, the other cases follow similarly.

Let $l_k = \inf_D x_k$ and $u_k = \sup_D x_k$. Since c is existentially redundant for x_i, both $a_i l_i \leq a_i u_i - U$ and $a_i u_i \geq a_i l_i - L$. Similarly, we have $a_j l_j \leq a_j u_j - U$ and $a_j u_j \geq a_j l_j - L$.

By definition $U_{ij} = \sup_D \sum_{k=1, k \neq i, k \neq j}^{n} a_i x_i - d = U - a_i u_i - a_j u_j \geq L - a_i l_i - a_j l_j = \inf_D \sum_{k=1, k \neq i, k \neq j}^{n} a_i x_i - d = L_{ij}$. From $U - a_i u_i - a_j u_j \geq L - a_i l_i - a_j l_j$ and $a_i u_i - U \geq a_i l_i$ we determine that $-a_j u_j \geq L - a_j l_j$, or equivalently $a_j l_j - a_j u_j \geq L$ and from $a_j u_j \geq a_j l_j - L$ we have $L \geq a_j l_j - a_j u_j$ hence $L = a_j l_j - a_j u_j$. Similar reasoning gives $U = a_j u_j - a_j l_j$, $L = a_i l_i - a_i u_i$ and $U = a_i u_i - a_i l_i$. Now $U_{ij} = U - a_i u_i - a_j u_j = (a_i u_i - a_i l_i) - a_i u_i - a_j u_j = -a_i l_i - a_j u_j$ and $L_{ij} = L - a_i l_i - a_j l_j = (a_j l_j - a_j u_j) - a_i l_i - a_j l_j = -a_i l_i - a_j u_j = U_{ij}$. Hence all variables $x_i, 1 \leq k \neq i \neq j \leq n$ must be fixed in D. □

There are constraints where there are multiple possible existentially redundant variables at a single domain. Consider $x_1 + x_2 + x_3 \geq 3$ with domain $D(x_1) = D(x_2) = D(x_3) = [0 \,..\, 10]$, then the propagator is existentially redundant for all variables at D. The algorithm above only considers one such variable, otherwise we have to backtrack undoing the marking, and try other possibilities. While this may find more variables to eliminate, it is certainly more expensive, and the cases of constraints which can have multiple possible existentially redundant variables for some domain are rare.

Under the assumption that the calculation of existential redundancy for propagator f is linear in the size of f, as in all the examples discussed above, and with an additional marking to ensure we do not visit the same propagator twice (see Section 5) the analysis is linear in the size of the problem.

Other uses of the analysis. The analysis takes into account the output variables OV which we never wish to eliminate since they will be needed for the final result, and the search variables SV which cannot be eliminated since we will add new propagators on these variables during search.

But what happens if the analyse algorithm never visits a variable v which is still involved in a propagator f? Then clearly no modification of the domains of the search variables can lead to a change in the domain of v. Hence the propagator f will never be executed. Since the propagator f is not known to be redundant the search procedure will return an answer without determining that the constraint that f implements is solved. Hence the program can return *wrong solutions*.

One can argue that we can check this simply by examining the remaining propagators, at the end of the search. If propagators remain, then the domain D may not encode only solutions. The difficulty with this is that the search may never return an answer because the fact that propagator f is incorrectly modeled means that not enough domain reduction may occur and the search may be stuck in an effectively infinite search space.

Hence whenever we run the analysis we should also check that no variable remains unvisited, unless it is involved in no propagators. If this is the case we should immediately abort execution and report the modeling error. In this sense we should always run the analysis independent of its use for improving execution behavior, just to catch such modeling errors.

5. Implementation

The algorithm in Figure 3 has been implemented in Gecode, but most of the decisions made in the implementation should readily carry over to other constraint programming systems.

Implementing analysis. While treatment of variables is generic in the analysis algorithm, the way how propagators are traversed depends on the particular propagator. Propagators are implemented as objects in Gecode (as in most other systems, see for example the architecture underlying ILOG Solver (Puget and Leconte 1995)). Propagators provide methods for propagation, creation, deletion, and so on. For traversal, we add a `traverse` method that can be implemented for each individual propagator. Many `traverse` methods can be reused through inheritance, for example, the same method can be reused for all reified propagators.

Like variables, also propagators maintain a field `interested` which is initialized to maybe. The `traverse` method for a propagator f will check whether some of its variables are existentially redundant. In case it finds a variable y to be existentially redundant and traversal finds that nobody is interested in y, the `interested` field of the propagator is set to no. Note that the propagator is not immediately deleted (see the discussion below). If there is no existentially redundant variable, the propagator will traverse its variables $\mathrm{vars}(f)$ and set `interested` to yes.

The analysis starts from a solver state that hosts variables and propagators belonging to a single node in the search tree. After marking the search and output variables as interested, the traversal starts from the search variables. Search variables are available from the labellings (branchings) which the solver state maintains for search, whereas output variables are available from the implemented model. Only after traversal finishes, propagators with `interested` set to no are deleted, while propagators with `interested` set to maybe are reported as a possible modeling error (see Section 4). Propagators are not immediately deleted during traversal. As all propagators have to be inspected to find those with `interested` set to maybe, it is simpler to delete propagators in this separate inspection pass.

Interestingly, analysis does not increase the memory required for variables and propagators: both provide sufficient space to maintain the information for marking as interested or visited. The availability of sufficient space is due to the fact that the analysis is only run when the solver is at fixpoint, hence some fields that are used during propagation can be used during analysis and are restored after analysis finishes.

One critical aspect for the analysis to be efficient is that our experimental implementation uses recursion as directly available in C++. Recursion (which is mutual and can be very deep during the analysis) in C++ is not very efficient with respect to both memory and runtime. A production quality implementation of the analysis might use an explicit stack to manage traversal.

Gecode uses advisors in addition to propagators for achieving incremental propagation (Lagerkvist and Schulte 2007). An advisor is associated with a single propagator to provide information about how a particular variable changes during propagation. When traversing the propagators $\mathrm{prop}(x)$ attached to a variable x this has to be reflected in that also advisors can be attached to the variable x. In case of an attached advisor, traversal will immediately continue with the advisor's propagator.

Dynamic event sets and watched literals. Due to the more experimental nature of our analysis implementation, we make the assumption that for all variables $x \in \mathrm{vars}(f)$ for a propagator f there exists also an edge in the propagation graph from the variable x to the propagator f (the algorithm in Figure 3 uses $\mathrm{prop}(x)$ to find all propagators for a variable x).

Some propagators might not require propagation even though the domains of some of its variables change. That is, even though $x \in \mathrm{vars}(f)$ it can be the case that $f \notin \mathrm{prop}(x)$. In (Schulte and Stuckey 2008) this is used for dynamic event sets and in (Gent et al. 2006) this is used for watched literals to speed up propagation. A typical example is a propagator for Boolean disjunction $or([x_1, \ldots, x_n])$ where it is sufficient that f is included in at least two of the sets $\mathrm{prop}(x_i)$.

The problem with the analysis as presented is that during traversal, a variable x can be visited and no propagator in $\mathrm{prop}(x)$ is interested in x, even though there might be a propagator f with $x \in \mathrm{vars}(f)$. A simple solution that would incur some general overhead would be to register all propagators f with $f \in \mathrm{vars}(x)$ also in $\mathrm{prop}(x)$ where these extra entries are specially marked such that they are only considered for analysis but not for propagation.

To avoid any overhead during propagation, these additional entries into $\mathrm{prop}(x)$ could be entered just before analysis is performed. After analysis finishes, the entries are removed again. Provided the system maintains a list of all relevant propagators, this would be straightforward as the datastructures for $\mathrm{prop}(x)$ are designed to support dynamic addition and deletion. An additional advantage of this idea is that the extra entries are only available during analysis, hence propagation would not have to check whether an entry is just needed for traversal.

When to run the analysis. An important aspect is when the analysis should actually be run. The highest accuracy is obviously obtained by performing the analysis each time after a fixpoint has been computed (that is, directly after the **while**-loop in Figure 2 terminates). This might be too expensive (evaluation in Section 6 confirms that this is indeed the case). A remedy is to run the analysis only every n-th fixpoint computed to achieve a good compromise between accuracy and overhead of the analysis.

A different strategy is to run the analysis just before the recursive call to search in Figure 1. Depending on how search is implemented, this can be beneficial. Gecode uses recomputation and copying for implementing backtracking (similar to the model described in (Schulte 2002, Chapter 7)). Here, a copy of the search state will be created just before the recursive call to search. This copy will then be reused for recomputation several times. One promising strategy for analysis might be to run the analysis just before creating the copy that is reused several times: the effort spent in analysis once is reused every time a copy is created for recomputation. Moreover, creating a copy after several propagators have been deleted due to analysis might save space and time during copying. Again, the analysis can be run only every n-th time a copy is created for recomputation.

6. Experimental evaluation

Evaluation platform. All experiments use Gecode, a C++-based constraint programming library (Gecode Team 2006). Gecode is one of the fastest constraint programming systems currently available, benchmarks comparing Gecode to other systems are available from Gecode's webpage. The version used here corresponds to Gecode 2.1.1. Gecode has been compiled with the Microsoft Visual Studio Express Edition 2008.

All examples have been run on a Laptop with a 2.2 GHz Core2 Duo CPU and 2048 MB main memory running 32bit Windows Vista. Runtimes are the average of 25 runs with a coefficient of deviation less than 5% for all benchmarks.

Example characteristics. Table 1 summarizes the characteristics of the examples used for evaluation. Time is runtime in milliseconds, memory is allocated memory in KB, and exec refers to the number of propagator executions. Note that the same runtime for

Table 1. Example characteristics

Example	variables	propagators	failures	time (ms)	memory (KB)	exec
`bibd-6-3-30`	9 281	8 535	2 303	252.08	1 741	1 112 526
`bibd-6-4-30`	4 631	4 260	1 344	156.62	969	678 027
`bibd-7-3-20`	10 760	9 925	426	139.76	3 473	343 879
`bibd-7-3-40`	21 540	19 865	866	536.64	5 915	1 240 958
`circle-6-21`	72	77	56 867	536.64	125	2 548 578
`circle-8-24`	128	145	32 345	485.44	245	2 766 188
`s-p-8-5`	128	145	19 706	212.16	131	1 558 275
`s-p-8-10`	128	145	635 591	6 786.60	132	49 574 399
`p-s-p-21-112`	5 810	5 978	150	96.72	3 652	1 548 480
`p-s-p-25-147`	8 894	9 142	1 109	412.44	4 869	1 761 354
`p-s-p-28-201`	13 226	13 548	833	741.92	7 236	7 964 519

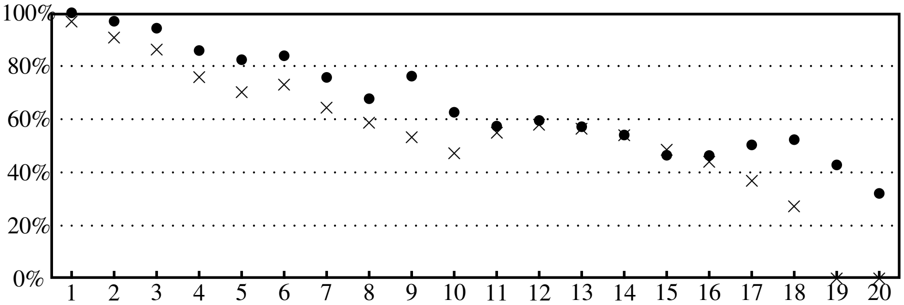

Figure 4. Average analysis accuracy: ● shows the accuracy at the n-th fixpoint and × shows the accuracy at the n-th invocation of search.

`bibd-7-3-40` and `circle-6-21` is just a coincidence and not a mistake in the evaluation.

The examples are as follows:

- `bibd`-v-k-l are instances of a balanced incomplete block design problem with parameters (v, k, l) (`prob028` in (CSPLib 2006)). The model involves Boolean-sum propagators and propagators implementing lexicographic order for symmetry breaking as described in Example 1.3.
- `circle`-n-s are circle packing problem where n circles must be packed into a $s \times s$ square. The example uses the propagators described in Example 1.2 for constraining circles to not overlap.
- `s-p`-n-s are variants of the square packing problem for n squares to be packed into a $s \times s$ square (`prob009` in (CSPLib 2006)), where multiple squares of the same size are allowed. The only constraints used are those discussed in Example 1.1.
- `p-s-p`-n-s is similar, in addition it uses many small reified constraints to improve capacity propagation by taking the size of squares packed at a particular x or y coordinate into account.

The choice of examples is motivated by the following facts. First, all examples feature propagators that our analysis could remove. Second, analysis for `bibd`-v-k-l is bound to be very expensive: many propagators need to be traversed during analysis and only few can be deleted during each run. For `circle`-n-s and `s-p`-n-s, analysis should be very efficient: all propagators can potentially be deleted. `p-s-p`-n-s provides an interesting contrast to `s-p`-n-s in that only few propagators could be deleted and also the impact of deletion will be rather small as most propagation is concerned with capacity.

Analysis accuracy. Figure 4 provides an overview of the accuracy of analysis. The analysis is run either every n-th fixpoint (shown as black bullet) or every n-th recursive invocation of search (shown as cross) where n ranges from 1 to 20 and accuracy from 0% to 100%. The measure of accuracy is based on the number of propagator executions avoided by deleting propagators during analysis. An accuracy of 100% is achieved by running the analysis immediately after *every* fixpoint computed. An accuracy of 50% means that only half of all propagator executions are avoided. The percentage numbers displayed in Figure 4 are the geometric means of the accuracy of all examples.

An important observation is that the accuracy deteriorates only relatively slowly with running the analysis more infrequently. Particularly interesting is that for both strategies, after fixpoint and before search, the analysis accuracy stays around 50% up to running the analysis only every 10 to 15 operations. This is significant as it gives ample opportunity to balance the overhead of analysis with its accuracy.

Best performance. Before studying in more detail the tradeoff between accuracy and cost of the analysis, let us first establish that the analysis can actually deliver speedup in most cases and independent of whether the analysis is run after fixpoint or before search.

Table 2 (a) gives the best n for running the analysis after every n-th fixpoint, whereas Table 2 (b) gives the best n for running the analysis before every n-th recursive call to search. Here, best refers to shortest runtime. Accuracy is shown as described earlier, whereas all other measures are given relatively to not running the analysis at all as in Table 1. A negative percentage means that the measure is decreased by that percentage (hence, better) and a positive percentage means that the measure is increased by that percentage (hence, worse).

The reason why many examples do not show an improvement in memory consumption is due to the fact that memory refers

Table 2. Best performance for analysis

Example	n	time	memory	exec	accuracy
bibd-6-3-30	17	−23.0%	−7.4%	−35.3%	96.3%
bibd-6-4-30	11	−16.7%	±0.0%	−25.8%	87.1%
bibd-7-3-20	19	−8.5%	−5.5%	−17.8%	84.1%
bibd-7-3-40	19	−7.0%	−6.5%	−19.1%	91.8%
circle-6-21	3	−24.7%	±0.0%	−20.4%	87.3%
circle-8-24	3	−7.2%	±0.0%	−7.1%	72.2%
s-p-8-5	5	−10.2%	±0.0%	−25.4%	79.3%
s-p-8-10	5	−13.6%	±0.0%	−23.2%	85.8%
p-s-p-21-112	13	+1.3%	±0.0%	−0.1%	29.3%
p-s-p-25-147	13	+2.1%	−1.3%	−0.5%	70.0%
p-s-p-28-201	13	+2.2%	−1.8%	±0.0%	40.0%

(a) Analysis after fixpoint

Example	n	time	memory	exec	accuracy
bibd-6-3-30	17	−23.1%	−7.4%	−35.1%	95.8%
bibd-6-4-30	11	−16.1%	±0.0%	−25.6%	86.4%
bibd-7-3-20	14	−6.8%	−5.5%	−19.0%	89.5%
bibd-7-3-40	20	−5.4%	−6.5%	−18.4%	88.6%
circle-6-21	1	−27.6%	±0.0%	−22.0%	94.3%
circle-8-24	1	−7.7%	±0.0%	−8.2%	83.5%
s-p-8-5	3	−15.7%	±0.0%	−28.5%	89.1%
s-p-8-10	3	−15.5%	±0.0%	−25.2%	93.3%
p-s-p-21-112	11	−0.4%	±0.0%	−0.1%	29.3%
p-s-p-25-147	7	−2.4%	−2.6%	−0.8%	100.0%
p-s-p-28-201	9	+1.1%	−1.8%	±0.0%	61.4%

(b) Analysis before search

Table 3. Analysis after fixpoint

Example	$n = 1$			$n = 2$			$n = 5$		
	time	mem	acc	time	mem	acc	time	mem	acc
bibd-6-3-30	+32.9%	− **7.4**%	**100.0**%	+3.7%	−**7.4**%	99.6%	−15.8%	−**7.4**%	98.0%
bibd-6-4-30	+41.4%	±**0.0**%	**100.0**%	+12.7%	±**0.0**%	98.7%	−**9.4**%	±**0.0**%	94.8%
bibd-7-3-20	+63.9%	−**12.9**%	**100.0**%	+30.1%	−11.1%	97.6%	+5.1%	−9.2%	92.1%
bibd-7-3-40	+73.9%	−**13.0**%	**100.0**%	+31.7%	−10.8%	98.6%	+5.6%	−9.7%	95.4%
circle-6-21	−21.8%	±**0.0**%	**100.0**%	−**23.3**%	±**0.0**%	94.0%	−21.5%	±**0.0**%	72.4%
circle-8-24	−0.9%	±**0.0**%	**100.0**%	−5.7%	±**0.0**%	84.5%	−**6.8**%	±**0.0**%	57.6%
s-p-8-5	+5.9%	±**0.0**%	**100.0**%	−6.3%	±**0.0**%	96.7%	−10.2%	±**0.0**%	79.3%
s-p-8-10	+2.9%	±**0.0**%	**100.0**%	−8.7%	±**0.0**%	96.8%	−13.6%	±**0.0**%	85.8%
p-s-p-21-112	+31.3%	−**3.5**%	**100.0**%	+15.9%	−**3.5**%	**100.0**%	+7.4%	±0.0%	60.7%
p-s-p-25-147	+53.4%	−**3.9**%	**100.0**%	+23.8%	−**3.9**%	99.6%	+9.4%	−2.6%	88.4%
p-s-p-28-201	+45.8%	−**1.8**%	**100.0**%	+22.5%	−**1.8**%	**100.0**%	+12.8%	−**1.8**%	95.0%

Example	$n = 10$			$n = 15$			$n = 20$		
	time	mem	acc	time	mem	acc	time	mem	acc
bibd-6-3-30	−18.7%	−**7.4**%	97.4%	−**20.5**%	−**7.4**%	95.0%	−16.8%	−3.7%	86.3%
bibd-6-4-30	−5.0%	±**0.0**%	70.2%	−9.0%	±**0.0**%	75.7%	−5.0%	±**0.0**%	38.7%
bibd-7-3-20	−3.1%	−5.5%	87.1%	−1.8%	−9.2%	85.0%	−**5.6**%	−5.5%	78.4%
bibd-7-3-40	+4.9%	−6.5%	92.4%	−3.9%	−8.7%	94.1%	−**4.5**%	−6.5%	88.6%
circle-6-21	−7.8%	±**0.0**%	37.1%	−7.9%	±**0.0**%	36.4%	+0.5%	±**0.0**%	0.8%
circle-8-24	−1.9%	±**0.0**%	28.3%	±0.0%	±**0.0**%	6.7%	−2.6%	±**0.0**%	27.7%
s-p-8-5	−5.0%	±**0.0**%	48.3%	−3.8%	±**0.0**%	50.8%	+1.5%	±**0.0**%	8.8%
s-p-8-10	−9.5%	±**0.0**%	64.5%	−7.6%	±**0.0**%	45.1%	−7.4%	±**0.0**%	40.0%
p-s-p-21-112	+5.9%	±0.0%	57.1%	+**1.4**%	±0.0%	31.9%	+1.9%	±0.0%	41.1%
p-s-p-25-147	+5.6%	−1.3%	59.5%	+4.4%	−1.3%	43.4%	+**3.0**%	−1.3%	56.1%
p-s-p-28-201	+5.6%	−**1.8**%	94.4%	+8.6%	−0.9%	49.2%	+**5.5**%	−0.9%	86.8%

to allocated rather than used memory. Gecode allocates relatively large blocks of memory that are then used: hence less memory might be in use even though the same amount of memory gets allocated.

In the best case, the number of propagator executions is reduced by one third. In these cases also the runtime is reduced by up to 25%.

The examples where it is most difficult to obtain a speedup are the p-s-p-n-s examples. One has to keep in mind that only a fraction of the propagators can be deleted by the analysis. With that in mind, the analysis is successful as it does not slowdown execution while it still saves a little memory. It is very important to put this into perspective: one should always run the analysis now and then to catch modeling errors as discussed in Section 4.

When small propagators due to decompositions are frequent the analysis shows its true potential: the runtime overhead of the analysis is easily outweighed by its benefits and regardless of how often the analysis runs, it will always save memory.

It is interesting to note that running the analysis before search rather than after fixpoint appears to be the better decision, even though the benefit might only be specific to Gecode as the system used for the evaluation. Running analysis before search slightly reduces the accuracy but the effect of every single analysis run is apparently reused several times.

How often to run the analysis. The key question is whether a user can determine how often she should run the analysis a priori. Finding an appropriate frequency by inspecting runtimes of several tries might be infeasible.

In the following we will restrict our attention to running the analysis directly after fixpoint, as the insight to be gained from running before search is similar. Table 3 shows relative runtime, memory usage, and accuracy for several values of n.

It is obvious that trying to run the analysis very often is infeasible, interesting values for n start with $n = 5$. More importantly, for all examples values between 10 and 15 offer a feasible compromise between accuracy and reduction in memory and runtime. Hence it is plausible that a user can use $n = 10$ as a starting point for the analysis. With vastly different problem sizes the user might decrease the frequency depending on problem size.

7. Related Work

The detection of redundant propagators, and their elimination from the propagation system in forward execution has been a part of propagation systems since almost the beginning. But this redundancy detection does not know which variables are of interest to the answer and hence cannot dynamically eliminate variables.

There has been earlier work on variable elimination, most notably in the context of constraint logic programming over the real numbers (Jaffar et al. 1992). Here, particularly with recursive definitions, many intermediate variables are introduced, and analysis can sometimes determine that they can be eliminated during execution. The difference here is that these solvers can use Gauss-Jordan variable elimination (Macdonald et al. 1993), and Fourier elimination (Fordan and Yap 1998) to eliminate variables that only appear in linear real constraints. While elimination that could be detected at compile time and removed using Gauss-Jordan elimination (Macdonald et al. 1993) was clearly beneficial, for the more complex Fourier elimination (Fordan and Yap 1998) the cost of the elimination was only paid back in certain circumstances. Elimination of variables in finite domain constraints is much more restricted.

There is a relationship of this work with so-called “don’t care” propagation in non-clausal SAT solvers (Thiffault et al. 2004). Here Boolean formulae are represented as a DAG with leaves made up of the Boolean variables. Don’t care propagation of the node representing e.g. a disjunctive constraint $\mathrm{or}(x_1, \ldots, x_n)$ realizes that if the node is true and x_i is true for some $1 \leq i \leq n$ then this node does not care about the remaining nodes $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$. If no parent cares about a node x_j then its value becomes “don’t-care” and this may propagate to its child nodes. The result is akin to dynamic variable elimination on the propagator graph representing the DAG. The solver used “don’t-care” values to avoid propagation rather than eliminating the variables altogether.

Finally, the work of Brand and Yap (Brand and Yap 2006) on finer control of propagation is related. It effectively extends the “don’t care” propagation of (Thiffault et al. 2004) to formulae involving non-Boolean leaf constraints, and uses this to prevent “unrolling” of complex constraint definitions. So it ties variable elimination and constraint definition together. In this way it can define 2-literal watching, and domain consistent lex propagation. In contrast our approach does not consider preventing unrolling/decomposition of global constraints but the existential redundancy approach is not restricted to Boolean variables.

8. Conclusion

Dynamic variable elimination is a useful optimization for finite domain constraint systems, since modeling requires the introduction of many intermediate variables, which may become irrelevant in later solving. We give a linear time analysis of the propagator graph to detect occurrences of dynamic variables to eliminate. We show that the analysis can improve space and time performance for finite domain problems. There is an ancillary benefit, the analysis can detect modelling errors that leave part of the propagation graph separated from the search variables.

An avenue for future research is to automatically find out when it is profitable to run the analysis. A simple scheme could start from the idea to dynamically adapt the frequency as follows: when a run of the analysis was useful (that is, many propagators were deleted), the frequency is increased. Otherwise, the frequency is decreased.

In a production level implementation we believe the overhead of the analysis could be reduced substantially from this prototype. Indeed if tied to the copying of search state, the analysis could be folded into the copying stage and should execute with little overhead compared to the copying itself.

Acknowledgments

Part of the work has been carried out while the first author has been a visiting researcher at the NICTA Victoria research laboratory, Melbourne, Australia. The authors thank Guido Tack for helpful comments.

References

Sebastian Brand and Roland H. C. Yap. Towards ”Propagation = Logic + Control”. In Sandro Etalle and Miroslaw Truszczynski, editors, *Logic Programming, 22nd International Conference*, pages 102–116, 2006.

Chiu Wo Choi, Warwick Harvey, Jimmy Ho-Man Lee, and Peter J. Stuckey. Finite domain bounds consistency revisited. In *AI 2006: Advances in Artificial Intelligence*, volume 4304 of *Lecture Notes in Computer Science*, pages 49–58. Springer-Verlag, Berlin, Germany, 2006.

CSPLib. CSPLib: a problem library for constraints, 2006. Available from http://www.csplib.org.

Andreas Fordan and Roland H. C. Yap. Early projection in CLP($\mathcal{R}$). In Michael J. Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming - CP98, 4th International Conference*, pages 177–191, 1998.

Gecode Team. Gecode: Generic constraint development environment, 2006. Available from http://www.gecode.org.

Ian P. Gent, Chris Jefferson, and Ian Miguel. Watched literals for constraint propagation in Minion. In Frédéric Benhamou, editor, *Twelfth International Conference on Principles and Practice of Constraint Programming*, volume 4204 of *LNCS*, pages 182–197, Nantes, France, September 2006. Springer.

Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H.C. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

Mikael Z. Lagerkvist and Christian Schulte. Advisors for incremental propagation. In Christian Bessière, editor, *Thirteenth International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 409–422, Providence, RI, USA, September 2007. Springer-Verlag.

A.D. Macdonald, P.J. Stuckey, and R.H.C. Yap. Redundancy of variables in CLP($\mathcal{R}$). In *Logic Programming: Proceedings of the 1993 International Symposium*, pages 75–93, Vancouver, Canada, October 1993. MIT Press.

Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

Jean-François Puget and Michel Leconte. Beyond the glass box: Constraints as objects. In John Lloyd, editor, *Proceedings of the International Symposium on Logic Programming*, pages 513–527, Portland, OR, USA, December 1995. The MIT Press.

Christian Schulte. *Programming Constraint Services*, volume 2302 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2002.

Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *Transactions on Programming Languages and Systems*, 2008. To appear.

Christian Schulte and Peter J. Stuckey. Speeding up constraint propagation. In M. Wallace, editor, *Proceedings of the International Conference on Principle and Practice of Constraint Programming*, volume 3258 of *LNCS*, pages 619–633. Springer-Verlag, 2004.

Christian Thiffault, Fahiem Bacchus, and Toby Walsh. Solving non-clausal formulas with DPLL search. In Mark Wallace, editor, *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference*, pages 663–678, 2004.