Speeding Up Constraint Propagation

Christian Schulte[1] and Peter J. Stuckey[2]

[1] IMIT, KTH - Royal Institute of Technology, Sweden, schulte@imit.kth.se
[2] Dept. of Comp. Sci. & Soft. Eng., Univ. of Melbourne, Australia, pjs@cs.mu.oz.au

Abstract. This paper presents a model and implementation techniques for speeding up constraint propagation. Two fundamental approaches to improving constraint propagation are explored: keeping track of which propagators are at fixpoint, and choosing which propagator to apply next. We show how idempotence reasoning and events help track fixpoints more accurately. We improve these methods by using them dynamically (taking into account current domains to improve accuracy). We define priority-based approaches to choosing a next propagator and show that dynamic priorities can improve propagation. We illustrate that the use of multiple propagators for the same constraint can be advantageous with priorities, and introduce staged propagators which combine the effects of multiple propagators with priorities for greater efficiency.

1 Introduction

At the core of a finite domain constraint programming system is a constraint propagation engine that repeatedly executes propagators for the constraints of a problem. Propagators discover and remove values from the domains of variables which can no longer take part in a solution of the constraints.

There are two important decisions the engine must make: which propagators should execute, and in which order should they execute. In this paper we investigate how to make a propagation engine as efficient as possible.

In order to make constraint propagation efficient, it is clear that the engine needs to take the following issues into account: avoid unnecessary propagator execution, restrict propagation to relevant variables, and choose the cheapest possible method for propagation. In this paper we show how propagation can be speeded up if the engine takes these issues into account.

The contributions of the paper are as follows. We give a formal definition of propagation systems including idempotent and event-based optimizations used in current propagation systems. We extend event-based propagation systems to use dynamically changing event sets. We introduce multiple propagators and staged propagators for use with propagation queues with priority. We give experimental results that clarify the impact of idempotent and event-based optimizations and show that dynamic event sets and staged propagators can be advantageous.

Plan of the Paper. The next section introduces propagation-based constraint solving, followed by a model for constraint propagation systems in Sect. 3. Section 4 presents how to optimize propagation by taking idempotence and events

into account. Which propagator should be executed next and how propagation can be organized is discussed in Sect. 5. The following section evaluates the different variants of constraint propagation discussed. Section 7 concludes.

2 Propagation-based Constraint Solving

This section defines terminology and the basic components of a constraint propagation engine. In this paper we restrict ourselves to integer constraint solving.

Domains. 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 $x \in \mathcal{V}$. Let $D_\perp(x) = \emptyset$ for all $x \in \mathcal{V}$. 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 shall use range notation $[l \mathinner{..} u]$ to define the set of integers $\{d \mid l \leq d \leq u\}$.

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. In an abuse of notation, 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 $x_i \in \mathit{vars}(\theta)$.

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

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

Propagators. We will *implement* a constraint c by a set of propagators $\mathit{prop}(c)$ which 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

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

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, if for domains D where all variables $\mathit{vars}(c)$ are fixed the following holds: $f(D) = D$ for all $f \in F$, iff the unique valuation $\theta \in D$ where $\mathit{vars}(\theta) = \mathit{vars}(c)$ is a solution of c ($\theta \in c$). We assume that $\mathit{prop}(c)$ is a set of propagators that are correct and checking for c.

The *output* variables $\mathit{output}(f) \subseteq \mathcal{V}$ of a propagator f are the variables changed by the propagator: $x \in \mathit{output}(f)$ if there exists a domain D such

that $f(D)(x) \neq D(x)$. The *input* variables $input(f) \subseteq \mathcal{V}$ of a propagator f is the smallest subset $V \subseteq \mathcal{V}$ such that for each domain D: $D =_V D'$ implies that $D' \sqcap f(D) =_{output(f)} f(D') \sqcap D$. Only the input variables are useful in computing the application of the propagator to the domain.

Example 1. For the constraint $c \equiv x_1 \leq x_2 + 1$ the function f_A defined by $f_A(D)(x_1) = \{d \in D(x_1) \mid d \leq \sup_D x_2 + 1\}$ and $f_A(D)(v) = D(v), v \neq x_1$ is a correct propagator for c. Its output variables are $\{x_1\}$ and its input variables are $\{x_2\}$. 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_B defined as $f_B(D)(x_2) = \{d \in D(x_2) \mid d \geq \inf_D x_1 - 1\}$ and $f_B(D)(v) = D(v), v \neq x_2$ is another correct propagator for c. Its output variables are $\{x_2\}$ and input variables $\{x_1\}$.

The set $\{f_A, f_B\}$ is checking for c. The domain $D(x_1) = D(x_2) = \{2\}$ corresponding to a solution of c is a fixpoint of both propagators. The non-solution domain $D(x_1) = \{2\}$, $D(x_2) = \{0\}$ is not a fixpoint (of either propagator).

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

$$solv(F, D) = \text{gfp}(\lambda d.iter(F, d))(D) \qquad iter(F, D) = \bigsqcap_{f \in F} f(D)$$

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

3 Constraint Propagation Systems

A constraint propagation system evaluates the function $solv(F, D)$ during backtracking search. We assume an execution model for solving a constraint problem with a set of constraints C and an initial domain D_0 as follows. We execute the procedure $\mathsf{search}(\emptyset, F, D_0)$ for an initial set of propagators $F = \cup_{c \in C} prop(c)$. This procedure is used to make precise the optimizations presented in the remainder of the paper.

```
search(F_o, F_n, D)
    D := isolv(F_o, F_n, D)                                  % propagation
    if (D is a false domain) return false
    if (∃x ∈ V.|D(v)| > 1)
        choose {c_1, ..., c_m} where C ∧ D ⊨ c_1 ∨ ··· ∨ c_m   % search strategy
        for i ∈ [1 .. m]
            if (search(F_o ∪ F_n, prop(c_i), D)) return true
    return true
```

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

The somewhat unusual definition of search is quite general. The default search strategy for many problems is to choose a variable x such that $|D(x)| > 1$ and explore $x = \inf_D x$ or $x \geq \inf_D x + 1$. This is commonly thought of as changing the domain D for x to either $\{\inf_D x\}$ or $\{d \in D(x) \mid d > \inf_D x\}$. This framework allows more general strategies, for example $x_1 \leq x_2$ or $x_1 > x_2$.

The basic incremental propagation solver algorithm is as follows:

```
isolv(F_o, F_n, D)
    F := F_o ∪ F_n; Q := F_n
    while (Q ≠ ∅)
        f := choose(Q) % select next propagator to apply
        Q := Q − {f}; D' := f(D)
        Q := Q ∪ new(f, F, D, D') % add propagators f' ∈ F
        D := D'                   % not necessarily at fixpoint at D'
    return D
```

The algorithm uses a queue of propagators to apply Q. Initially, Q contains the new propagators. Each time the while loop is executed, a propagator f is deleted from the queue, f is applied, and then all propagators that may no longer be at a fixpoint at the new domain D' are added to the queue. An invariant of the algorithm is that at the while statement $f(D) = D$ for all $f \in F - Q$.

isolv leaves two components undefined: $\mathsf{choose}(Q)$ chooses the propagator $f \in Q$ to be applied next; $\mathsf{new}(f, F, D, D')$ determines the set of propagators $f' \in F$ which are not guaranteed to be at their fixpoint at the domain D'. The remainder of the paper investigates how to best implement these two components.

4 Fixpoint Reasoning

The core aim of the constraint propagation solver $solv(F, D)$ is to find a domain that is a mutual fixpoint of all $f \in F$. The incremental solver $\mathsf{isolv}(F_o, F_n, D)$ already takes into account that initially D is a fixpoint of propagators $f \in F_o$. The role of new is (generally) to return *as few* propagators $f \in F$ as possible.

A basic definition of new is as follows

$$\mathsf{new}_{input}(f, F, D, D') = \{f' \in F \mid input(f') \cap \{x \in \mathcal{V} \mid D(x) \neq D'(x)\} \neq \emptyset\}$$

Here all propagators f' are added whose input variables domain have changed. By the definition of input variables, if none of them have changed for f', then $f'(D') = D'$ since $f'(D) = D$ if $f' \in F - Q$.

isolv with this definition of new (assuming $F_o = \emptyset$) is more or less equivalent to the propagation algorithm of Apt ([1] page 267).

4.1 Idempotence

A propagator f is *idempotent* if $f(D) = f(f(D))$ for all domains D. That is, applying f to any domain D yields a fixpoint of f.

Example 2. The propagator f_C defined by $f_C(D)(x_1) = \{d \in D(x_1) \mid \frac{3}{2}d \in D(x_2)\}$, $f_C(D)(x_2) = \{d \in D(x_2) \mid \frac{2}{3}d \in D(x_1)\}$, and $f_C(D)(x) = D(x), x \notin \{x_1, x_2\}$ is the domain propagator for the constraint $3x_1 = 2x_2$. The propagator f_C is idempotent.

Example 3. While many propagators are idempotent, some widely used ones are *not* idempotent. Consider the constraint $3x_1 = 2x_2$ and the propagator f_G:

$$\begin{aligned} f_G(D)(x_1) &= D(x_1) \cap [\lceil (2 \inf_D x_2)/3 \rceil \mathinner{..} \lfloor (2 \sup_D x_2)/3 \rfloor] \\ f_G(D)(x_2) &= D(x_2) \cap [\lceil (3 \inf_D x_1)/2 \rceil \mathinner{..} \lfloor (3 \sup_D x_1)/2 \rfloor] \\ f_G(D)(x) &= D(x) \quad x \notin \{x_1, x_2\} \end{aligned}$$

Then $prop(3x_1 = 2x_2) = \{f_G\}$ in almost all constraint programming systems. f_G is the the bounds propagator (considering solutions over the real numbers) for for $3x_1 = 2x_2$. Now f_G is not idempotent. Consider $D(x_1) = [0 \mathinner{..} 3]$ and $D(x_2) = [0 \mathinner{..} 5]$. Then $D' = f_G(D)$ is defined by $D'(x_1) = [0 \mathinner{..} 3] \cap [0 \mathinner{..} \lfloor 10/3 \rfloor] = [0 \mathinner{..} 3]$ and $D'(x_2) = [0 \mathinner{..} 5] \cap [0 \mathinner{..} \lfloor 9/2 \rfloor] = [0 \mathinner{..} 4]$. Now $D'' = f_G(D')$ is defined by $D''(x_1) = [0 \mathinner{..} 3] \cap [0 \mathinner{..} \lfloor 8/3 \rfloor] = [0 \mathinner{..} 2]$ and $D''(x_2) = [0 \mathinner{..} 4] \cap [0 \mathinner{..} \lfloor 9/2 \rfloor] = [0 \mathinner{..} 4]$. Hence $f_G(f_G(D)) = D'' \neq D' = f_G(D)$.

We can always create an idempotent propagator f' from a propagator f by defining $f'(D) = solv(\{f\}, D)$. Indeed, in some implementations (for example [8]) $prop(3x_1 = 2x_2)$ is defined as the fixpoint of applying f_G.

Static Idempotence. Assume that $idem(f) = \{f\}$ if f is an idempotent propagator and $idem(f) = \emptyset$ otherwise. The definition of new is improved by taking idempotence into account

$$\mathsf{new}_{sidem}(f, F, D, D') = \mathsf{new}_{input}(f, F, D, D') - idem(f)$$

An idempotent propagator is never put into the queue after application.

Note that this is an important optimization. Otherwise each propagator f that changes the domain is likely to be executed again to check it is at fixpoint. Almost all constraint propagation solvers take into account static idempotence (for example ILOG Solver [9], Choco [11], and SICStus [10]). Some systems even only allow idempotent propagators (for example Mozart [13]).

Dynamic Idempotence. Even if a propagator is not idempotent we can often determine that $f(D)$ is a fixpoint of f for a specific domain D.

We can make use of dynamic idempotence by extending a propagator f to return a new domain D' and an indication whether D' is a fixpoint. For simplicity we assume a function $idem(f, D)$ that returns $\{f\}$ if it can show that $f(D)$ is a fixpoint for f and $\emptyset$ otherwise (of course without calculating $f(f(D))$, otherwise we gain nothing). In practice this will be included in the code for f.

$$\mathsf{new}_{didem}(f, F, D, D') = \mathsf{new}_{input}(f, F, D, D') - idem(f, D)$$

Note that the dynamic case extends the static case since for idempotent f it holds that $idem(f, D) = \{f\}$ for all domains D.

Example 4. Consider applying f_G from Example 3 to the domain D'' from the same example. Now $D''' = f_G(D'')$ is defined by $D'''(x_1) = [0 \,..\, 2] \cap [0 \,..\, \lfloor 8/3 \rfloor] = [0 \,..\, 2]$ and $D'''(x_2) = [0 \,..\, 4] \cap [0 \,..\, \lfloor 6/2 \rfloor] = [0 \,..\, 3]$. Notice that the new bound $x_2 \leq 3$ is obtained without rounding $\lfloor 6/2 \rfloor = 6/2$. In this case we are guaranteed that the propagator is at a fixpoint ([8] Theorem 8).

4.2 Events

The next improvement for avoiding propagators to be put in the queue is to consider what changes in domains of input variables can cause the propagator to no longer be at a fixpoint. An *event* is a change in the domain of a variable.

Assume that the domain D changes to the domain $D' \sqsubseteq D$. The usual events defined in a constraint propagation system are:

- *fix*(x): the variable x becomes fixed, that is $|D'(x)| = 1$ and $|D(x)| > 1$.
- *lbc*(x): the lower bound of variable x changes, that is $\inf_{D'} x > \inf_D x$.
- *ubc*(x): the upper bound of variable x changes, that is $\sup_{D'} x < \sup_D x$.
- *dmc*(x): the domain of variable x changes, that is $D'(x) \subset D(x)$.

Clearly the events overlap. Whenever a *fix*(x) event occurs then a *lbc*(x) event, a *ubc*(x) event, or both events must also occur. If any of the first three events occur then a *dmc*(x) event occurs. These events satisfy the following property.

Definition 1. *An* event ϕ *is a change in domain defined by an event condition* $\phi(D, D')$ *which states that event* ϕ *occurs when the domain changes from* D *to* $D' \sqsubseteq D$*. The event condition must satisfy the following property*

$$\phi(D, D'') = \phi(D, D') \vee \phi(D', D'')$$

where $D'' \sqsubseteq D' \sqsubseteq D$*. So an event occurs on a change from* D *to* D'' *iff it occurs in either the change from* D *to* D' *or from* D' *to* D''*.*

Given a domain D *and a stronger domain* $D' \sqsubseteq D$*, then* $events(D, D')$ *is the set of events* ϕ *where* $\phi(D, D')$*. Suppose* $D'' \sqsubseteq D' \sqsubseteq D$*, then clearly* $events(D, D'') = events(D, D') \cup events(D', D'')$*.*

Most integer propagation solvers use the events defined above, although some systems collapse *ubc*(x) and *lbc*(x) into a single event (for example, SICStus [10] and ILOG Solver [9]). Choco [11] maintains an event queue and interleaves propagator execution with events causing more propagators to be added to the queue.

Static Event Sets. Re-execution of certain propagators can be avoided since they require certain events to generate new information.

Definition 2. *A propagator* f *is* dependent on *a set of events* $es(f)$ *iff* (a) *for all domains* D *if* $f(D) \neq f(f(D))$ *then* $events(D, f(D)) \cap es(f) \neq \emptyset$*, and* (b) *for all domains* D *and* D' *where* $f(D) = D$*,* $D' \sqsubseteq D$ *and* $f(D') \neq D'$ *then* $events(D, D') \cap es(f) \neq \emptyset$*.*

The definition captures the following. If f is not at a fixpoint then one of the events in its event set occurs. If f is at a fixpoint D then any change to a domain which is not a fixpoint D' involves an occurrence of one of the events in its set. Note that for idempotent propagators the case (a) never occurs.

For convenience later we will store the event set chosen for a propagator f in an array $evset[f] = es(f)$.

Clearly if we keep track of the events since the last invocation of a propagator, we do not need to apply a propagator if it is not dependent on any of these events.

Example 5. An event set for f_A is $\{ubc(x_2)\}$. An event set for f_B is $\{lbc(x_1)\}$. An event set for f_C is $\{dmc(x_1), dmc(x_2)\}$. An event set for f_G is $\{lbc(x_1), ubc(x_1), lbc(x_2), ubc(x_2)\}$. This is easy to see from the definitions of these propagators. If they use $\inf_D x$ then $lbc(x)$ is in the event set, similarly if they use $\sup_D x$ then $ubc(x)$ is in the event set. If they use the entire domain $D(x)$ then $dmc(x)$ is in the event set.

Indexical propagation solvers [17, 5, 4] are based on such reasoning. They define propagators in the form $f(D)(x) = D(x) \cap e(D)$ where e is an indexical expression. The event set for such propagators is automatically defined by the domain access terms that occur in the expression e.

Using events we can define a much more accurate version of new that only adds propagators for which one of the events in its event set has occurred.

$$\mathsf{new}_{events}(f, F, D, D') = \{f' \in F \mid evset[f'] \cap events(D, D') \neq \emptyset\} - idem(f, D)$$

This version of new (without dynamic idempotence) roughly corresponds with what most constraint propagation systems currently implement.

Dynamic Events Sets. Events help to improve the efficiency of a propagation-based solver. Just as we can improve the use of idempotence by examining the dynamic case, we can also consider dynamically updating event sets as more information is known about the variables in the propagator.

Definition 3. *A propagator f is* dependent on *a set of events $es(f, D)$* in the context of domain D *iff for all domains $D_0 \sqsubseteq D$ if $f(D_0) \neq f(f(D_0))$ then $events(D_0, f(D_0)) \cap es(f) \neq \emptyset$, and for domains D_0 and D_1 where $D_0 \sqsubseteq D$, $f(D_0) = D_0$, $D_1 \sqsubseteq D_0$ and $f(D_1) \neq D_1$ then $events(D_0, D_1) \cap es(f) \neq \emptyset$.*

Clearly given this definition $es(f, D)$ is monotonically decreasing with D. The simplest kind of event reduction occurs by entailment.

Definition 4. *A propagator f is* entailed *for domain D, if for each domain $D' \sqsubseteq D$ we have $f(D') = D'$.*

An entailed propagator makes no future contribution. If f is entailed by D then $es(f, D) = \emptyset$ and f is never re-applied. Most current constraint propagation systems take into account entailment.

Example 6. Consider the propagator f_A and the domain D with $D(x_1) = [1 .. 3]$ and $D(x_2) = [3 .. 7]$. Then the constraint holds for all $D' \sqsubseteq D$ and $es(f, D) = \emptyset$.

Changing event sets can occur in cases other than entailment.

Example 7. Consider the propagator f_H for $x_0 = \min(x_1, x_2)$ defined by

$$\begin{aligned} f_H(D)(x_0) &= D(x_0) \cap [\min(\inf_D x_1, \inf_D x_2) .. \min(\sup_D x_1, \sup_D x_2)] \\ f_H(D)(x_i) &= D(x_i) \cap [\inf_D x_0 .. +\infty] && i \in \{1, 2\} \\ f_H(D)(x) &= D(x) && x \notin \{x_0, x_1, x_2\} \end{aligned}$$

The static event set $es(f_H)$ is $\{lbc(x_0), lbc(x_1), ubc(x_1), lbc(x_2), ubc(x_2)\}$. Note that this propagator is idempotent. But given domain D where $D(x_0) = [1 .. 3]$ and $D(x_2) = [5 .. 7]$ we know that modifying the value of x_2 will never cause propagation. A minimal definition of $es(f_H, D)$ is $\{lbc(x_0), lbc(x_1), ubc(x_1)\}$.

Another example is a propagator for the exactly constraint [18]: exactly m out of the variables $x_1, \ldots, x_n$ are equal to a value k. As soon as one of the x_i becomes different from k, all events for x_i can be ignored.

Using dynamic event sets we can refine our definition of new as follows.

$\mathsf{new}_{devents}(f, F, D, D')$
  $F' := \{f' \in F \mid evset[f'] \cap events(D, D')\} - idem(f, D)$
  $evset[f] := es(f, D')$
  return F'

Every time a propagator f is applied its event set is updated to take into account newly available information.

A related idea is the "type reduction" of [16] where propagators are improved as more knowledge on domains (here called types) becomes available. For example, the implementation of $x_0 = x_1 \times x_2$ will be replaced by a more efficient one, when all elements in $D(x_1)$ and $D(x_2)$ are non-negative.

5 Which Propagator to Execute Next

We now address how to define which propagator f in the queue Q should execute first, that is how to define the choose function.

The simplest policy is implemented as a FIFO queue of propagators. Propagators are added to the queue, if they are not already present, and choose selects the oldest propagator in the queue. The FIFO policy ensures fairness so that computation is not dominated by a single group of propagators, while possibly not discovering failure (a false domain) from other propagators quickly.

5.1 Priorities

Static Priorities. A statically prioritized queue associates with each propagator a fixed priority, we will assume an integer in the range $[1 .. k]$. In effect the queue

Q is split into k queues, $Q[1], \ldots Q[k]$ where each $Q[i]$ is a FIFO queue for the propagators with priority i. Selection always chooses the oldest propagator in the lowest numbered queue $Q[i]$ which is non-empty. Static prioritization allows to ensure that quick propagators are executed before slow propagators.

Example 8. We will assume 7 static priorities, and give names to the integer priorities as follows: UNARY=1 BINARY=2, TERNARY=3 LINEAR=4, QUADRATIC=5, CUBIC=6, and VERYSLOW=7.

For example the propagator f_I for $x_1 \leq 4$ defined by $f_I(D)(x_1) = D(x_1) \cap [-\infty \,..\, 4]$ and $f_I(D)(x) = D(x)$ when $x \neq x_1$ might be given priority UNARY, while f_C and f_G might be given priority BINARY. The domain propagator defined by Régin [15] for the `alldifferent` constraint $\wedge_{i=1}^{n} \wedge_{j=i+1}^{n} x_i \neq x_j$ (with complexity $O(n^{2.5})$) might be given priority QUADRATIC. The `alldifferent` bounds propagator defined by Puget [14] (with complexity $O(n \log n)$) might be given priority LINEAR.

Another model for priorities in constraint propagation based on composition operators is [6]. The model, however, runs all propagators of lower priority before switching propagation back to propagators of higher priority.

Most systems have some form of static priorities, typically using two priority levels (for example, SICStus [10], Mozart [13]). The two levels are often not entirely based on cost: in SICStus all indexicals have high priority and all other lower priority. While ECL^{i}PS^e [19, 7] supports 12 priority levels, its finite domain solver also uses only two priority levels where another level is used to support constraint debugging. A similar, but more powerful approach is used by Choco [11] using seven priority levels allowing both LIFO and FIFO traversal.

Prioritizing particular operations during constraint propagation is important in general. For (binary) arc consistency algorithms, ordering heuristics for the operations performed during propagation can reduce the total number of operations required [20]. For interval narrowing, prioritizing constraints can avoid slow convergence, see for example [12].

Dynamic Priorities. As evaluation proceeds, variables become fixed and propagators can be replaced by more specialized versions. If a propagator is replaced by a more specialized version, also its priority should change.

Example 9. Consider the propagator f_J for updating x_1 in the constraint $x_1 = x_2 + x_3$ defined by

$$\begin{array}{ll} f_J(D)(x_1) = D(x_1) \cap [\inf_D(x_2) + \inf_D(x_3) \,..\, \sup_D(x_2) + \sup_D(x_3)] & \\ f_J(D)(x) \;\; = D(x) & x \neq x_1 \end{array}$$

might have initial priority TERNARY. When the variable x_2 becomes fixed to d_2 say, then the implementation for x_1 can change to

$$f_J(D)(x_1) = D(x_1) \cap [d_2 + \inf_D(x_3) \,..\, d_2 + \sup_D(x_3)]$$

and the priority can change to BINARY.

Changing priorities is also relevant when a propagator with $n > 3$ variables with priority LINEAR (or worse) reduces to a binary or ternary propagator.

5.2 Combining Propagation

Multiple Propagators. Once we have a prioritized propagation queue it makes sense to have multiple propagators, say f_1 and f_2, in $prop(c)$ where f_1 is strictly stronger than the f_2 ($f_1(D) \sqsubseteq f_2(D)$). Usually we should just run f_1. But with priorities it makes sense to run the weaker (and presumably faster) propagator f_2 with a higher priority than f_1. This makes information available earlier to other propagators. When the stronger propagator f_1 is eventually run, it is able to take advantage from propagation provided by other cheaper propagators.

Example 10. Consider the propagator $f_K(D)$ for the `alldifferent` constraint.

$E := \emptyset$
for $i \in [1 \,..\, n]$
  if $(\exists d.D(x_i) = \{d\})$
    if $(d \in E)$ **return** $D_\perp$ **else** $E := E \cup \{d\}$
for $i \in [1 \,..\, n]$
  if $(|D(x_i)| > 1)$ $D(x_i) := D(x_i) - E$
return D

The propagator does a linear number of set operations in each invocation and is checking. It can be made idempotent by testing that no variable becomes fixed.

Another propagator for the same constraint is the domain propagator f_L by Régin [15]. We can use both propagators: f_K with priority LINEAR, and f_L with priority QUADRATIC. This means that we will not invoke f_L until we have reached a fixpoint of f_K and all LINEAR and higher priority propagators.

If we just use f_L then we need to invoke the more expensive f_L to obtain the same domain changes as f_K, and then fail.

Staged Propagators. Once we are willing to use multiple propagators for a single constraint it becomes worth considering how to more efficiently manage them. Instead of using two (or more) distinct propagators we can combine the several propagators into a single propagator with more effective behavior.

We assume that a propagator has an internal state variable, called its stage. When it is invoked, the stage determines what form of propagation applies.

Example 11. Consider the `alldifferent` constraint with implementations f_K and f_L discussed in Example 10. We combine them into a staged propagator:

- On a *fix*(x) event, the propagator is moved to stage A, and placed in the queue with priority LINEAR.
- On a *dmc*(x) event, unless the propagator is in stage A already, the propagator is put in stage B, and placed in the queue with priority QUADRATIC.
- Execution in stage A uses f_K, the propagator is put in stage B, and placed in the queue with priority QUADRATIC, unless it is entailed.
- Execution in stage B uses f_L, afterwards the propagator is removed from all queues (stage NONE).

The behavior of the staged propagator is identical to the multiple propagators for the sample execution of Example 10. In addition to the obvious advantage of having a single staged propagator, the advantage comes from avoiding the execution of f_L when the constraint is entailed.

Example 12. Consider the unit coefficient linear equation $\Sigma_{i=1}^{n} a_i x_i = d$ constraint where $|a_i| = 1, 1 \leq i \leq n$. We have two implementations, f_M which implements bounds consistency (considering real solutions, with linear complexity) for the constraint, and f_N which implements domain consistency (with exponential complexity).

We combine them into a staged propagator as follows:

- On a $lbc(x)$ or $ubc(x)$ event, the propagator is moved to stage A, and placed in the queue with priority LINEAR.
- On a $dmc(x)$ event, unless the propagator is in stage A already, the propagator is put in stage B, and is placed in the queue with priority VERYSLOW.
- Execution in stage A uses f_M, afterwards the propagator is put in stage B, and placed in the queue with priority VERYSLOW, unless each x_i has a range domain in which case it is removed from all queues (stage NONE).
- Execution in stage B uses f_N, afterwards the propagator is removed from all queues (stage NONE).

The staged propagator is advantageous since the "fast" propagator f_M can more often determine that its result $D' = f_M(D)$ is also a fixpoint for f_N.

Staged propagators can be used similarly for the bounds version of the `alldifferent` constraint. Another area where staged propagators can be used is constraint-based scheduling, where typically different propagation methods with different strength and efficiency are available [2].

6 Evaluation

Evaluating dynamic idempotence, modifying event sets, or calculating priorities, for example, might take more time than the time saved by reduced evaluation of propagators. Hence it is important to experimentally verify the benefits of these improvements to propagation solvers.

All experiments use Gecode, a C++-based constraint programming library currently under development.[1] Gecode is a successor to the constraint programming support in Mozart [13]. Its current performance is considerably better than Mozart. All examples have been run on a Dell Laptop with a 1.5 GHz Pentium M CPU and 512 MB main memory running Windows XP. Runtimes are the average of 25 runs with a coefficient of deviation less than 2.5% for all benchmarks.

Table 1 gives the number of propagation steps and Table 2 the runtime for the different fixpoint reasoning approaches. Propagation steps and runtimes are given as absolute values for the base solver (*input*) and relative to the base solver for all other solvers.

[1] The library is available upon request from the first author.

Table 1. Propagation steps for different fixpoint reasoning.

Benchmark	*input*	*sidem*	*didem*	*events*	*devents*
	steps	relative %			
`cars`	12 018	100.0	100.0	**99.7**	**99.7**
`golomb-10-d`	7 210 956	79.5	79.5	**77.3**	**77.3**
`partition`	147 739	81.7	81.7	**81.2**	**81.2**
`photo`	480 778	99.3	96.8	**73.4**	**73.4**
`queens-100`	505 670	100.0	100.0	**3.2**	**3.2**
`alpha`	262 499	95.7	94.4	**73.0**	**73.0**
`golomb-10-b`	7 159 721	79.0	79.3	**77.1**	**77.1**
`knights-16`	170 376	94.3	129.9	**43.7**	**43.7**
`queens-100-a`	885	100.0	87.7	**58.3**	**58.3**
`m-seq-500`	116 451	100.0	99.9	100.0	**56.0**
`minsort`	55 638	81.1	81.1	84.7	**54.5**

Fixpoint Reasoning. The examples used are standard benchmarks but `minsort` which sorts 100 numbers by using 100 min constraints involving 100 variables each together with binary linear inequalities. A `-d` (resp. `-b`) at the end of the name means that domain (resp. bounds) propagation is used for all occurring `alldifferent` and linear equation constraints. `queens-100-a` uses three naive (as in Example 10) `alldifferent`-propagators, while `queens-100` uses quadratically many binary disequality propagators. For clarity, the examples have been run without using priorities and without multiple or staged propagators.

Table 2. Runtime for different fixpoint reasoning.

Benchmark	*input*	*sidem*	*didem*	*events*	*devents*
	milliseconds	relative %			
`cars`	**7.72**	100.4	100.4	100.2	101.3
`golomb-10-d`	10 107.72	99.9	**99.8**	102.4	102.4
`partition`	136.04	**90.3**	90.4	90.9	90.8
`photo`	200.69	99.0	98.4	**93.9**	94.0
`queens-100`	139.04	102.5	102.6	74.2	**72.7**
`alpha`	179.98	99.0	99.4	**90.1**	90.4
`golomb-10-b`	**6 897.52**	104.3	100.7	102.3	102.5
`knights-16`	313.53	99.5	101.4	**98.0**	98.1
`queens-100-a`	6.97	**99.7**	101.0	101.6	101.6
`m-seq-500`	634.51	101.9	99.7	101.8	**28.1**
`minsort`	142.08	97.3	97.4	98.7	**55.0**

The benchmarks show that static idempotence (*sidem*) reduces the number of propagation steps, in particular for examples using domain-consistent `alldifferent` propagators (`golomb-10-d`, `ortho-latin-5`). This can also be true for cheap propagators (`minsort`). However, the effect on runtime is less noticeable (one reason is that useless execution of a domain-consistent `alldifferent` propagator is cheap due to its incrementality).

Benchmarks that profit from events are in particular those with propagators that depend on $fix(x)$ events (such as the disequality propagators in `queens-100` and reified propagators in `photo` and `knights`).

The second group (separate box in the tables) of examples use propagators that can take advantage of dynamic idempotence (*didem*). Dynamic idempotence appears to be not beneficial, while it can both increase (`knights-16`) and decrease (`queens-100-a`) propagation steps, the runtime shows no improvement.

The third group of examples stress the importance of using dynamic event sets. For both examples, where `minsort` uses min propagators as described in Example 7 and `m-seq-500` uses exactly propagators, the propagation steps reduce by almost a factor of two with a considerable reduction in runtime.

Table 3. Propagation steps for different priority and multiple propagator approaches.

Benchmark	no priority			static priority			dynamic priority		
	none	multi stage		none	multi stage		none	multi stage	
	steps	relative %							
`alpha`	190 042	100.0	100.0	100.9	100.9	100.9	**54.2**	**54.2**	**54.2**
`cars`	11 981	100.0	100.0	102.9	102.9	102.9	**72.1**	**72.1**	**72.1**
`knights-16`	134 727	100.0	100.0	**45.8**	**45.8**	**45.8**	**45.8**	**45.8**	**45.8**
`m-seq-500`	65 227	100.2	100.2	**68.3**	**68.3**	**68.3**	94.6	94.6	94.6
`photo`	352 897	100.0	100.0	**89.1**	**89.1**	89.2	90.9	90.9	90.9
`color-1-d`	4 117 534	148.5	102.4	**98.7**	125.4	101.6	108.9	157.6	112.7
`color-2-d`	**4 705 953**	143.9	102.6	102.9	133.1	105.2	116.0	160.9	120.0
`donald-b`	**340**	120.9	105.3	100.0	109.1	105.3	100.0	109.1	105.3
`donald-d`	**34**	197.1	105.9	108.8	217.6	214.7	108.8	217.6	217.6
`golomb-10-b`	**5 520 302**	260.1	114.2	132.5	140.4	140.4	132.5	140.4	140.4
`golomb-10-d`	**5 571 319**	101.0	113.4	130.7	138.4	138.4	130.7	138.4	138.4
`ortho-latin-5-d`	547 872	140.3	108.4	143.8	**94.4**	139.4	146.9	104.0	137.4
`partition-d`	119 893	104.7	101.6	88.5	89.4	88.4	**84.5**	85.3	**84.5**
`square-5-b`	**71 609**	105.1	108.3	129.5	133.8	133.8	106.7	110.9	110.9
`square-5-d`	**56 632**	158.5	111.4	125.1	166.6	211.1	116.7	144.1	208.4

Priorities. Table 3 shows the number of propagation steps required to solve each benchmark, relative to the base solver. Table 4 shows the relative execution times. Important additional benchmarks are `color-1` and `color-2` implementing graph coloring on large graphs (50 nodes) with large cliques (for each clique a domain-consistent `alldifferent` propagator is used). The first group of benchmarks (the upper box in the tables) does not use multiple or staged propagators. The second group uses multiple and staged propagators (bounds-consistent `alldifferent` for the `-b` variants, domain-consistent `alldifferent` and linear equalities for the `-d` variants).

The addition of static or dynamic priorities can substantially decrease the number of propagations required. It can, more rarely, also increase the number of propagations required. Once we examine timing we see that dynamic priori-

Table 4. Runtime for different priority and multiple propagator approaches.

Benchmark	No priority			Static priority			Dynamic priority		
	none	multi	stage	none	multi	stage	none	multi	stage
	millisecs	relative %							
alpha	165.04	99.7	100.2	95.9	95.8	96.5	71.8	71.7	**71.6**
cars	7.66	99.9	100.0	104.5	104.8	104.7	86.6	86.6	**86.1**
knights-16	306.48	99.9	100.0	**97.3**	97.4	97.4	97.4	97.4	97.5
m-seq-500	175.66	99.9	**99.0**	113.2	114.0	114.1	108.0	107.3	108.8
photo	188.35	99.5	100.4	**96.8**	97.7	97.7	98.3	98.9	98.9
color-1-d	10 137.76	116.6	76.9	95.9	103.0	**76.6**	104.6	123.3	83.2
color-2-d	12 036.52	108.3	80.0	99.4	99.9	**79.1**	107.5	117.4	89.1
donald-b	0.87	101.5	**85.5**	101.2	92.2	87.8	101.7	91.6	87.8
donald-d	46.80	94.0	96.3	93.6	95.5	**90.8**	96.5	91.2	96.3
golomb-10-b	7 022.12	256.2	82.0	53.1	**50.5**	50.9	52.9	50.8	51.1
golomb-10-d	10 281.96	101.9	84.7	51.8	50.5	**50.1**	51.8	50.9	50.2
ortho-latin-5-d	2 075.58	104.1	92.2	125.5	63.8	**58.1**	124.8	70.1	65.3
partition-d	123.46	103.7	96.9	80.6	79.2	78.2	78.5	77.4	**76.1**
square-5-b	143.64	102.5	82.6	82.0	78.4	77.2	78.6	75.2	**73.5**
square-5-d	2 830.28	83.9	99.6	120.5	54.0	**46.4**	112.2	54.0	51.2

ties are slightly advantageous over static priorities in most cases since they run cheaper propagators.

The addition of multiple or staged propagators unsurprisingly increases the number of propagations required, since we are increasing the number of propagators in the model, occasionally quite badly (for example, `square-5-d`). For large examples, both approaches offer considerable faster runtimes, where staged propagators are unsurprisingly always better than using multiple propagators. It is important to note that staged propagation is always best when using priorities.

7 Conclusion and Future Work

We have given a formal definition of propagation systems including idempotence, events, and priorities used in current propagation systems and have evaluated their impact. We have introduced dynamically changing event sets which are shown to improve efficiency considerably. The paper has introduced multiple and staged propagators which are shown to be an important optimization in particular for improving the efficiency of costly global constraints.

While the improvements to an engine of a propagation based constraint solver have been discussed for integer constraints, the techniques readily carry over to arbitrary constraint domains such as finite sets.

A rather obvious way to further speed up constraint propagation is to consider not only cost but also estimated impact for a propagator. However, while computing cost is straightforward it is currently not clear to us how to accurately predict propagation impact.

References

1. K. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.
2. P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-based Scheduling*. Kluwer Academic Publishers, 2001.
3. N. Beldiceanu, W. Harvey, M. Henz, F. Laburthe, E. Monfroy, T. Müller, L. Perron, and C. Schulte. TRICS 2000. Technical Report TRA9/00, School of Computing, National University of Singapore, Sept. 2000.
4. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *PLILP'97*, volume 1292 of *LNCS*, pages 191–206. Springer-Verlag, 1997.
5. P. Codognet and D. Diaz. Compiling constraints in `clp(FD)`. *The Journal of Logic Programming*, 27(3):185–226, June 1996.
6. L. Granvilliers and E. Monfroy. Implementing constraint propagation by composition of reductions. In *ICLP'03*, volume 2916 of *LNCS*, pages 300–314. Springer-Verlag, 2003.
7. W. Harvey. Personal communication, Apr. 2004.
8. W. Harvey and P. J. Stuckey. Constraint representation for propagation. In *CP'98*, volume 1520 of *LNCS*, pages 235–249. Springer-Verlag, 1998.
9. ILOG S.A. *ILOG Solver 5.0: Reference Manual*. Gentilly, France, 2000.
10. Intelligent Systems Laboratory. SICStus Prolog user's manual, 3.11.1. Technical report, Swedish Institute of Computer Science, 2004.
11. F. Laburthe. CHOCO: implementing a CP kernel. In Beldiceanu et al. [3], pages 71–85.
12. O. Lhomme, A. Gotlieb, and M. Rueher. Dynamic optimization of interval narrowing algorithms. *The Journal of Logic Programming*, 37(1–3):165–183, 1998.
13. T. Müller. The Mozart constraint extensions reference, 1999. Available from `www.mozart-oz.org`.
14. J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 359–366, Madison, WI, USA, July 1998. AAAI Press/The MIT Press.
15. J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, volume 1, pages 362–367, Seattle, WA, USA, 1994. AAAI Press.
16. P. Savéant. Constraint reduction at the type level. In Beldiceanu et al. [3], pages 16–29.
17. 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.
18. P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58:113–159, 1992.
19. M. Wallace, S. Novello, and J. Schimpf. Eclipse: A platform for constraint logic programming. Technical report, IC-Parc, Imperial College, London, UK, 1997.
20. R. J. Wallace and E. C. Freuder. Ordering heuristics for arc consistency algorithms. In *Ninth Canadian Conference on Artificial Intelligence*, pages 163–169, Vancouver, Canada, 1992.