Advisors for Incremental Propagation

Mikael Z. Lagerkvist and Christian Schulte

School of Information and Communication Technology
KTH - Royal Institute of Technology, Sweden
{zayenz, cschulte}@kth.se

Abstract. While incremental propagation for global constraints is recognized to be important, little research has been devoted to how propagator-centered constraint programming systems should support incremental propagation. This paper introduces advisors as a simple and efficient, yet widely applicable method for supporting incremental propagation in a propagator-centered setting. The paper presents how advisors can be used for achieving different forms of incrementality and evaluates cost and benefit for several global constraints.

1 Introduction

Global constraints are essential in constraint programming as they are useful for modeling and crucial for efficient and powerful propagation. For many propagators implementing global constraints, incrementality is important for efficiency.

The key features to support incremental propagation are state for propagators (to store datastructures for incremental propagation) and modification information (which variables have been modified and how have their domains changed). Without state, incrementality is impossible. Without modification information, the asymptotic complexity of a propagator is at least linear in the number of variables: a propagator must scan all its variables for modification.

Propagation comes in two flavors: variable- or propagator-centered. Variable-centered propagation is controlled by the set of modified variables with some additional information (for example, variable and constraint in AC3 [17], variable and value in AC4 [18]). Propagator-centered propagation is controlled by the set of propagators still to be propagated, see for example [2].

Providing modification information to a propagator is straight-forward with variable-centered propagation, and is used in systems such as Choco [15], ILOG Solver [13], and Minion [10]. This is not true for propagator-centered propagation which is for example used in CHIP [8], SICStus [7], and Gecode [9, 23]. While propagator-centered propagation typically lacks support for modification information, it is simple and has important advantages such as fixpoint reasoning, priorities, and priority-based staging [23].

This paper presents *advisors* as a simple, efficient, yet widely applicable method for supporting incremental propagation in a propagator-centered setting. The idea for advisors is not new; similar concepts called demons are used in CHIP [8] and SICStus [6]. This paper, however, is the first attempt to define a model, to describe an implementation, and to analyze advisors.

Basic requirements and approach. For propagator-centered propagation, it is not too difficult to record for a propagator its modified variables. However, information about modified variables is often not what a propagator needs. For example, when a variable x is modified, a propagator might need to know the position of x in an array, or the node in a variable-value graph corresponding to x. That is, a propagator requires propagator-specific information.

Providing information on domain change is difficult in a propagator-centered setting: the information is specific to each propagator, in contrast to variable-centered propagation where the information is the same for all constraints. Moreover, domain change information should be computed on demand: the overall efficiency of a system should not be compromised.

Taking these issues into account, advisors are programmed for a particular propagator to support propagator-specific modification information. Like propagators, advisors are generic in that they can be used with arbitrary variable domains. Advisors are second-class citizens compared to propagators: advisors cannot propagate, they can only advise propagators to achieve incremental and more efficient propagation. The second-class citizen status is deliberate: advisors are designed to be the simplest possible extension to support incremental propagation while introducing close to no overhead.

Contributions. The contributions of the paper are as follows:

- a simple, general, and implementation-independent model for propagation with advisors in a propagator-centered setup;
- a description of essential properties to make advised propagation well-behaved;
- implementation aspects and design decisions for advisors;
- examples of how to use advisors for global constraints.

Plan of the paper. The next section reviews propagator-based propagation in a setup that is extended to advised propagation in Sect. 3. Implementation aspects and an assessment of cost and potential benefit follow in Sect. 4. Section 5 demonstrates how advisors can be used in practice. The following section concludes the paper.

2 Simple Propagation

This section introduces the basic notions for constraint propagation with propagators. The setup is slightly uncommon: a propagator is a function that takes a domain and a state as input and returns a log as a sequence of tell operations and a new state, where the log describes the domain obtained by propagation. State and log will be essential for propagation with advisors. This paper focuses entirely on propagators. Information on how a propagator faithfully implements a constraint can for example be found in [22].

Domains. A *domain* d is a complete mapping from a finite set of variables $\mathcal{V}$ into the set of subsets of a finite subset of the integers. A domain d_1 is *stronger* than a domain d_2, written $d_1 \leq d_2$, if $d_1(x) \subseteq d_2(x)$ for all $x \in \mathcal{V}$. A domain d_1 is *strictly stronger* than a domain d_2, written $d_1 < d_2$, if $d_1 \leq d_2$ and $d_1 \neq d_2$. The *disagreement set* $\operatorname{dis}(d_1, d_2)$ of domains d_1 and d_2 is $\{x \in \mathcal{V} \mid d_1(x) \neq d_2(x)\}$. Domains d_1 and d_2 are *equal* with respect to a set of variables $X \subseteq \mathcal{V}$, written $d_1 =_X d_2$, if $d_1(x) = d_2(x)$ for all $x \in X$.

Tells and logs. A *tell* $x \sim n$ describes how to update a domain, where $x \in \mathcal{V}$ and $n \in \mathbb{Z}$ and $\sim$ is one of the relation symbols $\leq, \geq, \neq$. The *domain update* $d[x \sim n]$ of a domain d by a tell $x \sim n$ is as follows: $d[x \sim n](x) = \{m \in d(x) \mid m \sim n\}$ and $d[x \sim n](y) = d(y)$ if $x \neq y$. Note that $d[t] \leq d$ for any tell t and domain d. A tell t is *pruning* for a domain d, if $d[t] < d$. The relation symbols in a tell capture domain updates typically found in systems.

Propagators describe the result of propagation by a tuple of tells, called *log*. The *domain update* $d[l]$ of a domain d by a log l successively applies the updates from l to d. The update $d[\langle\rangle]$ by the empty log is d itself. For a non-empty log $\langle t_1, \ldots, t_n\rangle$ with $n > 0$, the update $d[\langle t_1, \ldots, t_n\rangle]$ is defined as $(d[t_1])[\langle t_2, \ldots, t_n\rangle]$. Clearly, $d[l] \leq d$ for any log l and domain d.

A log $\langle t_1, \ldots, t_n\rangle$ is *pruning* for a domain d, if $n = 0$, or t_1 is pruning for d and $\langle t_2, \ldots, t_n\rangle$ is pruning for $d[t_1]$. Note that the empty log is pruning and that a pruning log can contain multiple tells for the same variable.

Propagators. A propagator can use state for incremental propagation where the exact details are left opaque. A *propagator* is a function p that takes a domain d and a state s as input and returns a pair $\langle l, s'\rangle$ of a log l and a new state s'. The domain obtained by propagation is the update $d[l]$ of d by l. It is required that l is pruning for d (capturing that a propagator only returns pruning and hence relevant tells but not necessarily a minimal log).

As a simplifying assumption, the result of propagation is independent of state: for a propagator p and a domain d for any two states s_i with $p(d, s_i) = \langle l_i, s_i'\rangle$ $(i = 1, 2)$ it holds that $d[l_1] = d[l_2]$. Hence, the *result* of propagation $p[\![d]\!]$ is defined as $d[l]$ where $p(d, s) = \langle l, s'\rangle$ for an arbitrary state s.

A propagator p is *contracting*: by construction of a log, $p[\![d]\!] \leq d$ for all domains d. A propagator p must also be *monotonic*: if $d_1 \leq d_2$ then $p[\![d_1]\!] \leq p[\![d_2]\!]$ for all domains d_1 and d_2. A domain d is a *fixpoint* of a propagator p, if $p[\![d]\!] = d$ (that is, if $p(d, s) = \langle l, s'\rangle$ for states s, s', the log l is empty).

Variable dependencies. A set of variables $X \subseteq \mathcal{V}$ is *sufficient* for a propagator p, if it satisfies the following properties. First, no output on other variables is computed, that is, $d =_{\mathcal{V}-X} p[\![d]\!]$ for all domains d. Second, no other variables are considered as input: if $d_1 =_X d_2$, then $p[\![d_1]\!] =_X p[\![d_2]\!]$ for all domains d_1, d_2.

For each propagator p a sufficient set of variables, its *dependencies*, $\operatorname{var}[p] \subseteq \mathcal{V}$ is defined. Dependencies are used in propagation as follows: if a domain d is a fixpoint of a propagator p, then any domain $d' \leq d$ with $\operatorname{var}[p] \cap \operatorname{dis}(d, d') = \emptyset$ is also a fixpoint of p. To better characterize how propagators and variables are

organized in an implementation, the set of propagators prop[x] depending on a variable x is defined as: $p \in \text{prop}[x]$ if and only if $x \in \text{var}[p]$.

$N \leftarrow P$;
while $N \neq \emptyset$ **do**
  remove p from N;
  $\langle l, s\rangle \leftarrow p(d, \text{state}[p])$;
  $d' \leftarrow d[l]$; $\text{state}[p] \leftarrow s$;
  $N \leftarrow N \cup \bigcup_{x \in \text{dis}(d,d')} \text{prop}[x]$;
  $d \leftarrow d'$;
return d;

Algorithm 1: Simple propagation

Propagation. Propagation is shown in Algorithm 1. It is assumed that all propagators are contained in the set P and that state[p] stores a properly initialized state for each propagator $p \in P$.

The set N contains propagators not known to be at fixpoint. The remove operation is left unspecified, but a realistic implementation bases the decision on priority or cost, see for example [23]. Computing the propagators to be added to N does not depend on the size of the log l. While the log can have multiple occurrences of a variable, each variable from $\text{dis}\,(d, d')$ is considered only once.

Algorithm 1 does not spell out some details. Failure is captured by computing a failed domain (a domain d with $d(x) = \emptyset$ for some $x \in \mathcal{V}$) by propagation. A real system will also pay attention to entailment or idempotency of propagators. Propagation events describing how domains change are discussed in Sect. 4.

The result computed by Algorithm 1 is well known: the weakest simultaneous fixpoint for all propagators $p \in P$ stronger than the initial domain d. For a proof of this fact in a related setup, see for example [1, page 267].

3 Advised Propagation

Advised propagation adds advisors to the model to enable a broad and interesting range of techniques for incremental propagation while keeping the model simple. Simplicity entails in particular that capabilities of propagators are not duplicated, that the overhead for advisors is low, and that the essence of Algorithm 1 is kept. Ideally, a system with advisors should execute propagators not using advisors without any performance penalty.

The design of advisors takes two aspects into account: how an advisor gives advice to propagators (output) and what information is available to an advisor (input). Advisors are functions, like propagators are functions. From the discussion in the introduction it is clear that the input of an advisor must capture which variable has been changed by propagation and how it has been changed.

Based on the input to an advisor function, the only way an advisor can give advice is to modify propagator state and to decide whether a propagator must

be propagated ("scheduled"). Modifying the state of a propagator enables the propagator to perform more efficient propagation. Deciding whether a propagator must be propagated enables the advisor to avoid useless propagation.

The model ties an advisor to a single propagator. This decision is natural: the state of a propagator should only be exposed to advisors that belong to that particular propagator. Additionally, maintaining a single propagator for an advisor simplifies implementation.

Advisors. An *advisor* a is a function that takes a domain d, a tell t, and a state s as input and returns a pair $a(d, t, s) = \langle s', Q \rangle$ where s' is a state and Q a set of propagators. An advisor a gives advice to a single propagator p, written as $\text{prop}[a] = p$ where p is referred to as a's propagator (not to be confused with the propagators $\text{prop}[x]$ depending on a variable x). The set of propagators Q returned by a must be either empty or the singleton set $\{p\}$. The intuition behind the set Q is that an advisor can suggest whether its propagator p requires propagation ($Q = \{p\}$) or not ($Q = \emptyset$). To ease presentation, $\text{adv}[p]$ refers to the set of advisors a such that $\text{prop}[a] = p$.

As for propagators, the model does not detail how advisors handle state: if $a(d, t, s_i) = \langle s_i', Q_i \rangle$ $(i = 1, 2)$ then $Q_1 = Q_2$. In contrast to propagators, advisors have no own state but access to their propagators' state (an implementation most likely will decide otherwise).

Dependent advisors. Like propagators, advisors depend on variables. An advisor a, however, depends on a single variable $\text{var}[a] \in \mathcal{V}$. This restriction is essential: whenever an advisor a is executed, it is known that $\text{var}[a]$ has been modified. Similar to propagators, the set of advisors $\text{adv}[x]$ depending on a variable x is: $a \in \text{adv}[x]$ if and only if $x = \text{var}[a]$ (not to be confused with the advisors $\text{adv}[p]$ for a propagator p).

Variables of a propagator p and variables of its advisors are closely related. One goal with advised propagation is to make informed decisions by an advisor when a propagator must be re-executed. The idea is to trade variables on which the propagator depends for advisors that depend on these variables.

The set of *advised variables* $\text{avar}[p]$ of a propagator is defined as $\{x \in \mathcal{V} \mid \text{exists } a \in \text{adv}[p] \text{ with } \text{var}[a] = x\}$. For a propagator p, the set of dependent variables and advisors $\text{var}[p] \cup \text{avar}[p]$ must be *sufficient* for p: if a domain d is not a fixpoint of p (that is, $p[\![d]\!] < d$), then for all pruning tells $x \sim n$ for d' such that $d'[x \sim n] = d$ holds, $x \in \text{var}[p]$ or $a(d, x \sim n, s) = \langle s', \{p\} \rangle$ for some advisor $a \in \text{adv}[x] \cap \text{adv}[p]$.

Propagation. Algorithm 2 performs advised propagation. The only difference to simple propagation is that the update by the log computed by a propagator executes advisors.

Advisors are executed for each tell t in the order of the log l. Each advisor can schedule its propagator by returning it in the set Q and potentially modify the state of its propagator. Note the difference between variables occurring in the log l and variables from $\text{dis}(d, d')$: if a variable x occurs multiply in l, also

```
N ← P;
while N ≠ ∅ do
    remove p from N;
    ⟨l, s⟩ ← p(d, state[p]);
    d′ ← d; state[p] ← s;
    foreach x ∼ n ∈ l do
        d′ ← d′[x ∼ n];
        foreach a ∈ adv[x] do
            ⟨s, Q⟩ ← a(d′, x ∼ n, state[prop[a]]);
            state[prop[a]] ← s; N ← N ∪ Q;
    N ← N ∪ ⋃_{x∈dis(d,d′)} prop[x];
    d ← d′;
return d;
```

Algorithm 2: Advised propagation

all advisors in adv$[x]$ are executed multiply. Variables in dis (d, d') are processed only once. The reason for processing the same variable multiply is to provide each tell $x \sim n$ as information to advisors.

Again, the propagation loop computes the weakest simultaneous fixpoint for all propagators in P. Consider the loop invariant: if $p \in P - N$, then d is a fixpoint of p. Since the set of advised variables and dependencies of a propagator is sufficient for a propagator and an advisor always provides sufficient advice, the loop invariant holds. Hence, the result of advised propagation is as before.

The algorithm makes a rather arbitrary choice of how to provide tell information to an advisor: it first updates the domain d' by $x \sim n$ and then passes the updated domain $d'[x \sim n]$ together with $x \sim n$ to the advisor. It would also be possible to pass the not-yet updated domain d' and $x \sim n$. This decision is discussed in more detail in Sect. 4.

An essential aspect of advised propagation is that it is *domain independent*: the only dependencies on the domain of the variables are the tells. All remaining aspects readily carry over to other variable domains.

The algorithm reveals the benefit of making advisors second-class citizens without propagation rights. Assume that an advisor could also perform propagation (by computing a log). Then, after propagation by an advisor, all advisors would need to be reconsidered for execution. That would leave two options. One option is to execute advisors immediately, resulting in a recursive propagation process for advisors. The other is to organize advisors that require execution into a separate datastructure. This would clearly violate our requirement of the extension to be small and to not duplicate functionality. Moreover, both approaches would have in common that it would become very difficult to provide accurate information about domain changes of modified variables.

Dynamic dependencies. One simplifying assumption in this paper is that propagator dependencies and advised variables are static: both sets must be sufficient for all possible variable domains. Some techniques require dynamically changing dependencies, such as watched literals in constraint propagation [11]. The ex-

tension for dynamic dependencies is orthogonal to advisors, for a treatment of dynamic dependency sets see [24].

4 Implementation

This section discusses how advisors can be efficiently implemented: it details the model and assesses the basic cost and the potential benefit of advisors. Advisors will be included in Gecode 2.0.0 [9].

Advisors. Advisors are implemented as objects. Apart from support for construction, deletion, and memory managment, an advisor object maintains a pointer to its propagator object. The actual code for an advisor is implemented by a runtime-polymorphic method `advise` of the advisor's propagator. The call of `advise` corresponds to the application of an advisor in the model. Both advisor and modification information are passed as arguments to `advise`. As an advisor's propagator implements `advise`, the advisor does not require support for runtime polymorphism and hence uses less memory.

Advisors are attached to variables in the same way as propagators are. Systems typically provide one entry per propagation event where dependent propagators are stored (corresponding to prop[x] for a variable x). Typically, the propagators are organized in a suspension list, whereas in Gecode they are stored in an array. To accommodate for advisors, a variable x provides an additional entry where dependent advisors adv[x] are stored. This design in particulars entails that advisors do not honor events (to be discussed below).

Logs. The log in the model describes how propagation by a propagator should modify the domain of its variables. Most systems do not implement a log but perform the update by tells immediately. This is also the approach taken in Gecode. A notable exception is SICStus Prolog, which uses a datastructure similar to logs for implementing global constraints [14].

Performing updates immediately also executes advisors immediately. This differs from the model: the model separates propagator and advisor execution. In an implementation with immediate updates, the advisors of a propagator will be run while the propagator is running itself. When designing advisors and propagators this needs to be taken into account, in particular to guarantee consistent management of the propagator's state.

Modification information. During propagation, the domain and the tell provide information to an advisor which variable has changed and how it has changed. This information, provided as a suitable data structure, is passed as an argument to the advise function of an advisor object.

As discussed in Sect. 3, there are two options: either first modify the domain and then call the advisor, or the other way round. We chose to first modify the domain as in Algorithm 2: many advisors are only interested in the domain after update and not in how the domain changed.

There is an obvious tradeoff between information accuracy and its cost. The most accurate information is $\Delta(x) = d'(x) - d'[x \sim n]\,(x)$ as the set of values removed by $x \sim n$ from d'. Accuracy can be costly: whenever a variable x is modified by a tell, $\Delta(x)$ must be computed regardless of whether advisors actually use the information.

As a compromise between accuracy and cost, our implementation uses the smallest interval $I(x) = \{\min \Delta(x), \ldots, \max \Delta(x)\}$ as approximation. Hence, for a domain d' the interval for the pruning tell $x \leq n$ is $\{n+1, \ldots, \max d'(x)\}$, for $x \geq n$ is $\{\min d'(x), \ldots, n-1\}$, and for $x \neq n$ is $\{n\}$. For other domain operations, such as the removal of arbitrary values, $\emptyset$ can be passed to signal that anything might have changed.

Propagation events. Systems typically use propagation events to characterize changes to domains by tells. For finite domain systems, common propagation events are: the domain becomes a singleton, the minimum or maximum changes, or the domain changes. Sets of dependent variables for propagators are then replaced by event sets: only when an event from a propagator's event set occurs, the propagator is considered for re-execution.

The same approach can be taken for advisors: using sets of advised events rather than sets of advised variables. In our implementation, advisors do not use propagation events for the following reasons. Events are not essential for a system where propagator execution has little overhead [23, 24]. Per event type additional memory is required for each variable. Events for advisors would increase the memory overhead even in cases no advisors are being used. The domain change information available to an advisor subsumes events, albeit not with the same level of efficiency.

Performance assessment. Advisors come at a cost. For memory, each variable x requires an additional entry for $\text{adv}[x]$ regardless of whether advisors are used or not. If an advisor for a variable x and a propagator p is used rather than using x as a dependency of p (that is, $x \in \text{var}[p]$), additional memory for an advisor is required (this depends on the additional information an advisor stores, in our implementation the minimal overhead is 8 bytes on a 32-bit machine). For runtime, each time a variable x is modified by a tell, the tell information must be constructed and the advise function of all advisors in $\text{adv}[x]$ must be called.

Table 1. Performance assessment: runtime

Example	`base`	`a-none`	`a-run`	`a-avoid`
`stress-exec-1`	45.38	+0.2%	+55.1%	+63.5%
`stress-exec-10`	114.93	+0.9%	+88.7%	+98.7%
`queens-n-400`	519.14	±0.0%	+1316.7%	+634.5%
`queens-s-400`	14.57	+0.7%	+28.6%	+12.2%

Table 1 shows the runtime for systems using advisors compared to a system without advisors (`base`, runtime given in milliseconds). The system `a-none` provides advisors without using them, `a-run` uses advisors that always schedule their propagators (fully replacing propagator dependencies by advised variables), whereas advisors for `a-avoid` decide whether the execution of a propagator can be avoided. All runtime are relative to `base`.

All examples have been run on a Laptop with a 2 GHz Pentium M CPU and 1024 MB main memory running Windows XP. Runtimes are the average of 25 runs, the coefficient of deviation is less than 5% for all benchmarks.

The example `stress-exec-1` posts two propagators for $x < y$ and $y > x$ with $d(x) = d(y) = \{0, \ldots, 1000000\}$, whereas `stress-exec-10` posts the same propagators ten times. The advisor for avoiding propagation (system `a-avoid`) checks by $\max d(x) < \max d(y)$ and $\min d(x) > \min d(y)$ whether its propagator is already at fixpoint. `queens-n-400` uses $O(n^2)$ binary disequality propagators, whereas `queens-s-400` uses 3 alldifferent propagators to solve the 400-Queens problem.

Table 2. Performance assessment: memory

Example	`base`	`a-none`	`a-run`	`a-avoid`
`queens-n-400`	24 656.0	±0.0%	+67.6%	+67.6%
`queens-s-400`	977.0	±0.0%	+5.6%	+5.6%

These analytical examples clarify that the overhead of a system with advisors without using them is negligible and does not exceed 1%. Advisors for small and inexpensive propagators as in `stress-exec-*` and `queens-n-400` are too expensive, regardless of whether propagation can be avoided. Only for sufficiently large propagators (such as in `queens-s-400`), the overhead suggests that advisors can be beneficial. Exactly the same conclusions can be drawn from the memory overhead shown in Table 2, where memory is given as peak allocated memory in KB.

Table 3. Performance assessment: break-even

Example	`base`	`a-none`	`a-avoid`
`bool-10`	0.01	+0.2%	−16.6%
`bool-100`	0.09	+7.6%	−22.3%
`bool-1000`	1.43	+33.0%	−30.2%
`bool-10000`	238.23	+20.6%	−94.7%

Table 3 gives a first impression that advisors can actually be useful. `bool-`n has a single propagator propagating that the sum of $4n+1$ Boolean variables is at least $2n$ where $2n$ variables are successively assigned to 0 and then propagated. System `a-avoid` uses $2n+1$ advisors (constant runtime) where the other systems use a single propagator (linear runtime) with $2n+1$ dependencies (using techniques similar to those from [10]). As the number of variables increases, the benefit of advisors truly outweigh their overhead.

5 Using Advisors

This section demonstrates advisors for implementing incremental propagation. Central issues are to *avoid* useless propagation, to *improve* propagation efficiency, and to *simplify* propagator construction.

Extensional constraints. We consider two algorithms for implementing n-ary extensional constraints, GAC-2001 [5] and GAC-Schema [4]. Implementing GAC-Schema with advisors is straightforward. If a variable is modified, support for the deleted values is removed from the support lists. If a value loses a support, a new support is found. If no support can be found, the value is deleted. Advisors remove supports, while the propagator deletes values. However, advisors as well as the propagator can potentially find new supports.

Table 4. Runtime and propagation steps for extensional propagation

Example	`base`		`cheap`		`expensive`	
`rand-10-20-10-0`	4 010.33	16 103	−11.4%	−24.3%	+164.0%	−57.9%
`rand-10-20-10-1`	64 103.00	290 163	−23.1%	−37.1%	+163.7%	−63.0%
`rand-10-20-10-2`	68 971.00	257 792	−16.0%	−18.3%	+239.5%	−56.6%
`rand-10-20-10-3`	7 436.80	34 046	−20.8%	−36.5%	+165.5%	−63.2%
`rand-10-20-10-4`	4 362.33	16 988	−1.6%	−29.7%	+168.6%	−65.4%
`rand-10-20-10-5`	28 009.20	84 805	−16.3%	−7.4%	+224.5%	−53.8%
`crowded-chess-5`	1.44	586	−1.1%	+0.7%	+7.4%	+0.5%
`crowded-chess-6`	468.29	2 720	−17.1%	−2.7%	+273.7%	−3.1%

Table 4 compares runtime (left in a table cell) and number of propagator executions (right in a table cell) for different extensional propagators. `base` is the GAC-2001 propagator, `cheap` is a GAC-Schema propagator where the propagator searches for new supports, and `expensive` is a GAC-Schema propagator where advisors search for new supports.

Examples `rand-10-20-10-`n are random instances from the Second International CSP Solver Competition, and are originally from [16]. `crowded-chess-`n is a structured problem where several different chess pieces are placed on an

$n \times n$ chess board. The placement of bishops and knights is modeled by two n^2-ary extensional constraints on 0/1 variables.

Table 4 clarifies that using an incremental approach to propagate extensional constraints reduces the number of propagator executions. Using advisors to remove supports also reduces runtime. Finding new supports by advisors reduces the number of propagations the most, but is also consistently slowest: many more supports are entered into the support-lists as new supports are searched for eagerly. In contrast, searching for a new support in the propagator is done on demand. There is also a problem with priority inversion, where expensive advisors are run before cheap propagators.

As for memory, GAC-Schema will naturally use more memory than GAC-2001 since it uses an additional large datastructure. For the random problems, the memory overhead is around 5 to 6 times.

Regular. The regular constraint, introduced by Pesant in [19], constrains the values of a variable sequence to be a string of a regular language. The propagator for the regular constraint is based on a DFA for a regular language. The propagator's state maintains all possible DFA transitions for the values of the variables: values are pruned if they are no longer supported by a state reachable via a chain of possible transitions. The algorithm used in our experiments deviates slightly from both variants presented in [19]: it is less incremental in that it rescans all support information for an entire variable, if one of the predecessor or successor states for a variable is not any longer reachable.

Advisors for regular store the index of the variable in the variable sequence. When an advisor is executed, it updates the supported values taking the information on removed values into account. If a predecessor or a successor state changes reachability after values have been updated, the advisor can avoid scheduling the propagator. This can potentially reduce the number of propagator invocations. Besides improving propagator execution, advisors lead to a considerably simpler architecture of the propagator: advisors are concerned with how supported values are updated, while the propagator is concerned with analyzing reachability of states and potentially telling which variables have lost support.

Table 5. Runtime and propagation steps for regular

Example	`base`		`advise`		`domain`	
`nonogram`	803.13	122 778	+11.6%	+3.1%	+11.9%	+3.1%
`placement-1`	214.35	2 616	±0.0%	−44.8%	−0.5%	−44.8%
`placement-2`	7 487.81	91 495	−4.4%	−50.4%	−4.8%	−50.4%

Table 5 compares runtime (left in a table cell) and number of propagator executions (right in a table cell) not using advisors (`base`), using advisors but ignoring domain change information (`advise`), and using advisors and domain

change information (`domain`). The memory requirements are the same for all examples. `nonogram` uses regular over 0/1 variables to solve a 25×25 nonogram puzzle, `placement-*` uses regular to place irregularly shaped tiles into a rectangle (8×8 with 10 tiles, 10×6 with 12 tiles).

The advisor-based propagators reduce the number of propagation steps by half in case there is little propagation (propagation for `nonogram` is rather strong due to its 0/1 nature). But the reduction in propagation steps does not translate directly into a reduction in runtime: executing the regular propagator in vain is cheap. With larger examples a bigger improvement in runtime can be expected, suggested by the improvement for `placement-2` compared to `placement-1`.

Alldifferent. The propagator used for domain-consistent alldifferent follows [21]. The key to making it incremental is how to compute a maximal matching in the variable-value graph: only if a matching edge (corresponding to a value) for a variable x is removed, a new matching edge must be computed for x. An observation by Quimper and Walsh [20] can be used to avoid propagation: if a variable domain changes, but the number of values left still exceeds the number of variables of the propagator, no propagation is possible.

Table 6. Runtime and propagation steps for alldifferent

Example	`base`		`avoid`		`advise`		`domain`	
`golomb-10`	1 301.80	3 359 720	+5.6%	±0.0%	+12.9%	−18.6%	+11.2%	−18.6%
`graph-color`	191.90	150 433	+1.7%	−3.4%	+3.1%	−8.1%	+4.9%	−7.3%
`queens-s-400`	3 112.13	2 397	−0.1%	−0.1%	+27.4%	−0.3%	+23.3%	−0.3%

Table 6 shows the number of propagator executions and runtimes for examples using the domain-consistent alldifferent constraint. `base` uses no advisors, for `avoid` advisors use the above observation to check whether the propagator can propagate, for `advise` advisors maintain the matching and use the observation, and for `domain` advisors maintain the matching by relying entirely on domain change information. For `domain`, the observation is not used to simplify matching maintenance by advisors. `golomb-10` finds an optimal Golomb ruler of size 10, `graph-color` colors a graph with 200 nodes based on its cliques, and `queens-s-400` is as above.

While the number of propagator invocations decreases, runtime never decreases. Using the observation alone is not beneficial as it does not outweigh the overhead of advisors. The considerable reduction in propagator executions for `advise` and `domain` is due to early detection of failure: advisors fail to find a matching without executing their propagator. The increase in runtime is not surprising: edges are matched eagerly on each advisor invocation. This is wasteful as further propagation can remove the newly computed matching edge again before the propagator runs. Hence, it is beneficial to wait until the propagator actually

runs before reconstructing a matching. Another problem with eager matching is similar to the observations for extensional constraints: prioritizing matching by advisors over cheaper propagators leads to priority inversion between cheap propagators and expensive advisors.

Advisors again lead to an appealing separation of concerns, as matching becomes an orthogonal issue. However, the examples clarify another essential aspect of incremental propagation: even if a propagator does not use advisors, it can perform incremental propagation (such as matching incrementally). And for some propagators, it can be important to defer computation until perfect information about all variables is available when the propagators is actually run. Being too eager by using advisors can be wasteful.

Summary. The above experiments and the observations in Sect. 4 can be summarized as follows. Advisors are essential to improve asymptotic complexity for some propagators (in particular for propagators with sub-linear complexity, such as Boolean or general linear equations [12]). Advisors help achieving a good factorization of concerns for implementing propagators. However, the effort spent by an advisor must comply with priorities and must not be too eager. Efficiency improvements might only be possible for propagators with many variables.

6 Conclusions

This paper has added advisors to a propagator-centered setup for supporting more efficient propagation. Advisors are simple and do not duplicate functionality from propagators (no propagation and immediate execution). In particular, advisors satisfy the key requirement to not slow down propagation when not being used. That makes advisors a viable approach also for other propagator-centered constraint programming systems.

Advisors are shown to be useful for: increasing efficiency, in particular improving asymptotic complexity, and achieving a better factorization of concerns in the implementation of propagators (relying on the fact that advisors are programmable). The paper has clarified two other issues. First, advisors must comply with priorities in a propagator-centered approach with priorities. Second, for some propagators it is more important that an incremental algorithm is used rather than running the algorithm eagerly on variable change.

Advisors, like propagators, are generic. It can be expected that for variable domains with expensive domain operations (such as sets), the domain change information provided to an advisor can be more useful than for finite domain propagators. Adapting advisors for a particular variable domain only needs to define which domain change information is passed to an advisor by a tell.

Acknowledgments. We are grateful to Guido Tack and the anonymous reviewers for helpful comments. The authors are partially funded by the Swedish Research Council (VR) under grant 621-2004-4953.

References

1. K. Apt. *Principles of Constraint Programming*. Cambridge University Press, Cambridge, United Kingdom, 2003.
2. F. Benhamou. Heterogeneous constraint solving. In *Proceedings of the Fifth International Conference on Algebraic and Logic Programming (ALP'96), LNCS 1139*, pages 62–76, Aachen, Germany, 1996. Springer-Verlag.
3. F. Benhamou, editor. *Principles and Practice of Constraint Programming*, volume 4204 of *Lecture Notes in Computer Science*, Nantes, France, Sept. 2006. Springer-Verlag.
4. C. Bessière and J.-C. Régin. Arc consistency for general constraint networks: Preliminary results. In *IJCAI (1)*, pages 398–404, 1997.
5. C. Bessière, J.-C. Régin, R. H. C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.
6. M. Carlsson. Personal communication, 2006.
7. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. H. Hartel, and H. Kuchen, editors, *PLILP*, volume 1292 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 1997.
8. M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, pages 693–702, Tokyo, Japan, Dec. 1988.
9. Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.
10. I. P. Gent, C. Jefferson, and I. Miguel. Minion: A fast scalable constraint solver. In G. Brewka, S. Coradeschi, A. Perini, and P. Traverso, editors, *ECAI*, pages 98–102. IOS Press, 2006.
11. I. P. Gent, C. Jefferson, and I. Miguel. Watched literals for constraint propagation in Minion. In Benhamou [3], pages 284–298.
12. W. Harvey and J. Schimpf. Bounds consistency techniques for long linear constraints. In N. Beldiceanu, P. Brisset, M. Carlsson, F. Laburthe, M. Henz, E. Monfroy, L. Perron, and C. Schulte, editors, *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a workshop of CP 2002*, number TRA9/02, pages 39–46, 55 Science Drive 2, Singapore 117599, Sept. 2002.
13. ILOG Inc., Mountain View, CA, USA. *ILOG Solver 6.3 reference Manual*, 2006.
14. Intelligent Systems Laboratory. SICStus Prolog user's manual, 4.0.0. Technical report, Swedish Institute of Computer Science, Box 1263, 164 29 Kista, Sweden, 2007.
15. F. Laburthe. CHOCO: implementing a CP kernel. In N. Beldiceanu, W. Harvey, M. Henz, F. Laburthe, E. Monfroy, T. Müller, L. Perron, and C. Schulte, editors, *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, number TRA9/00, pages 71–85, 55 Science Drive 2, Singapore 117599, Sept. 2000.
16. C. Lecoutre and R. Szymanek. Generalized arc consistency for positive table constraints. In Benhamou [3], pages 284–298.
17. A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
18. R. Mohr and G. Masini. Good old discrete relaxation. In Y. Kodratoff, editor, *Proceedings of the 8th European Conference on Artificial Intelligence*, pages 651–656, Munich, Germany, 1988. Pitmann Publishing.

19. G. Pesant. A regular language membership constraint for finite sequences of variables. In Wallace [25], pages 482–495.
20. C.-G. Quimper and T. Walsh. The all different and global cardinality constraints on set, multiset and tuple variables. In B. Hnich, M. Carlsson, F. Fages, and F. Rossi, editors, *CSCLP*, volume 3978 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2005.
21. 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.
22. C. Schulte and M. Carlsson. Finite domain constraint programming systems. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, chapter 14, pages 495–526. Elsevier Science Publishers, Amsterdam, The Netherlands, 2006.
23. C. Schulte and P. J. Stuckey. Speeding up constraint propagation. In Wallace [25], pages 619–633. An extended version is available as [24].
24. C. Schulte and P. J. Stuckey. Efficient constraint propagation engines, 2006. Available from `http://arxiv.org/abs/cs.AI/0611009`.
25. M. Wallace, editor. *Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*. Springer-Verlag, Sept. 2004.