Weakly Monotonic Propagators

Christian Schulte[1] and Guido Tack[2]

[1] KTH - Royal Institute of Technology, Sweden, `cschulte@kth.se`
[2] Programming Systems Lab, Saarland University, Germany, `tack@ps.uni-sb.de`

Abstract. Today's models for propagation-based constraint solvers require propagators as implementations of constraints to be at least contracting and monotonic. These models do not comply with reality: today's constraint programming systems actually use *non-monotonic* propagators. This paper introduces the first realistic model of constraint propagation by assuming a propagator to be *weakly monotonic* (complying with the constraint it implements). Weak monotonicity is shown to be the minimal property that guarantees constraint propagation to be sound and complete. The important insight is that weak monotonicity makes propagation *in combination* with search well behaved. A case study suggests that non-monotonicity can be seen as an opportunity for more efficient propagation.

1 Introduction

When implementing a propagator for a constraint, the propagator must comply with policies mandated by the underlying constraint programming system such that constraint propagation becomes well behaved. The most obvious property is *contraction*: values are removed but never added. A second property is *monotonicity*: a propagator can perform stronger pruning only when being applied to stronger input (fewer values for variables). Contraction captures pruning as the very essence of constraint propagation, while monotonicity guarantees that the same result (the weakest possible) is computed regardless of propagation order.

However, some propagators are non-monotonic. They may compromise between propagation strength and efficiency, like task intervals in scheduling [1] and propagating the `circuit` (Sect. 5) or `multicost-regular` constraint [2]. Other approaches yield non-monotonic propagators due to delaying or adapting propagation [3,4], using randomization [5,6], or approximation [7]. For some propagators, it may just not be obvious whether they are monotonic.

Systems implement some of these non-monotonic propagators, for example Choco, Gecode, Oz, and SICStus Prolog. It is realistic to assume that many more non-monotonic propagators are used, and that many more systems rely on them. Essentially, many systems use non-monotonic constraint propagation while not spelling out the most basic guarantees: Is the result of propagation unique? Is propagation correct? Do two runs of the same problem return the same solution? Do techniques such as recomputation for search still work?

This paper attempts to answer these questions. We show that even non-monotonic propagators have to be monotonic to a certain extent, in order to ensure soundness. This leads to the definition of *weakly monotonic propagators*, a class of propagators that covers approximative, heuristic, and randomized algorithms, while yielding strong enough guarantees to keep propagation sound.

After presenting preliminaries in Sect. 2, the paper contributes the first theory of non-monotonic propagators, based on *weak monotonicity*, that fills the gap between models for propagation and reality (Sect. 3). It analyzes the interaction between propagation of weakly monotonic propagators and search, including an analysis of recomputation (Sect. 4). Finally, it provides a case study that suggests that non-monotonicity should be seen as an opportunity rather than a problem (Sect. 5) and concludes with Sect. 6.

2 Preliminaries

We assume a finite set of *variables* $\mathit{Var} = \{x_1, \ldots, x_n\}$ and a finite set of *values* Val. Constraints are characterized by *assignments* $a \in \mathit{Asn}$ that map variables to values: $\mathit{Asn} = \mathit{Var} \to \mathit{Val}$. A *constraint* $c \in \mathit{Con}$ is a relation over the variables: a set of all assignments that satisfy the constraint, $\mathit{Con} = 2^{\mathit{Asn}}$. Constraints are defined for all variables in Var. Typically, only a subset $\mathrm{vars}(c)$ of the variables is *significant*; the constraint is the full relation for all $x \notin \mathrm{vars}(c)$.

A *domain* $d \in \mathit{Dom}$ maps each variable $x \in \mathit{Var}$ to a set of possible values, the *variable domain* $d(x) \subseteq \mathit{Val}$. A domain d can be identified with the set of assignments $\{a \mid \forall x : \ a(x) \in d(x)\}$. We can therefore identify domains with constraints. In particular, $\{a\}$ is a domain and a constraint for any assignment a.

A domain d_1 is *stronger* than a domain d_2 ($d_1 \subseteq d_2$), iff $\forall x \in \mathit{Var} : d_1(x) \subseteq d_2(x)$. By $\mathrm{dom}(c)$ we refer to the *strongest domain* including all valid assignments of a constraint: $\min \{d \in \mathit{Dom} \mid c \subseteq d\} = \{a \mid \forall x \ \exists b \in c. \ a(x) = b(x)\}$. Note that the minimum exists (domains are closed under intersection) and that not every constraint can be captured by a domain. For a constraint c and a domain d, $\mathrm{dom}(c \cap d)$ refers to removing all values from d not supported by c.

A *constraint satisfaction problem* (CSP) is a pair $\langle d, C\rangle$ of a domain d and a set of constraints C. The *solutions* of a CSP $\langle d, C\rangle$ are all assignments that satisfy all constraints: $\mathrm{sol}(d, C) = \{a \in \mathit{Asn} \mid \{a\} \subseteq d, \ \forall c \in C : a \in c\}$.

3 Weakly Monotonic Propagators

Propagators, sometimes also referred to as constraint narrowing operators or filter functions, serve as implementations of constraints. They are usually defined as contracting functions over domains: $p \in \mathit{Dom} \to \mathit{Dom}$, $p(d) \subseteq d$. Requiring propagators to be contracting is uncontroversial, after all it captures the very essence of constraint propagation and guarantees termination.

Many models of propagation additionally require propagators to be idempotent ($p(p(d)) = p(d)$) and monotonic ($d_1 \subseteq d_2 \Rightarrow p(d_1) \subseteq p(d_2)$). Then, propagators are *closure* or *consequence operators* over the lattice of domains [8,9,10]. Examples for definitions of propagators as closure operators are [11,12,13,14].

Theorem 1. *Given a* propagation problem*, a pair $\langle d, P\rangle$ of a domain d and a set of monotonic propagators P, there is a unique weakest simultaneous fixpoint of all $p \in P$ that is stronger than d. It can be computed by iteration:*

```
$\text{prop}(d, P) \equiv$ **while** $\exists p \in P : \ p(d) \neq d$ **do**
                              $d \leftarrow p(d)$
                          **return** $d$
```

The theorem still holds without idempotency. In practice, it is better to determine the fixpoint status of a propagator dynamically [15]. Similarly, *strength* of propagators is irrelevant (bounds or domain consistency, or forward checking). Consequently, propagators are sometimes defined to be contracting and monotonic. Examples for this definition of propagators are [16,17,15].

In order to relax the definition of propagators, consider how a propagator p can implement a constraint c. The first condition is *correctness*, p must not remove solutions of c: $a \in c \wedge a \in d \Rightarrow a \in p(d)$ for any assignment a and domain d. The second condition is that for an assignment a, p *checks* whether a is a solution of c: $p(\{a\}) = \{a\} \Leftrightarrow a \in c$. The interesting connection between these properties and monotonicity is that *every monotonic propagator implements exactly one constraint.*

Definition 2. *A monotonic propagator p* implements *the constraint defined as the set of assignments* accepted *by the propagator, $\{a \mid p(\{a\}) = \{a\}\}$. This is called the* induced constraint c_p of p.

By definition, a propagator p checks whether assignments are solutions of c_p. And by monotonicity, if $a \in c_p$ and $a \in d$, then $p(\{a\}) \subseteq p(d)$, and hence $a \in p(d)$. Thus, p is correct for c_p. Having correct propagators for constraints, it makes sense to define the set of solutions of a propagation problem $\langle d, P\rangle$ for a domain d and a set of propagators P as the set of solutions of the induced constraints: $\text{sol}(d, P) = \text{sol}(d, \{c_p \mid p \in P\})$.

In Def. 2 monotonicity is used to enforce correctness of the propagator. However, monotonicity was *only used for assignments.* This leads to the central definition used in this paper.

Definition 3. *A function p over domains is called* weakly monotonic *iff $a \in d \Rightarrow p(\{a\}) \subseteq p(d)$ for all assignments a and domains d. A* propagator *is a contracting and* weakly monotonic *function over domains.*

Every weakly monotonic propagator also induces a single constraint. Weak monotonicity yields a minimal definition of propagators, as every propagator can be made weakly monotonic. Given a non-monotonic propagator that is correct for a constraint c, we can turn it into a weakly monotonic propagator that implements c by composing it with a function that checks c on assignments.

Lemma 4. *A monotonic propagator is weakly monotonic.*

The lemma follows directly from the definitions. Conversely, a weakly monotonic propagator is not necessarily monotonic. Assume a propagator p that only prunes the domain if $|d(x)| \in \{1, 3\}$. A domain with $|d(x)| = 2$ can be stronger than a domain with $|d(x)| = 3$ but yield weaker propagation, so p is not monotonic. But it is weakly monotonic, because p does propagate when $|d(x)| = 1$ and thus still checks assignments.

Lemma 5. *Propagation preserves solutions:* $\mathrm{sol}(d, P) = \mathrm{sol}(p(d), P)$ *for* $p \in P$.

Theorem 1 does not hold for non-monotonic propagators. But, $\mathrm{prop}(d, P)$ still terminates, as propagators are contracting and domains are finite. Thus, $\mathrm{prop}(d, P)$ still produces simultaneous fixpoints for all $p \in P$. However, these fixpoints can now be different for different orders of propagator application. Thus, prop turns into a relation. For convenience, we will continue to write $d' = \mathrm{prop}(d, P)$ instead of $d' \in \mathrm{prop}(d, P)$.

Lemma 6. *Assume that* $\mathrm{prop}(d, P) = d_1$ *and* $\mathrm{prop}(d, P) = d_2$. *Then* d_1 *and* d_2 *may not be comparable:* $d_1 \not\subseteq d_2$ *and* $d_2 \not\subseteq d_1$.

Consider propagators p and q with $c_p \equiv (x > 0)$ and $c_q \equiv (x < 2)$. To make them non-monotonic, assume that both p and q only propagate if $|d(x)| \in \{1, 3\}$. Given the domain $d = (x \mapsto \{0, 1, 2\})$, there are two incomparable fixpoints $d_1 = p(q(d)) = (x \mapsto \{0, 1\})$ and $d_2 = q(p(d)) = (x \mapsto \{1, 2\})$.

Although there is no unique weakest fixpoint, the different fixpoints are still well behaved in that they contain all solutions of the original problem. The following lemma will be central for the discussion of search in the next section.

Lemma 7. *If* $\mathrm{prop}(d, P) = d'$, *then* $d' \subseteq d$ *and* $\mathrm{sol}(d, P) = \mathrm{sol}(d', P)$.

This follows from the fact that if some $p \in P$ prunes an assignment a from d, then weak monotonicity guarantees that $p(\{a\}) = \emptyset$. Therefore, a is no solution of c_p, and hence not of $\langle d, P\rangle$, either.

But how are these fixpoints related to the unique weakest fixpoint computed by monotonic propagation? We define the *strongest possible propagator* for a constraint c. This so-called *domain propagator* $\hat{p}_c$ establishes *domain consistency* (also known as generalized arc consistency), it removes all values from all variable domains that cannot be extended to a solution of c. That is, $\hat{p}_c$ returns the strongest domain that contains all solutions of c and d: $\hat{p}_c(d) = \mathrm{dom}(c \cap d)$.

Lemma 8. *Any propagator* p *implementing* c *returns a weaker domain than* $\hat{p}_c$: $\hat{p}_c(d) \subseteq p(d)$. *For any constraint* c, $\hat{p}_c$ *is monotonic.*

Any non-monotonic propagator for a constraint c is weaker than $\hat{p}_c$. Fixpoints are therefore always weaker than those obtained by domain propagation.

The astute reader may have noticed that none of the results depends on propagators being functions, except when being applied to assignments. This is an important insight, as it allows for example *randomized propagation*: on the same domain d, a propagator may return *different* results. In other words, propagators can be relaxed to be contracting and weakly monotonic *relations* over $Dom \times Dom$, as long as they are functional on assignments.

4 Search

A constraint solver interleaves propagation with search. It starts with a propagation problem $\langle d, P \rangle$ and computes a fixpoint. If this fixpoint is neither failed (an empty domain) nor solved (all variables assigned), the solver splits the problem and solves the resulting subproblems recursively. Splitting creates two *branches* (we assume binary search for simplicity), adding propagators to each branch that make the problem simpler (for instance $x = i$ for one branch and $x \neq i$ for the other). Splitting must partition the solution space of the original problem.

Thus, the solver explores a *search tree.* A solver is *sound* if all solutions that it finds by exploring the search tree are solutions of the original problem. It is *complete* if the search tree contains all solutions of the original problem.

If all propagators are monotonic, there is a unique fixpoint for each propagation problem. As long as the addition of propagators to the branches is deterministic, the search tree is completely determined by the initial propagation problem. With non-monotonic propagators, the order of propagation matters. As a result, the shape of the tree also depends on the order of propagation chosen by the solver. As discussed in Lemma 6, the resulting fixpoints may be incomparable, resulting in a different search tree. The good news is that non-monotonic propagation is still correct, it does not remove solutions.

Lemma 9. *A combination of non-monotonic propagation and search is a sound and complete solver for propagation problems. The* set *of solutions found by search is thus determined solely by the original problem to be solved. The* order *of the solutions in the tree depends on the order of propagation.*

For a solver, propagation order may depend on the environment, for instance on memory allocation or other things out of the control of the solver, and may be different for different runs of the same problem. Hence, the first solution may not even be the same between different runs of the same solver. While this may seem inconvenient, non-monotonic propagation is not alone in this respect: parallel search and random restarts share the same properties.

Mozart solves the problem of non-unique fixpoints by fixing the order in which non-monotonic propagators are executed [17]. This technique however clashes with priorities for propagator scheduling, which has proven extremely useful [15].

Recomputation. Recomputation is an important technique for making solvers based on copying efficient [18], and for enabling trailing solvers to perform more advanced search strategies such as best-first search [19].

The main idea is to recompute a node in the search tree from a state further up in the tree, using a *path* that describes the choices that lead to the node. Such a path can consist of a sequence of moves to a child of a node (for example 1.2.2.1.1). Recomputation amounts to redoing the exploration along this path, computing fixpoints for every intermediate step. This is the method described in [18], called *fixpoint recomputation.* Alternatively, a path is a sequence of propagators added by splitting (for example $P_1.P_2.P_3$). Then recomputation adds all propagators to the original propagation problem and computes a single fixpoint. We will refer to this as *path recomputation* (called *batch recomputation* in [20] and *decomposition based search* in [21]).

Fixpoint recomputation fails in the presence of non-monotonic propagators. Assume that splitting adds $x = 1$ on the left and $x \neq 1$ on the right branch. When recomputing the same node, splitting may make an entirely different decision, choosing $y = 1$ and $y \neq 1$ instead. But recomputation just explores the right branch, resulting in an *incomplete search*: possible solutions with $y = 1$ are lost. The Mozart approach of fixing propagator order solves this problem.

Path recomputation is well-behaved even for non-monotonic propagation. In the above example, the splitting decision would be made only once (during the original exploration), and recomputation just adds $x \neq 1$ on the right branch. As propagation preserves solutions, the set of solutions of the right branch is exactly the same as during original exploration. Interestingly, Choi et al. [20] state that path recomputation requires monotonic propagators. This over-cautious assumption underlines the importance of a theory of non-monotonic propagation.

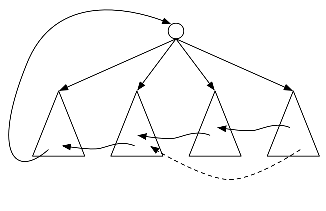
With non-monotonic propagation, the fixpoint after recomputation can differ from the one during exploration. Instead of failure, the solver may recompute a non-failed node, and instead of a solution, a node where some variables are not assigned yet. However, as the set of solutions is the same, further search will only produce failure or the very same solution, respectively. The solver must thus be able to perform less or additional search when recomputing.

Theorem 10. *Constraint propagation with weakly monotonic propagators, combined with search (possibly based on path recomputation), yields a sound and complete solver for propagation problems.*

5 Case Study: Propagating `circuit`

The $\texttt{circuit}(x_1, \ldots, x_n)$ constraint over n finite domain integer variables is true iff the graph with edges $i \rightarrow j$ where $x_i = j$ has a single cycle covering all nodes. Domain-consistent propagation of `circuit` is of course NP-hard.

A simple monotonic propagator for the `circuit` constraint based on the graph G with nodes i and edges $i \rightarrow j$ for all $j \in d(x_i)$ for $1 \leq i \leq n$ is as follows: use a standard `alldifferent` propagation algorithm, as all x_i have to be pairwise distinct; check the mandatory condition that G has only a single strongly connected component. Checking the mandatory condition can be done using DFS on G. However, the DFS spanning tree starting from a node i with $|d(x_i)| > 1$ also offers potential for propagation [22].



The disjoint subtrees explored by DFS are sketched as triangles. For `circuit`, the following must hold: There must be an edge from each subtree to its predecessor subtree, and an edge from the leftmost subtree to the root (otherwise, there is no covering cycle). There must not be any edges between non-neighbor subtrees (such as the dotted edge: the root node must be visited twice if following this edge). This insight can be propagated: prune edges between non-neighbor subtrees, and if there is a single edge between neighbors or from the leftmost subtree to the root, assign the corresponding variable.

This algorithm is obviously non-monotonic: pruning is likely to increase with the number of subtrees in the DFS spanning tree. The DFS spanning tree has more subtrees, if the variable from which DFS starts has more values. Hence, the more values the variable has from which DFS starts, the more pruning.

The propagator is weakly monotonic, as it checks the constraint on assignments. Propagation depends on where DFS starts, so different heuristics for selecting the start variable can be: the *first* variable; a *random* variable; a variable with the *largest* domain. The experiments below confirm that the propagator is indeed non-monotonic: pruning depends on where DFS starts, as witnessed by the different number of fails during search.

The following table shows the effect of non-monotonicity for `circuit`. We have used Gecode 3.0.2 on a MacPro with 2×2.8 GHz Intel Xeons and 8 GB memory running Windows Vista. The runtimes (average of 20 runs, with a coefficient of deviation less than 2% except for random) and number of failures for pruning are relative to just checking by DFS. The `alldifferent` part of `circuit` uses domain-consistency. `knights-`n finds a Knights tour on an $n \times n$ board and `tsp-*` finds an optimal traveling salesman tour.

Example	check		first		largest		random	
	time (s)	fail	time	fail	time	fail	time	fail
`knights-18`	0.36	6 070	86.1%	83.4%	66.9%	57.4%	11.4%	2.2%
`knights-20`	0.05	39	98.1%	92.3%	101.9%	100.0%	100.0%	87.2%
`knights-22`	45.03	543 384	94.4%	93.2%	81.3%	75.6%	0.9%	0.6%
`knights-24`	4.36	42 260	28.4%	26.1%	67.8%	54.0%	4.9%	1.9%
`tsp-br17`	0.83	48 804	98.2%	97.9%	100.4%	98.1%	102.7%	99.1%
`tsp-ftv33`	1423.71	31 013 229	99.8%	99.5%	99.8%	97.9%	96.9%	92.5%

Random variable selection vastly outperforms the other strategies for difficult `knights` instances and shows some speedup for the medium-sized `tsp-ftv33` instance. The coefficient of deviation for runtime and number of failures for random is less than 5% for `tsp` and around 45% for `knights-`n. That is, for `knights-22` and `knights-24` the speedup is almost always at least one order of magnitude thanks to randomization now legalized by weak monotonicity.

6 Conclusion

This paper has introduced a minimal model of propagation based on weakly monotonic propagators and has clarified the properties of propagation and search based on the model. By this, the paper for the first time gives a model to capture the essential properties of many constraint programming systems that use non-monotonic propagators. The hope is that non-monotonic propagation is seen as a general opportunity for more efficient propagation rather than a problem that is best ignored.

Acknowledgements. We thank Nicolas Beldiceanu, Mats Carlsson, Mikael Lagerkvist, Gert Smolka, and Peter Stuckey for insightful discussions. Part of this work was done while Guido Tack was at NICTA Victoria Laboratory, Melbourne, Australia.

References

1. Baptiste, P., Le Pape, C.: A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling. In: IJCAI. (1995) 600–606
2. Menana, J., Demassey, S.: Sequencing and counting with the multicost-regular constraint. In: CP-AI-OR. LNCS 5547, Springer (2009) 178–192
3. Katriel, I.: Expected-case analysis for delayed filtering. In: CP-AI-OR. LNCS 3990, Springer (2006) 119–125
4. Stergiou, K.: Heuristics for dynamically adapting propagation. In: ECAI. (2008) 485–489
5. Katriel, I., Van Hentenryck, P.: Randomized filtering algorithms. Technical Report CS-06-09, Brown University, Providence, RI, USA (2006)
6. Mehta, D., van Dongen, M.R.C.: Probabilistic consistency boosts MAC and SAC. In: IJCAI. (2007) 143–148
7. Sellmann, M.: Approximated consistency for Knapsack constraints. In: CP. LNCS 2833, Springer (2003) 679–693
8. Ward, M.: The closure operators of a lattice. Annals of Mathematics **43**(2) (1942) 191–196
9. Tarski, A.: Fundamentale Begriffe der Methodologie der deduktiven Wissenschaften. I. Monatshefte für Mathematik **37**(1) (1930) 361–404
10. Tarski, A.: V. In: Logic, semantics, metamathematics. Second edn. Hackett Publishing Company (1983) 60–109
11. Saraswat, V.A., Rinard, M.C., Panangaden, P.: Semantic foundations of concurrent constraint programming. In: POPL. (1991) 333–352
12. Benhamou, F., McAllester, D.A., Van Hentenryck, P.: CLP(Intervals) revisited. In: ILPS, The MIT Press (1994) 124–138
13. Van Hentenryck, P., Saraswat, V.A., Deville, Y.: Constraint processing in cc(FD). Technical report, Brown University (1991)
14. Apt, K.R.: Principles of Constraint Programming. Cambridge University Press (2003)
15. Schulte, C., Stuckey, P.J.: Efficient constraint propagation engines. ACM Trans. Program. Lang. Syst. **31**(1) (2008) 2:1–2:43
16. Benhamou, F.: Heterogeneous Constraint Solving. In: ALP. LNCS 1139, Springer (1996) 62–76
17. Müller, T.: Constraint Propagation in Mozart. Doctoral dissertation, Universität des Saarlandes, Saarbrücken, Germany (2001)
18. Schulte, C.: Programming Constraint Services. LNAI 2302. Springer (2002)
19. Perron, L.: Search procedures and parallelism in constraint programming. In: CP. LNCS 1713, Springer (1999) 346–360
20. Choi, C.W., Henz, M., Ng, K.B.: Components for state restoration in tree search. In: CP. LNCS 2239, Springer (2001)
21. Michel, L., Van Hentenryck, P.: A decomposition-based implementation of search strategies. ACM Trans. Comput. Logic **5**(2) (2004) 351–383
22. Carlsson, M.: Personal communication (February 2007)