Finite Domain Constraint Programming Systems

Demetrio Girardi

April 1st, 2009

Introduction

On a high level, the steps involved in solving a constraint satisfaction problem are *constraint propagation* and the actual *search.*

A useful constraint programming system must be able to effectively perform the aforementioned, as well as providing an adequate *interface* to the user.

In this talk we discuss how to build such a system, restricting our scope to propagation-based, finite domain constraint programming systems.

Interface

The interface is the language exposed to the user; this can be an existing programming language (e.g. Prolog) or an extension of one (like a c library); nothing prevents the inspired programmer from inventing his own language and building an interpreter for it.

At a very minimum the language must allow the definition of variables and constraints; additional language tools might be provided for a number of different things, such as the choice of search techniques, definition of new types of constraints and so on.

Constraint propagation

Representing constraints

Extensionally, a constraint c over a set of variables X is defined as a set of value assignments over X (our usual tuple notation). We might then decide to choose this as our representation and use some propagation algorithm on top of it.

This is inefficient, both because the number of tuples can quickly grow unmanageable, and because efficient propagation algorithms use knowledge on the structure of specific constraints (which is obscured in the tuple representation).

Instead of *representing* constraints we *implement* them, or rather we implement propagation for every constraint.

Propagators

A *propagator* p is a procedure that takes a set of variables X as parameters and removes values from the domain of some of them by means of a filtering algorithm designed for the constraint that p is implementing.

Formally, p is a function that maps domain to domains that is decreasing ($\forall D : p(D) \sqsubseteq D$) and monotonic ($D_1 \sqsubseteq D_2 \implies p(D_1) \sqsubseteq p(D_2)$).

A propagator is said to be *correct* for a constraint c when it does not remove assignments for c:

$$\forall D : \{a \in D\} \cap c = \{a \in p(D)\} \cap c$$

where $\{a \in D\}$ is the set of all possible value assignments in D.

A set P of propagators is said to be *checking* for c if only solutions to c are fixpoints of P when all variables are fixed:

$$\forall D \text{ such that } \forall x \in vars(c) \, |D(x)| = 1 : p(D) = D \, \forall p \in P \iff a \in c$$

where a is the unique value assignment over $vars(c)$.

Example: propagator

Consider the constraint $c: x_1 \leq x_2$. We can define a propagator p as:

$p(D)$:
  $D' \leftarrow D$
  for all $n \in D(x_1)$ **do**
    if n is greater than the maximum value in $D(x_2)$ **then**
      remove n from $D'(x_1)$
  return D'

p is correct for c: for every assignment $a = (x_1 \mapsto n_1, x_2 \mapsto n_2) \in D$, if $n_1 \leq n_2$ then $a \in p(D)$;

p is checking for c: for every domain D such that $D(x_1) = \{n_1\}$ and $D(x_2) = \{n_2\}$, $p(D) = D$ iff $n1 \leq n2$.

x_1 is said to be an *output variable* of p, x_2 an *input variable.*

Performing propagation

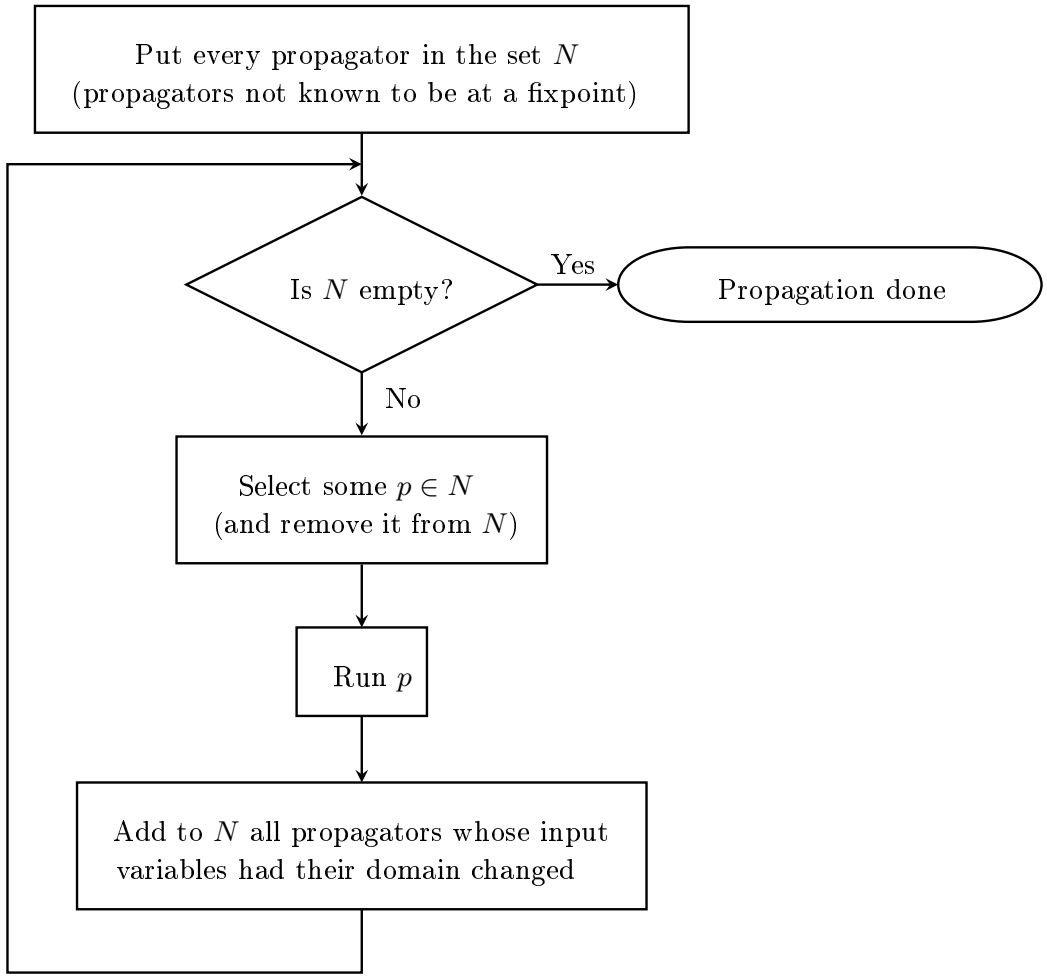
Once we have implemented a set of propagators that is correct and checking for every constraint in our CSP, a *propagation engine* is in charge of actually performing propagation.

A propagation engine for a set of propagators P and an initial domain D_0 computes the greatest mutual fixpoint of all $p \in P$.

propagate(P_f, P_n, D) :

$N \leftarrow P_n$
$P \leftarrow P_f \cup P_n$
while $N \neq 0$ **do**
  $p \leftarrow \text{select}(N)$
  $N \leftarrow N - \{p\}$
  $D' \leftarrow p(D)$
  $M \leftarrow \{x \in X | D(x) \neq D'(x)\}$
  $N \leftarrow N \cup \{p' \in P | \text{input}(p') \cap M \neq 0\}$
  $D \leftarrow D'$
return D

A simple propagation engine



Example: propagation

Consider the propagator p defined as

$p(D, x_1, x_2)$:
  for all $n \in D(x_1)$ **do**
    if n is greater than the maximum value in $D(x_2)$ **then**
      remove n from $D(x_1)$

Now consider the domain D : $D(x_1) = \{0, 2, 6\}, D(x_2) = \{-1, 2, 4\}$. A run of our simple propagation engine, for an initial domain $D_0 = D$ and a set of propagators $P = \{p_1 = p(D, x_1, x_2), p_2 = p(D, x_2, x_1)\}$, would look like this:

Step	Selected p	$D(x_1)$	$D(x_2)$	N
0	-	$\{2, 3, 6\}$	$\{-1, 2, 4\}$	$\{p_1, p_2\}$
1	p_1	$\{2, 3\}$	$\{-1, 2, 4\}$	$\{p_2\}$
2	p_2	$\{2, 3\}$	$\{-1, 2\}$	$\{p_1\}$
3	p_1	$\{2\}$	$\{-1, 2\}$	$\{p_2\}$
4	p_2	$\{2\}$	$\{-1, 2\}$	$\{\}$

Improving our engine

Quite often a propagator remains at a fixpoint despite some change in the domain of its input variables: our simple propagating engine does not know this, and might repeatedly execute propagators that have no effect.

Knowledge about the behavior of a propagator can be taught to the engine in several ways. Common properties of propagators that are useful in this regard are *idempotence* and *entailment*.

> A propagator p is said *idempotent* if $\forall D : p(p(D)) = p(D)$.

Running an idempotent propagator twice in a row will have no additional effect, but our engine might decide to do so if the first run modified the domain of its input variables.

> A propagator p is said *entailed* by a domain D if $\forall D' \sqsubseteq D : p(D') = D'$.

Running an entailed propagator will never yield any benefit.

Idempotence and entailment

Consider the propagator

$p(D)$:
  for all $n \in D(x_1)$ **do**
    if n is greater than the maximum value in $D(x_2)$ **then**
      remove n from $D(x_1)$
  for all $n \in D(x_2)$ **do**
    if n is less than the minimum value in $D(x_1)$ **then**
      remove n from $D(x_2)$

x_1 and x_2 are both input and output variables for p, but after every execution we can safely remove p from N, as p is idempotent.

Moreover, if every value in $D(x_1)$ is less than every value in $D(x_2)$, p is entailed and we can avoid running it no matter how other propagators might modify D.

Events

Often a propagator is able to decide whether it still is at a fixpoint depending on *how* the domain has changed. Propagation events describe different ways in which the domain can change.

Usual events are:

- a variable has become fixed,
- the minimum/maximum of a variable has changed,
- the domain of a variable has changed.

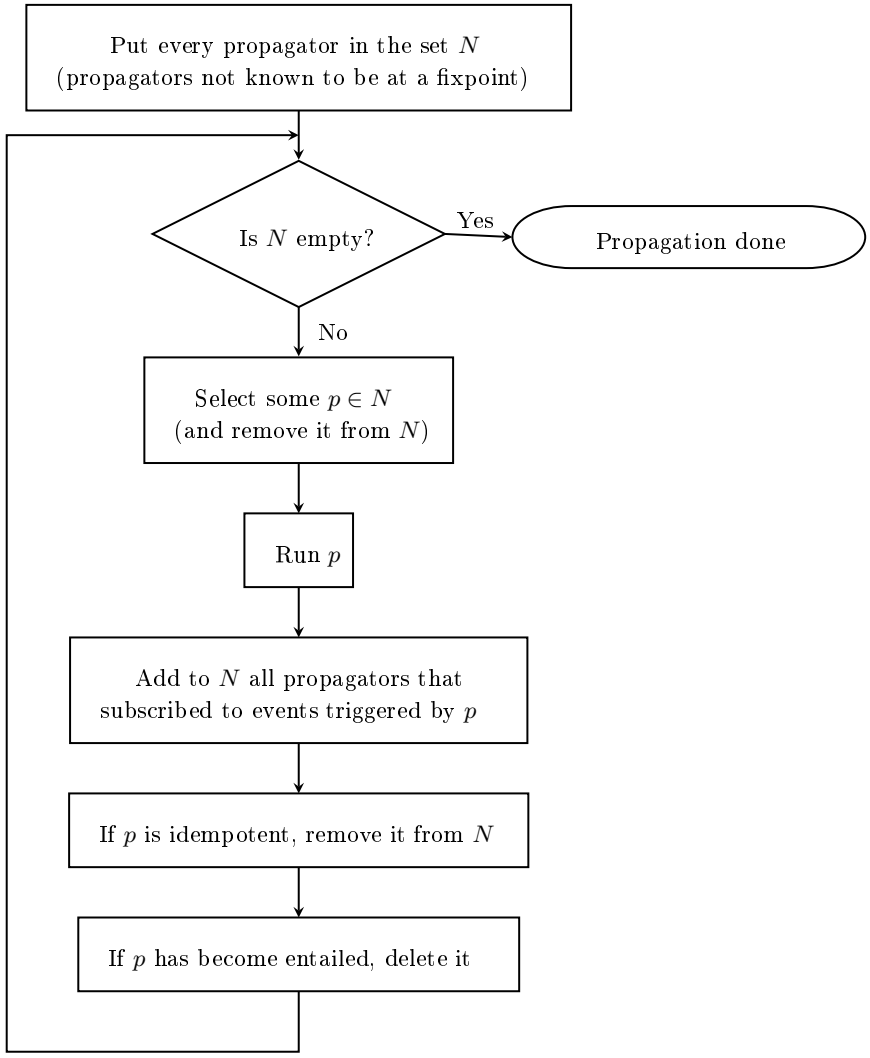
A propagation engine can reduce running time by keeping track of events triggered on domain changes and schedule for execution only propagators that subscribe to those events.

Events (2)

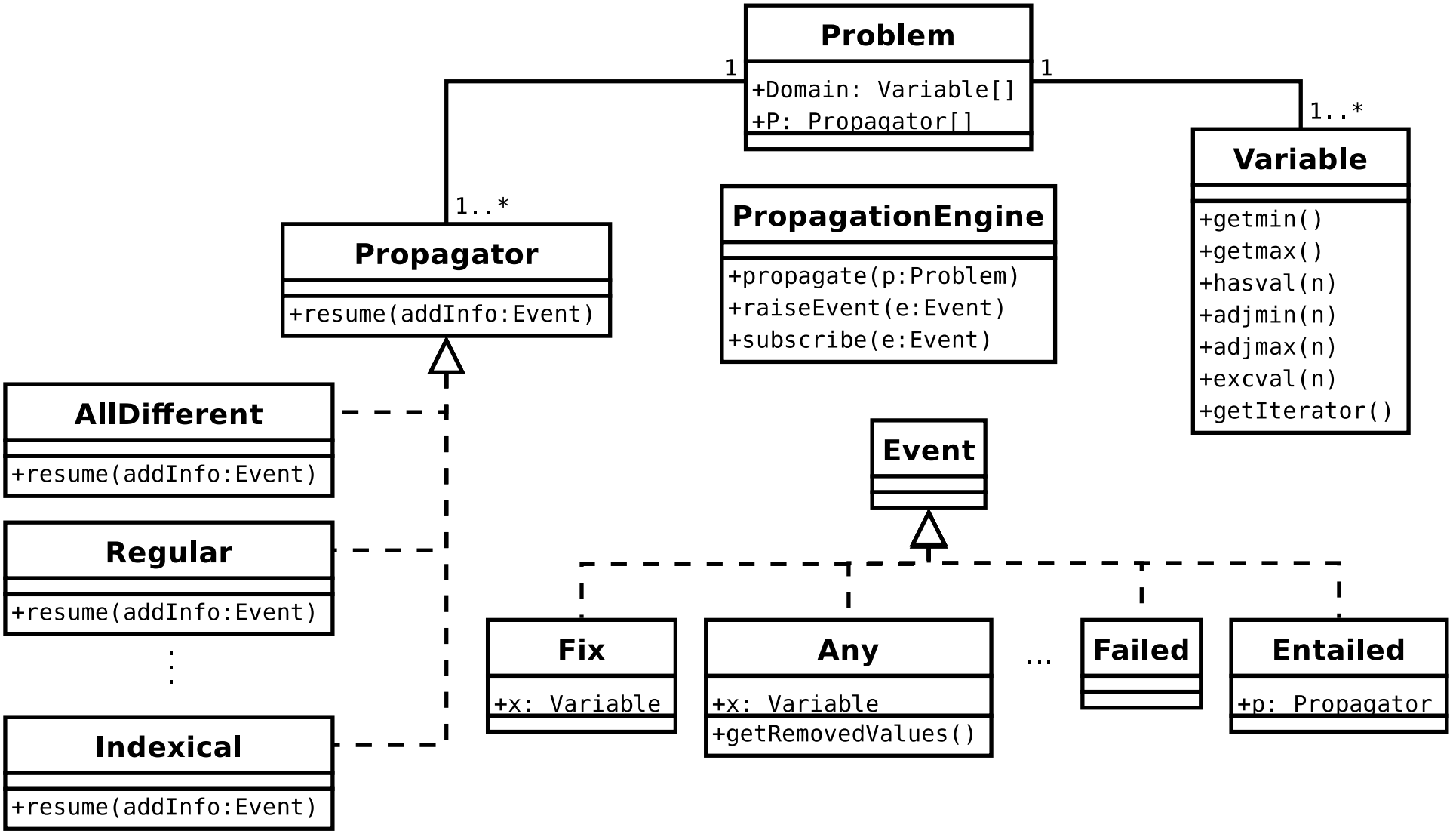
$p(D)$:
  for all $n \in D(x_1)$ **do**
    if n is greater than the maximum value in $D(x_2)$ **then**
      remove n from $D(x_1)$
  for all $n \in D(x_2)$ **do**
    if n is less than the minimum value in $D(x_1)$ **then**
      remove n from $D(x_2)$

Example: p remains at a fixpoint unless the maximum value in $D(x_2)$, or the minimum value in $D(x_1)$, changes; it is useless to run it when the domain changes without affecting those values.

A revised propagation engine



Propagation system architecture



Search

Branching strategies

Recall talk 4: a node in the search tree is defined as a set s of *branching constraints*, and extending s generates branches $s_1 = s \cup \{c_1\}, \ldots, s_n = \cup\{c_n\}$ where $\{c_1, \ldots, c_n\}$ is an exhaustive set of branching constraints ($C \wedge D \models c_1 \vee \ldots \vee c_n$).

Example

fixing a variable x to value $n \in D(x)$ generates two branches (subproblems) where $C_1 = C \cup \{x = n\}$ and $C_2 = C \cup \{x \neq n\}$.

The set $\{c_1, \ldots, c_n\}$ can be called a set of *alternatives* available at a certain point in the search. A branching strategy defines how to compute the alternatives.

A possible branching strategy is: choose some non-fixed variable x and fix it to $n = min\,(D\,(x))$.

Branching strategies (2)

In our architecture constraints are translated into propagators. We can then implement a branching strategy with a set of sets of propagators $\{S_1, \ldots, S_n\}$ that, at any point in the search, are correct and checking for the available alternatives. Our system can then extend a node by extending the list of propagators P for every branch: $P_i = P \cup S_i$.

$p_1(D)$:
  select the non-fixed variable x with the smallest domain
  for all $n \in D(x)$ **do**
    if $n > min(D(x))$ **then**
      remove n from $D(x)$

$p_2(D)$:
  select the non-fixed variable x with the smallest domain
  remove $min(D(x))$ from $D(x)$

$\{\{p_1\}, \{p_2\}\}$ implement a 2-way branching using the **dom** ordering heuristic.

Exploration strategies

The most common exploration strategy is probably depth-first, but systems can support a variety of different search techniques.

Some of them have been discussed in talk 4 and 5: sophisticated backtracking, randomization and restart. Other notable strategies are discrepancy-based search (LDS, DDS) and resource-bounded search.

State restoration

To perform search a system must be able to backtrack to a previous node in the tree.

In our architecture a node consists of a domain and a set of propagators (including their internal state); a simple approach is keeping a copy of every node, i.e. cloning the relevant objects before branching.

Other possible approaches are *trailing* and *recomputation*.

Trailing

To perform *trailing*, a system keeps track of every state-changing operation and stores information on how to undo it.

At low-level (e.g. a Prolog interpreter), a state-changing operation is a memory update; a trail consists then of a list of memory locations together with their old values.

At higher levels (e.g. a Java program), trailing is more difficult to achieve, as it requires every data structure to implement it separately.

Recomputation

If we keep track of the path from the tree root to the current node, we can backtrack to any previous node by computing it again.

This is usually too expensive timewise, but we can combine recomputation with copying or trailing to reduce space requirements: the idea is keeping a copy (or trail) for only part of the search tree, and recompute nodes that have been forgotten.