DM877

Constraint Programming

Propagation Events and Implementations

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

Outline

1. Generic Rules Iteration

2. Systems

Algorithms for constraint propagation:

- scheduling steps of atomic reduction
- termination criterion: local consistency

- How to schedule the application of reduction rules to guarantee termination?
- How to avoid (at low cost) the application of redundant rules?
- Have all derivations the same result?
- How can we characterize it?

Propagators

- Given $\mathcal{P}$ a reduction rule is a function f from $\mathcal{S}_{\mathcal{P}}$ to $\mathcal{S}_{\mathcal{P}}$ for all $\mathcal{P}' \in \mathcal{S}_{\mathcal{P}}$, $f(\mathcal{P}') \in \mathcal{S}_{\mathcal{P}}$.
(most cases take care of a single variable and a single constraint)
- Given $\mathcal{P}$ a propagator f for C is a reduction rule from $\mathcal{S}_{\mathcal{P}}$ to $\mathcal{S}_{\mathcal{P}}$ that tightens only domains independently of the constraints other than C.
- A propagator f is correct for C iff it does not remove any assignment for C:
$\{a \in D\} \cap C = \{a \in f(D)\} \cap C$

Systems consider set of propagators to implement a constraint
(However global constraints have a single propagator.)

Example

$C \equiv x_1 \leq x_2 + 1$

$$f(D, x_1) = p(D)(x_1) = \{n \in D(X_1) \mid n \leq \max_D\{x_2\} + 1\}$$

$\text{input}(p) = x_2$, $\text{output}(p) = x_1$

Propagators

Properties of propagators:

- A propagator f is:
  - contracting (or decreasing): for all $\mathcal{P} \in \mathcal{S}_{\mathcal{P}}$: $f(\mathcal{P}) \leq \mathcal{P}$, that is: $\mathcal{D}(f(\mathcal{P})) \subseteq \mathcal{D}(\mathcal{P})$
- A propagator f can be:
  - monotonic if $\mathcal{P}_1 \leq \mathcal{P}_2 \Rightarrow f(\mathcal{P}_1) \leq f(\mathcal{P}_2)$
  - commuting if $fg(\mathcal{P}) = gf(\mathcal{P})$
  - idempotent for $\mathcal{P}$ if $f(f(\mathcal{P})) = f(\mathcal{P})$ (weak: for some $\mathcal{P} \in \mathcal{S}_{\mathcal{P}}$)
  - subsumed (or entailed) by $\mathcal{P}$ iff $\forall \mathcal{P}_1 \leq \mathcal{P} : f(\mathcal{P}_1) = \mathcal{P}_1$ (strong: for all $\mathcal{P} \in \mathcal{S}_{\mathcal{P}}$)
    Eg:

    $$f(D, x) = D(x) \cap \{1, 2, 3\}$$

    implementing the domain constraint $x \in \{1, 2, 3\}$. After f has been executed once, there is no point to execute f again as for all D' $D' \leq f(D) \implies f(D') = D'$
    (particular case when all variables are instantiated)

Example

$$\mathcal{P}_1 = \langle X = (x_1, x_2); D_1(x_1) = \{1, 2\}, D_1(x_2) = \{2\}; \mathcal{C} \equiv \{x_1 = x_2\}\rangle$$

$$\mathcal{P}_2 = \langle X = (x_1, x_2); D_2(x_1) = \{1, 2, 3\}, D_2(x_2) = \{2\}; \mathcal{C} \equiv \{x_1 = x_2\}\rangle$$

f removes values from $D(x_1)$ that have no support on C if less than half of them have support.

- $f(D_2(x_1)) = \{2\}$
- $D(f(\mathcal{P}_1)) \not\subseteq D(f(\mathcal{P}_2))$ whereas $D(\mathcal{P}_1) \subseteq D(\mathcal{P}_2)$ $\rightsquigarrow$ not monotonic

g removes one of the values from x_1 that have no support on C if such a value exists.

- $g(D_2(x_1)) = \{1, 2\}$, $gg(D_2(x_1)) = \{2\}$
- $gg(\mathcal{P}_2) \neq g(\mathcal{P}_2)$

- Iteration: Let $\mathcal{P} = \langle X, \mathcal{D}, \mathcal{C} \rangle$ and $F = \{f_1, \ldots, f_k\}$ a finite set of propagators on $\mathcal{S}_{\mathcal{P}}$. An iteration of F on $\mathcal{P}$ is a sequence $\langle \mathcal{P}_0, \mathcal{P}_1, \ldots \rangle$ of elements of $\mathcal{S}_{\mathcal{P}}$ defined by

$$\begin{aligned} \mathcal{P}_0 &= \mathcal{P} \\ &\vdots \\ \mathcal{P}_j &= f_{n_j}(\mathcal{P}_{j-1}) \end{aligned}$$

  where $j > 0$ and $n_j \in [1, \ldots, k]$.
- $\mathcal{P}$ is stable for F iff $\forall f \in F, f(\mathcal{P}) = \mathcal{P}$
- There may be several stable $\mathcal{P}$ but if F are monotonic then unique
- Let $\mathcal{P} = \langle X, \mathcal{D}, \mathcal{C} \rangle$ and $F = \{f_1, \ldots, f_k\}$. If $\langle \mathcal{P}_0, \mathcal{P}_1, \ldots \rangle$ is infinite iteration of F where each $f \in F$ is activated infinitely often then there exists $j \geq 0$ such that $\mathcal{P}_j$ is stable for F ($\equiv j$ is finite!)
- If $\mathcal{P}$ is stable for F then it is its weakest simultaneous fixed point (greatest mutual fixed point of all propagators).
  A strongest simultaneous fixed point would be a solution (hence possibly not unique) which would not violate solution preservation

Iteration of Reduction Rules

```
procedure Generic-Iteration(N, F);
  G ← F;
  while G ≠ ∅ do
    select and remove g from G;
    if N ≠ g(N) then
      update(G);
      N ← g(N);

/* update(G) adds to G at least all functions f in F \ G for which
g(N) ≠ f(g(N)) */
```

If the propagator is contracting then Generic-Iteration terminates.
If propagator is monotonic then the final result does not change with the order in which propagators are applied.

If propagators in addition to monotonic are also idempotent and commutative then:

```
procedure Direct-Iteration(N, F);
  G ← F;
  while G ≠ ∅ do
    select and remove g from G;
    N ← g(N);
```

Iteration of Reduction Rules

Example

Recall for arc consistency:
Arc Consistency rule 1 (propagator):

$$\frac{\langle C; x \in D(x), y \in D(y)\rangle}{\langle C; x \in D'(x), y \in D(y)\rangle}$$

where $D'(x) := \{a \in D(x) \mid \exists b \in D(y), (a, b) \in C\}$

Set of propagators $F_{AC} = \{f_{ij} \mid x_i \in X, c_j \in \mathcal{C}\}$ all monotonic. $\Rightarrow$ terminates in arc consistency closure, which is fixed point for F_{AC}.

Improvements

Generic iteration is an example of propagator engine
What makes it naive?

- Termination relies on strict contraction
- We always have to check all propagators for one that can strictly contract

Ideas:

- Maintain propagators which are known to be at fixpoint
- Look at the variables that propagators actually compute with Dependency-directed propagation

Fixpoint knowledge avoids useless execution (idempotence, subsumption) knowledge provided by propagator

Improvements

Generic iteration is an example of propagator engine

propagate(P_f,P_n,D)

$N \leftarrow P_n$
$P \leftarrow P_f \cup P_n$
while $N \neq \emptyset$ **do**
  $p \leftarrow \mathsf{select}(N)$
  $N \leftarrow N - \{p\}$
  $D' \leftarrow p(D)$
  $M \leftarrow \{x \in \mathcal{V} \mid D(x) \neq D'(x)\}$
  $N \leftarrow N \cup \{p' \in P \mid \mathrm{input}(p') \cap M \neq \emptyset\}$
  $D \leftarrow D'$
return D

P_f is set of propagators at fixed point (idempotent or subsumed)

Scheduling p: adding a propagator to the set N (not known to be at fixed point). Yet undefined how a propagator is chosen from N

Note: search can be seen as doing incremental propagation

Improvements: Events

Most solvers implement arithemitc-oriented propagators
$\rightsquigarrow$ a reduction of a domain of a variable has different implications depending on the type of reduction

Four types of Events:

- `Any` or `RemValue`: when a value v is removed from $D(x_i)$
- `Min` or `IncMin`: when the minimum value of $D(x_i)$ increases
- `Max` or `DecMax`: when the maximum value of $D(x_i)$ decreases
- `Fix` or `Instantiate`: when $D(x_i)$ becomes a singleton

AC3 like

Modified AC3 to handle parameter `Mtype` (modification type)

function `Constraint-Propag`(**in** X: set): **Boolean** ;
begin
  foreach $c \in C$ **do** perform `init-propag` on c and update Q with relevant events;
  while $Q \neq \emptyset$ **do**
    select and remove $(x_i, c, x_j, Mtype)$ from Q;
    if *Revise*$(x_i, c, (x_j, Mtype), Changes)$ **then**
      if $D(x_i) = \emptyset$ **then** return **false** ;
      foreach $c' \in \Gamma^C(x_i), Mtype \in Changes$ **do**
        foreach $x_j \in X(c'), j \neq i$ **do** $Q \leftarrow Q \cup \{(x_j, c', x_i, Mtype)\}$;
  return **true** ;
end
/* $\Gamma^C(x_i)$ is the set of constraints with x_i in their scheme */

The presence of $(x_j, c, x_i, \mathtt{Mtype})$ in Q means that x_j should be revised on c because of an `Mtype` change in $D(x_i)$.

Process constraint propagation differently according to the type of event

function revise(**inout** x_i; **in** $c \equiv x_{k_1} \leq x_{k_2}$; **in** $(x_j, Mtype)$; **out** $Changes$)**:**
  Boolean ;
  $Changes \leftarrow \emptyset$;
  switch $Mtype$ **do**
    case *RemValue*
      nothing;
    case *IncMin*
      if $j = k_1$ **then** remove all $v < min_D(x_j)$ from $D(x_i)$;
    case *DecMax*
      if $j = k_2$ **then** remove all $v > max_D(x_j)$ from $D(x_i)$;
    case *Instantiate*
      if $j = k_1$ **then** remove all $v < min_D(x_j)$ from $D(x_i)$;
      else remove all $v > max_D(x_j)$ from $D(x_i)$;
  $Changes \leftarrow$ the types of changes performed on $D(x_i)$;

Also: for a certain constraint it can be that a given event cannot alter the other variables of the constraint. Hence it makes sense to:

6: **foreach** $c' \in \Gamma^{c}_{\texttt{Mtype}}(x_i)$, $\texttt{Mtype} \in Changes$ **do** ...

Example. Let $c \equiv x_1 \leq x_2$. The only events that require propagation are `IncMin` and `Instantiate` on x_1 , and `DecMax` and `Instantiate` on x_2.

select and remove $(x_i, c, x_j, Mtype, \Delta_j)$ from Q;
if `Revise`$(x_i, c, (x_j, Mtype, \Delta_j), Changes, \Delta_i)$ **then**
  if $D(x_i) = \emptyset$ **then** return **false**;
  foreach $c' \in \Gamma^{C}_{Mtype}(x_i), Mtype \in Changes$ **do**
    foreach $x_j \in X(c'), j \neq i$ **do** $Q \leftarrow Q \cup \{(x_j, c', x_i, Mtype, \Delta_i)\}$

---

function **`revise`(inout** x_i; **in** $c \equiv x_{k_1} = x_{k_2} + m$; **in** $(x_j, Mtype, \Delta_j)$;
out $Changes$; **out** Δ_i**): Boolean** ;

$Changes \leftarrow \emptyset$;
switch $Mtype$ **do**
  case *`RemValue`*
    if $j = k_1$ **then** **foreach** $v \in \Delta_j$ **do** remove $(v - m)$ from $D(x_i)$;
    else **foreach** $v \in \Delta_j$ **do** remove $(v + m)$ from $D(x_i)$;
  case *`IncMin`*
    if $j = k_1$ **then** remove all $v < min_D(x_j) - m$ from $D(x_i)$;
    else remove all $v < min_D(x_j) + m$ from $D(x_i)$;
  case *`DecMax`*
    if $j = k_1$ **then** remove all $v > max_D(x_j) - m$ from $D(x_i)$;
    else remove all $v > max_D(x_j) + m$ from $D(x_i)$;
  case *`Instantiate`*
    if $j = k_1$ **then** assign $min_D(x_j) - m$ to x_i;
    else assign $min_D(x_j) + m$ to x_i;
$Changes \leftarrow$ the types of changes performed;
$\Delta_i \leftarrow$ all values removed from $D(x_i)$;

More Optimization

Priorities
Choose propagator

- according to cost: cheapest first
- according to expected impact
- general (queue): last-in last-out (starvation avoided), first-in first-out

Propagator Rewriting

Another observation:
propagator for

$$\max(x, y) = z$$

and values for x are smaller than for y
Replace by propagator for $y = z$

Outline

1. Generic Rules Iteration

2. Systems

Architecture

- Detecting failure and entailment
- Domains: single data structure continously updated.
  constraint store $\equiv$ domain extension $\mathcal{D}$
- State restoration
- Finding dependent propagators (compute events and find propagators)
- Variables for propagators

Propagation Services

- Events
- Selecting next propagator

Variable Domains

- Domain representation
  range sequence: $s = \{[n_1, m_1], \ldots, [n_k, m_k]\}$ (singly/doubly linked lists) bit vector
- Value operations `x.getmin()`, `x.getmax()`, `x.hasval()`, `x.adjmin(n)`, `x.adjmax(n)`, `x.excval(n)`
- Iterators:

```
for (IntVarValues i(x); i(); ++i)
  std::cout << i.val() << ' ';

for (IntVarRanges i(x); i(); ++i)
  std::cout << i.min() << ".." << i.max() << ' ';
```

- Domain operations
- Subscriptions (p is executed whenever the domains of one of its variables changes according to an event). Options:
  - list E_i, p_i pair event propagator that requires execution
  - a list for each event and one for each propagator
  - array of propagators partitioned by events

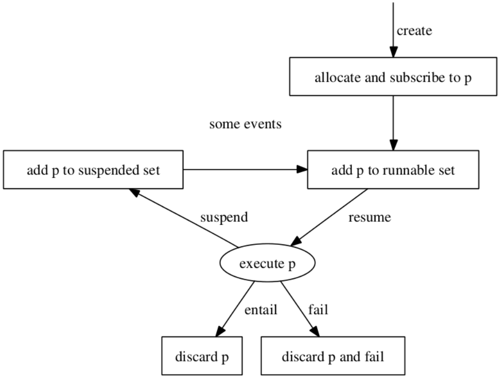
Operations	**Range sequence**	**Bitvector**
x.getmin()	$O(1)$	$O(1)$
x.getmax()	$O(1)$	$O(1)$
x.hasval(n)	$O(r)$	$O(1)$
x.adjmin(n)	$O(r)$	$O(1)$
x.adjmax(n)	$O(r)$	$O(1)$
x.excval(n)	$O(r)$	$O(v)$
i.done()	$O(1)$	$O(v)$
i.value()	$O(1)$	$O(1)$
i.next()	$O(1)$	$O(v)$

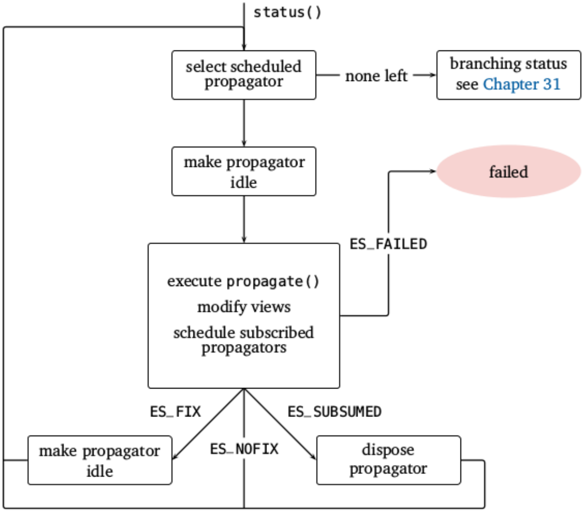
Propagators

Piece of software with some private state that implements a constraint C over some variables or *parameters*

The algorithm implemented is called filtering algorithm. It uses value and domain operations and raises events that cause scheduling of other propagators

Life cycle




status()
select scheduled propagator
none left
branching status see Chapter 31
make propagator idle
failed
ES_FAILED
execute propagate()
modify views
schedule subscribed propagators
ES_FIX
ES_SUBSUMED
ES_NOFIX
make propagator idle
dispose propagator

- Idempotency: it may be costly and difficult to guarantee. Some propagators return a state:
  - fixpoint (weak idempotent, ie, with respect to x rather than for all),
  - no fixpoint (we do not know),
  - subsumed (entailed),
  - failure.

References

Apt K.R. (2003). **Principles of Constraint Programming**. Cambridge University Press.

Barták R. (2001). **Theory and practice of constraint propagation**. In *Proceedings of CPDC2001 Workshop*, pp. 7–14. Gliwice.

Bessiere C. (2006). **Constraint propagation**. In *Handbook of Constraint Programming*, edited by F. Rossi, P. van Beek, and T. Walsh, chap. 3. Elsevier. Also as Technical Report LIRMM 06020, March 2006.

Schulte C. (2011). **Course notes, constraint programming (id2204), vt 2012**. Unpublished.

Schulte C. and Carlsson M. (2006). **Finite domain constraint programming systems**. In *Handbook of Constraint Programming*, edited by F. Rossi, P. van Beek, and T. Walsh. Elsevier.