



# Incrementality

## The idea

- ▶ The complexity of a FA typically considers one run from scratch.
- ▶ When the FA is rerun after some domain update, we can do much better than that.
- ▶ It's more interesting to consider the amortized complexity over a branch of the search tree.

# Incrementality

## Some problems

- ▶ Only valid while in the same branch.
- ▶ Normally, recompute from scratch upon backtracking.
- ▶ If too much has changed, incrementality costs more than recomputation.
- ▶ Adaptive approaches are possible.
- ▶ If  $\# \text{modifications} < \# \text{remaining things}$ , incrementality is a good idea [Régis's rule of thumb].

# Some opportunities

- ▶ FA can maintain a private **state**.
- ▶ Maintain partitioning of parameters as (nonground, ground, done).
- ▶ Detect in  $O(1)$  time that nothing can be pruned.
- ▶ `all_different` reuses and augments the previous graph and matching.
- ▶ `gcc` reuses and augments the previous graph and flow.
- ▶ `case` does not revisit failed DAG nodes.
- ▶ `disjoint2` maintains **witnesses** for each rectangle to avoid sweeps.
- ▶  $\leq_{\text{lex}}$  remembers where it exited the automaton.

# Idiosyncracies of `library(clpfd)` in SICStus Prolog

But not terribly SICStus specific

- ▶ Embedded in Prolog  $\Rightarrow$  **trailing** as opposed to copying, domains are subject to backtracking.
- ▶ FA must check whether its private state is still **valid** (no failure since last time).
- ▶ FA is not told **which domains were updated**, let alone **their old values**.
- ▶ Domain variables and dependency lists reside in “Prolog memory”.

# Idiosyncracies of `library(clpfd)` in SICStus Prolog

But not terribly SICStus specific

## FA outline (on resumption)

1. If there has been a failure since last time, then recompute everything that depends on the current domains.
2. Import **relevant** domain variables from Prolog to C.
3. Impose consistency on the imported variables.
4. Export **updated** domain variables to Prolog from C.
5. Optionally signal failure or entailment.

# Filtering algorithms and daemons

- ▶ FA does not know **which domains were updated**  
⇒ FA is somewhat heavy-weight.
- ▶ A *daemon* is a light-weight procedure for (a) incremental state updates and (b) detecting whether any pruning is possible.
- ▶ Suppose constraint  $c$  has parameter  $x$ .
- ▶ By default, when  $\text{dom}(x)$  ( $\min(x)$ ,  $\max(x)$ ) is updated, the FA implementing  $c$  is scheduled.
- ▶ Alternatively, a daemon for  $c$  is invoked **immediately**:
  - ▶ The daemon knows which parameter  $x$  (exactly one) was updated.
  - ▶ The daemon may update the private state.
  - ▶ The daemon cannot update any domain variables.
  - ▶ The daemon may schedule the FA.

# Example: scalar products

## Reference

W. Harvey, J. Schimpf, *Bounds Consistency Techniques for Long Linear Constraints*, proc. TRICS workshop at CP'2002, Ithaca, NY.

## Constraint

$$C \equiv \sum_{i=1}^n a_i x_i \leq b \quad \text{w.l.o.g. } a_i > 0, x_i \text{ distinct}$$

# Example: scalar products

Let:

$$l_j = a_j(\bar{x}_j - x_j)$$
$$F = b - \sum_{i=1}^n a_i \underline{x}_i$$

**Note:**  $F < 0 \Rightarrow$  failure. Then the condition for  $x_j$  to be bounds consistent (BC) is:

$$x_j \leq \frac{F}{a_j} + \underline{x}_j$$

Thus the FA is triggered by  $\min(x_j)$  adjustments, which decrease  $F$ , which force  $\max(x_j)$  adjustments.

**Note:** if  $\bar{x}_j \leq \frac{F}{a_j} + \underline{x}_j$ , i.e. if  $l_j \leq F$ , then  $x_j$  is already BC. Thus if  $F \geq \max_j l_j$ , then  $C$  is BC.

# Example: scalar products

If  $l_j \leq F$ , then  $x_j$  is BC.

To check this **incrementally**:

When  $\min(x_i)$  is updated:

- ▶ Update  $F$  (requires private state to cache  $\min(x_i)$ ).
- ▶ Update  $l_i$ .
- ▶ Update  $\max_j l_j$  (e.g. maintain a heap of  $l_j$ ).
- ▶ If  $F < \max_j l_j$ , then schedule the full FA.

**Note:** the heap of  $l_j$  comes in very handy for the FA—it pinpoints which  $x_j$  need updating.

# Distinguishing relevant parameters

## Setting

A global constraint is posted over some “objects” (tasks, rectangles, line segments, integers, etc.), which constrain each other.

## Key idea

Partition objects into:

**SOURCE** Objects that are ground and checked.

**TARGET** Objects that are nonground or not yet checked.

All objects are initially in TARGET. Motivation:

- ▶ SOURCE objects are known to be consistent.
- ▶ FA complexity is typically in  $|\text{TARGET}|$ .
- ▶ No need to import/export domain variables for SOURCE objects.
- ▶ TARGET is  $\emptyset \Rightarrow$  entailment.

# Distinguishing relevant parameters

## Implementation

- ▶ Maintain in the private state an array of the form [TARGET | SOURCE], as well as the T/S boundary.
- ▶ Before FA exits, repartition (à la Quicksort) the TARGET part. The order of TARGET doesn't matter!

## Repairing the state after failure

1. Ensure that the T/S boundary is trailed. Then the array will still be valid after failure. Or,
2. Assume that all objects are in TARGET. Or,
3. Call SOURCE objects that will remain so in the rest of the search tree COMMITTED. No need to trail the S/C boundary:  
[TARGET | SOURCE | COMMITTED]

# Distinguishing relevant parameters

Example: “naive” all\_different

A non-fixpoint (non-idempotent) FA in SICStus.

1. Partition TARGET into nonground  $T_1$  and ground  $T_2$ .
2. Sort  $T_2$  (can be avoided?).
3. If  $T_2$  contains duplicates, then fail.
4. Remove all values in  $T_2$  from all domains in  $T_1$ .
5. Move  $T_2$  from TARGET into SOURCE (i.e. adjust the boundary).

# Distinguishing relevant parameters

Example: arc-consistent `all_different`

## Partitioning refinement

[KERNEL | TARGET | SOURCE]

where  $\forall x \in \text{KERNEL} : |\text{dom}(X)| < |\text{KERNEL}|$ .

- ▶ Only KERNEL needs to be involved in the matching and SCC computations. **Exercise: why?**
- ▶ All values in all SCC:s found are removed from all domains in TARGET, as well as from relevant domains in KERNEL.
- ▶ KERNEL is **not** maintained incrementally—as domains shrink, KERNEL grows.
- ▶ T/S boundary is maintained as usual.

# Distinguishing relevant parameters

Example: disjoint line segments, or tasks on a unary resource, or non-overlapping rectangles

## Partitioning refinement

[TARGET | SOURCE | DONE]

- ▶ Objects in DONE are ground and outside the bounding box formed by TARGET, and so cannot affect anything.
- ▶ Objects in TARGET are ground but intersect the bounding box formed by TARGET, and so could prune TARGET.

# Distinguishing relevant parameters

Example: disjoint line segments, or tasks on a unary resource, or non-overlapping rectangles

## Partitioning refinement

[TARGET | SOURCE | DONE]

## Issues

- ▶ The T/S and S/D boundaries are trailed.
- ▶ Before FA exits, TARGET is repartitioned as usual.
- ▶ SOURCE is NOT repartitioned. **Exercise: why?**  
Instead, S/D boundary is placed after the last object that intersects the bounding box.
- ▶ Status bits reflect the exact status (SOURCE vs. DONE).

# Caching support

A.k.a. witnesses

## Context

- ▶ Lots of simple constraints  $\Gamma$  involving two given variables  $x, y$ .
- ▶ We want to make  $x, y$  BC wrt.  $\Gamma$ .
- ▶ Assume that checking  $\Gamma(v, w)$  is **a lot cheaper** than adjusting the bounds of  $x, y$ .
- ▶ Think about disjoint2.

## Implementation

- ▶ For  $\min(x)$ , cache a support  $y = y_1$ .
- ▶ For  $\max(x)$ , cache a support  $y = y_2$ .
- ▶ For  $\min(y)$ , cache a support  $x = x_1$ .
- ▶ For  $\max(y)$ , cache a support  $x = x_2$ .
- ▶ Main consideration: when to invalidate the cache.

**Exercise.**

## Context

- ▶ Some global constraint  $\Gamma$  over a set of “objects”  $\Omega$ .
- ▶ We want to approximate bounds consistency.
- ▶ Assume that we have a relation  $\succeq$  such that:  
*If  $o_1$  is BC wrt.  $\Gamma$ , and  $o_1 \succeq o_2$ , then  $o_2$  is also BC wrt.  $\Gamma$ .*
- ▶ Assume that checking  $o_1 \succeq o_2$  is **a lot cheaper** than making  $o_2$  BC.
- ▶ Think about disjoint2.

## Implementation

- ▶ We make one object BC at a time.
- ▶ During this procedure, we maintain a cache of a BC object  $o'$  that maximizes the likelihood of  $o' \succeq o''$ .
- ▶ We process the objects in “most dominating first” order.