CLFD

A Finite Domain Constraint Solver in Common Lisp

Stephan Frank

20.06.2005

Outline

Outline

What is it?

Given a set of relations on variables:

Is there a variable configuration such that the relation formulas hold? (and find these variable values)

	S	E	N	D
+	M	O	R	E
M	O	N	E	Y

$(\neq S\ E\ N\ D\ M\ O\ R\ Y) \quad \wedge$
$S, E, N, D, M, O, R, Y \in \{0, \ldots, 9\} \quad \wedge$
$M > 0$

What is it?

Given a set of relations on variables:

Is there a variable configuration such that the relation formulas hold? (and find these variable values)

$$
\begin{array}{ccccc}
 & \mathrm{S} & \mathrm{E} & \mathrm{N} & \mathrm{D} \\
+ & \mathrm{M} & \mathrm{O} & \mathrm{R} & \mathrm{E} \\
\hline
\mathrm{M} & \mathrm{O} & \mathrm{N} & \mathrm{E} & \mathrm{Y}
\end{array}
$$

$(\neq S\ E\ N\ D\ M\ O\ R\ Y) \quad \wedge$
$S, E, N, D, M, O, R, Y \in \{0, \ldots, 9\} \quad \wedge$
$M > 0$

What is it?

Given a set of relations on variables:

Is there a variable configuration such that the relation formulas hold? (and find these variable values)

$$
\begin{array}{ccccc}
 & 9 & 5 & 6 & 7 \\
+ & 1 & 0 & 8 & 5 \\
\hline
1 & 0 & 6 & 5 & 2
\end{array}
$$

$$(\neq S\ E\ N\ D\ M\ O\ R\ Y) \quad \wedge$$
$$S, E, N, D, M, O, R, Y \in \{0, \ldots, 9\} \quad \wedge$$
$$M > 0$$

Constraint Domains/Solvers

To ensure efficient solving constraint solvers operate on different constraint domains. Examples are:

- linear equations over reals (simplex method)
- interval arithmetic over reals
- set constraints
- finite domain constraints

Existing Solvers

- **ILOG** commercial solvers (C++/Java) with a wide range of domains
- **Koalog** commercial competition (Java)
- **Eclipse/Sicstus** Prolog systems with a wide range of solvers for different domains (non-commercial offerings available)
- **Figaro/firstCS** finite domain solvers in C++/Java (not available)
- **Facile** fast and powerful solver in Ocaml (free)
- **Gecode** finite domain solver (?) in C++ (not yet released)
- **Screamer** interval arithmetic solver in CL

Why CLFD?

Problems with current offerings:

- Available modern solvers are either
  - commercial offerings, and/or
  - not easy to interface from Common Lisp
- Screamer's design decisions (which are well justified for the underlying interval arithmetic) make it hard to integrate current (finite domain) pruning techniques
- needed a test-bed for pruning algorithm and search heuristics

Constraint Satisfaction Problems

CPSs consist of:

- a finite number of variables $X_1, \ldots, X_n$,
- each with Domain D_i, as finite set of enumerable values
- a constraint C on variables $X_i, \ldots, X_j$ describes a subset of $D_i \times \cdots \times D_j$
- a CSP is a finite set of variables $\mathcal{X}$ with a finite set of constraints $\mathcal{C}$, each on a subset of $\mathcal{X}$

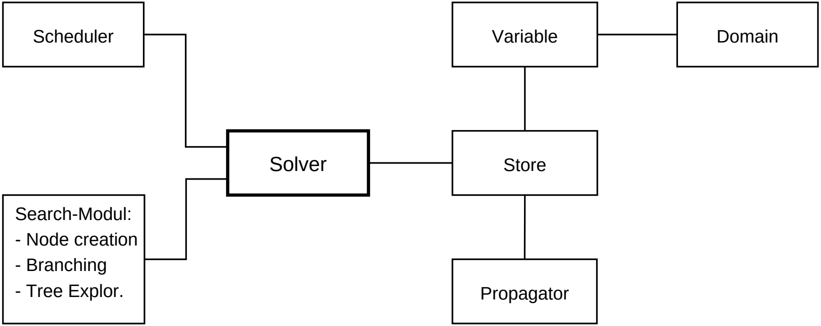
A CSP is *solved* if each each Domain consists of a single value only, or *inconsistent* if at least one domain is empty.

Outline

CLFD: Foundations and Goals

- finite domain constraints (for now)
- roughly based on Figaro design
- CLOS based
- state of the art pruning algorithms
- easy replacement of solver parts for experimentation
- work in progress

Module Structure



Domain Representation I

Requirements

- encodes domain state (set of allowed integers)
- space efficient
- fast operations for domain reduction
- fast access to single elements (for inclusion check)

Operations:

- interval subjoin
- bound restriction ($\leq$, $\geq$)
- difference, intersection
- element inclusion, element count (cardinality)

Domain Representation II

Currently three alternative representations:

- Common Lisp bit-vector
- integer encoded bit-vector
- diet splay tree (*Discrete Interval Encoding Tree*)

Domain Representation II

Currently three alternative representations:

- Common Lisp bit-vector
- integer encoded bit-vector
- diet splay tree (*Discrete Interval Encoding Tree*)

$$\{1, 3\text{–}5, 9\} \Rightarrow \langle \underbrace{0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, \ldots, 0}_{\text{constant size}} \rangle$$

Domain Representation II

Currently three alternative representations:

- Common Lisp bit-vector
- integer encoded bit-vector
- diet splay tree (*Discrete Interval Encoding Tree*)

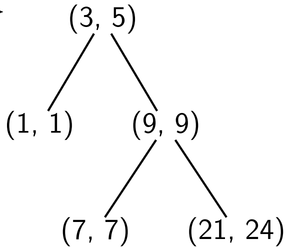
$$\{1, 3\text{–}5, 9\} \Rightarrow 570 \Rightarrow 0\mathrm{x}b\underbrace{1000111010}_{\text{dynamic size}}$$

Domain Representation II

Currently three alternative representations:

- Common Lisp bit-vector
- integer encoded bit-vector
- diet splay tree (*Discrete Interval Encoding Tree*)

$\{1, 3–5, 7, 9, 21–24\} \Rightarrow$



Variables

The `variable` class is responsible for

- encapsulating the variable domain
- recording the propagators (i.e. constraints) the variable participates in
- variables can be *aliased* when direct equality ($x = y$) is inferred

Propagators

- represent the actual constraints
- each propagator must provide a `propagate-constraint` method that is responsible for pruning the variable domain values.
- affected variables are recorded in each propagator instance
- *stateful* and *stateless* variants
- example propagators are:
  - $x \neq y + c$
  - $x + y = z$
  - $x \cdot y = z$
  - $x = y^n$
  - linear equations
  - `all-different`

The Store

Encapsulates the overall constraint system state:

- variables and their domains
- propagator instances

The store must be able to backup the current state for non-deterministic search (later).

Outline

Scheduler

- records the pending propagators
- chooses the next propagator to run until fix-point is reached

simple scheduler:

```
(defmethod run-propagation ((scheduler basic-scheduler) store)
  (let ((queue (scheduler-queue scheduler)))
    (loop for propagator = (dequeue queue)
          while propagator do
            (unless (propagate-and-schedule store scheduler propagator)
              (return-from run-propagation nil))
          finally (return t))))
```

Scheduler

- records the pending propagators
- chooses the next propagator to run until fix-point is reached

simple scheduler:

```
(defmethod run-propagation ((scheduler basic-scheduler) store)
  (let ((queue (scheduler-queue scheduler)))
    (loop for propagator = (dequeue queue)
          while propagator do
            (unless (propagate-and-schedule store scheduler propagator)
              (return-from run-propagation nil))
          finally (return t))))
```

Scheduler Variations

Propagation order is one vital point for overall solver performance

- prioritise depending on complexity
- prioritise depending on scheduling order
- dynamically re-prioritise [SS04]

Scheduler Variations

Propagation order is one vital point for overall solver performance

- prioritise depending on complexity
- prioritise depending on scheduling order
- dynamically re-prioritise [SS04]

Outline

Search Elements

Currently consists of three elements (after Figaro):

Nodes	choice points during search (system state must be restorable)
Branch Heuristics	how is the next search tree level produced?
Exploration Strategy	how is the search tree explored (DFS, BFS)?

Problems:

- unnecessarily complex yet inflexible
- no easy combination of search goals

Functional style interface using higher order function under construction.

Search Elements

Currently consists of three elements (after Figaro):

Nodes choice points during search (system state must be restorable)

Branch Heuristics how is the next search tree level produced?

Exploration Strategy how is the search tree explored (DFS, BFS)?

Problems:

- unnecessarily complex yet inflexible
- no easy combination of search goals

Functional style interface using higher order function under construction.

Search Elements

Currently consists of three elements (after Figaro):

Nodes	choice points during search (system state must be restorable)
Branch Heuristics	how is the next search tree level produced?
Exploration Strategy	how is the search tree explored (DFS, BFS)?

Problems:

- unnecessarily complex yet inflexible
- no easy combination of search goals

Functional style interface using higher order function under construction.

Outline

Problem Definition

User macros to enable simplified problem definition:

```
(define-constraint-system *pyth*
    (:store copying-store
     :domain integer-finite-domain
     :scheduler basic-scheduler)
  ((in a (1 . 22))
   (in b (1 . 22))
   (in c (1 . 22))
   (= (+ (^ a 2) (^ b 2))
      (^ c 2))))
```

```
(search-tree (make-instance 'dfs-exploration)
             (make-instance 'copying-node :store *pyth*)
             (make-instance 'fail-first-branching)
             :find-all)
```

Problem Definition

User macros to enable simplified problem definition:

```
(define-constraint-system *pyth*
    (:store copying-store
     :domain integer-finite-domain
     :scheduler basic-scheduler)
  ((in a (1 . 22))
   (in b (1 . 22))
   (in c (1 . 22))
   (= (+ (^ a 2) (^ b 2))
      (^ c 2))))

(search-tree (make-instance 'dfs-exploration)
             (make-instance 'copying-node :store *pyth*)
             (make-instance 'fail-first-branching)
             :find-all)
```

Some Results

Simple speed comparison

	S-M-M[1]	8-Queens	Pythagorean Triples (range $\{1,...,22\}$)
Screamer	2.78	0.09	0.12
CLFD	0.05	0.23 – 0.5	0.77

- search currently too slow
- mostly due to too much gf dispatch and unecessary consing
- redesign of search infrastructure next that currently simply resembles the one of Figaro

[1] find all 25 solution, leading zeros allowed

Outline

Conclusion & Future Work

CLFD:

- work in progress finite domain solver
- modular architecture
- generic function protocol for simple module replacement
- propagator implementations easily extensible

Future:

- more propagators (global constraints)
- better search interface
- profiling

Conclusion & Future Work

CLFD:

- work in progress finite domain solver
- modular architecture
- generic function protocol for simple module replacement
- propagator implementations easily extensible

Future:

- more propagators (global constraints)
- better search interface
- profiling

Thank You

Christian Schulte and Peter J. Stuckey.
Speeding Up Constraint Propagation.
In Mark Wallace, editor, *Tenth International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, pages 619–633, Toronto, Canada, September 2004. Springer-Verlag.