Search by Constraint Propagation

Thierry Martinez    François Fages    Sylvain Soliman

Inria Paris-Rocquencourt, France

Thierry.Martinez@inria.fr

PPDP ’15, July 14–16, 2015, Siena, Italy
Search Procedures for Constraint Programming

Constraint programming

\[ \text{Constraint model} + \text{Search procedure} \]

- relational
- high level
- MiniZinc

- hardly declarative
- very dependent to the solver
- low-level languages

Search procedures are crucial to solve hard combinatorial (typically, NP-complete) problems.
Rectangle-packing Problem (or Korf’s Problem)

Given:

- a set of rectangles
- a specific enclosing rectangle

Question: can all the given squares fit within the boundaries of the enclosing rectangle without any overlap?

NP-complete (by reduction from bin-packing).

Rectangle-packing Problem (or Korf’s Problem)

Given:
- a set of rectangles
- a specific enclosing rectangle

Question: can all the given squares fit within the boundaries of the enclosing rectangle without any overlap?

NP-complete (by reduction from bin-packing).

A CP Strategy for the Rectangle-Packing Problem


Variables: $(x_i, y_i)$ for each rectangle $i$ to pack, ordered by increasing size.

Strategy:

1. interval splitting on $x_n, x_{n-1}, \ldots, x_1$,
   \[ x_i: \]

2. dichotomy on $x_n, x_{n-1}, \ldots, x_1$,
   \[ x_i: \]

3. interval splitting on $y_n, y_{n-1}, \ldots, y_1$,

4. dichotomy on $y_n, y_{n-1}, \ldots, y_1$.

This strategy was implemented in Sicstus Prolog.
Search Procedures are closely related to modelling choices (constraints)

H. Simonis and B. O’Sullivan’s model for the Rectangle-Packing Problem:

- A 2D-disjoint constraint between rectangles.

\[ \bigcap_{i} [x_i, x_i + w_i] \times [y_i, y_i + h_i] = \emptyset \]

- A cumulative constraint that ensures that for every abscissa \( x \), all rectangles can fit in the enclosing height \( H \).

\[ \forall x, \sum_{i \mid x_i \leq x < x_i + w_i} h_i < H \]

- A cumulative constraint that ensures that for every ordinate \( y \), all rectangles can fit in the enclosing width \( W \).

\[ \forall y, \sum_{i \mid y_i \leq y < y_i + h_i} w_i < W \]
The Clp2Zinc Theorem

Constraint model $\mathcal{M}$ \quad Constraint model $\mathcal{M} \cup \mathcal{M}_t$

Tree search procedure $t$ \quad $\sim$ \quad Basic labeling $t'$

computationally similar

Reified constraint: $X = 1 \Leftrightarrow c$ is true.

Contributions:

- A high-level language for tree search procedure (ClpZinc),
- The resulting model can be solved by any solver.
Arithmetic constraints for Interval Splitting and Dichotomic Search

Compiling And/Or-Trees into Reified Contraints

Search Transformers via Meta-interpretation

Beyond And-Or Trees

Conclusion
Arithmetic constraint for Interval Splitting

For a fixed step $s \geq 1$ and for $x \in [0, n[$.

$x$: 

\[
\begin{array}{c}
\xrightarrow{\text{domain filtering and constraint propagation of the Euclidean division equation}} \\
\end{array}
\]

\[
x = s \times q + r
\]

where $r \in [0, s[$.
Interval Splitting: The Search Tree

\[ x \in [0, n[ \]

\[ q = 0 \quad x \in [0, s[ \]

\[ q = 1 \quad x \in [s, 2 \cdot s[ \]

\[ q = \lceil \frac{n}{s} \rceil - 1 \quad x \in [\lceil \frac{n}{s} \rceil \cdot s, n[ \]
Arithmetic constraint for Dichotomic Search

For \( x \in [0, 2^d[. \)

\[ x = \sum_{0 \leq k < d} x_k 2^k \]

Obtained by **domain filtering** and **constraint propagation** of the **binary decomposition** of \( x \).

with \( x_k \in \{0, 1\} \).
Dichotomic Search: The Search Tree

\[ x \in [0, 2^d] \]

\[ x_{d-1} = 0 \]

\[ x_{d-1} = 1 \]

\[ x \in [0, 2^{d-1}] \]

\[ x_{d-2} = 0 \]

\[ x_{d-2} = 1 \]

\[ x \in [0, 2^{d-2}] \]

\[ x \in [2^{d-2}, 2^{d-1}] \]

\[ x \in [2^{d-1}, 2^{d-1} + 2^{d-2}] \]

\[ x \in [2^{d-1} + 2^{d-2}, 2^d] \]
Choice through labeling

The following ClpZinc program (CLP):

```
var 1..10: x;
:- (x <= 5 ; x >= 6).
```

can be reified into the following MiniZinc program (CSP):

```
var 1..10: x;
var 0..1: X1;
constraint X1 = 0 -> x <= 5;
constraint X1 = 1 -> x >= 6;
solve :: seq_search([ int_search([X1], input_order, indomain_min, complete) ])
satisfy;
```
...and even better, if we detect that constraints are opposite. The following ClpZinc program (CLP):

```clp
var 1..10: x;
:- (x <= 5 ; x > 5).
```

can be reified into the following MiniZinc program (CSP):

```mzn
var 1..10: x;
var 0..1: X1;
constraint X1 = 0 <-> x <= 5;
solve :: seq_search([int_search([X1], input_order, indomain_min, complete)]) satisfy;
```
Multiple choices

The following ClpZinc program (CLP):

\[
\text{var } 1..10: \ x; \\
:\text{- } (x \leq 3 \ ; \ x \geq 4, \ x \leq 7 \ ; \ x \geq 8).
\]

can be reified into the following MiniZinc program (CSP):

\[
\text{var } 1..10: \ x; \\
\text{var } 0..2: \ \text{X1}; \\
\text{constraint } \text{X1} = 0 \rightarrow x \leq 3; \\
\text{constraint } \text{X1} = 1 \rightarrow x \geq 4; \\
\text{constraint } \text{X1} = 1 \rightarrow x \leq 7; \\
\text{constraint } \text{X1} = 2 \rightarrow x \geq 8; \\
\text{solve :: seq_search([} \\
\text{\qquad \text{int_search([X1], input_order, indomain_min, complete) }]\text{)} satisfy;}
\]
Nested choices

The following ClpZinc program (CLP):

\[
\text{var } 1..10: x; \\
\text{:- (} x \leq 3 \text{ ; } x \geq 4, (x \leq 7 \text{ ; } x \geq 8)).
\]

can be reified into the following MiniZinc program (CSP):

\[
\text{var } 1..10: x; \\
\text{var } 0..1: X2; \; \text{var } 0..1: X1; \\
\text{constraint } X1 = 0 \rightarrow x \leq 3; \\
\text{constraint } X1 = 1 \rightarrow x \geq 4; \\
\text{constraint } X1 = 1 \land X2 = 0 \rightarrow x \leq 7; \\
\text{constraint } X1 = 1 \land X2 = 1 \rightarrow x \geq 8; \\
\text{constraint } X1 = 0 \rightarrow X2 = 0; \\
solve :: \text{seq_search}([ \\
\quad \text{int_search}([X1], input\_order, indomain\_min, complete), \\
\quad \text{int_search}([X2], input\_order, indomain\_min, complete) \\
\]) \text{ satisfy};
\]
From And-Or Trees to Reified Constraints

- Each or-node is mapped to a variable.
- And-nodes are reflected in the variable-ordering in the labeling.
- We should take care about variables in other branches in order to reduce unnecessary labeling choices.
CLP as modelling language

1. Constraints as predicates
   \[ \text{non} \_\text{overlap}([O_1, \ldots, O_n]) \]

2. Sequence
   \[ G_1, G_2 \]

3. Choice-points
   \[ G_1; G_2 \]

4. General form of recursion: predicate definition
   (additional conditions to ensure termination)
Zinc as target language

```zinc
var 0..10: x;
var 0..1: _x1;
var 0..1: _x2;
solve :: seq_search([
    int_search([_x1], input_order, interval(0, 1), complete),
    int_search([_x2], input_order, interval(0, 1), complete)
])
satisfy;
```
ClpZinc

A Modeling Language for Constraints and Search.

- **Modeling** search independently from the underlying constraint solver through tree search procedures with state variables.
- Extending MiniZinc with **Horn clauses with constraints** (Prolog-like search description language).

Available compiler targeting most common solvers:

http://lifeware.inria.fr/~tmartine/clp2zinc

A compiler from CLP($\mathcal{H} + \mathcal{X}$) to CSP($\mathcal{X}'$).

- $\mathcal{H}$: domain of Herbrand terms,
- $\mathcal{X}'$: domain of the underlying constraint system.

Depth-first, left-to-right.
“Angelic” transformation.
Dichotomic Search: The Code

\[
dichotomy(X, \text{Min}, \text{Max}) :-
\quad \text{dichotomy}(X, \text{ceil}(\log(2, \text{Max} - \text{Min} + 1))).
\]

\[
dichotomy(X, \text{Depth}) :-
\quad \text{Depth} > 0,
\quad \text{Middle} = (\text{min}(X) + \text{max}(X)) \div 2,
\quad (X \leq \text{Middle}; X > \text{Middle}),
\quad \text{dichotomy}(X, \text{Depth} - 1).
\]

\[
dichotomy(X, 0).
\]

\[
\text{var 0..5: x;}
\quad :- \text{dichotomy(x, 0, 5)}.
\]
interval_splitting(X, Step, Min, Max) :-
    Min + Step <= Max, NextX = min(X) + Step,
    (X < NextX ; X >= NextX,
        interval_splitting(X, Step, Min + Step, Max)
    )
).

interval_splitting(X, Step, Min, Max) :-
    Min + Step > Max.
var 0..5: x;
:- interval_splitting(x, 2, 0, 5).
From $\text{CLP}(\mathcal{H} + \mathcal{X})$ to and/or-trees over $\mathcal{X}$

Translation function with environment $\llbracket \cdot \rrbracket_s$ to trees with holes $\square_s$.

$\llbracket \text{true} \rrbracket_s \rightarrow \square_s$

$\llbracket \text{false} \rrbracket_s \rightarrow \bot$

$\llbracket X = v \rrbracket_s \rightarrow \square_s \wedge (X = v)$

$\llbracket c \rrbracket_s \rightarrow$

\begin{center}
$\wedge$
\end{center}

\begin{center}
fresh \quad \square_s
\end{center}

where $c$ is a constraint or a search annotation
Translation for sequences

\[ [A, B]_s \rightarrow [A]_s \]

\[ [B]_{s_1} \quad \quad [B]_{s_n} \]

all \( \square_{s_i} \) of \([A]_s\) are filled with \([B]_{s_i}\)

i.e., \([A]_s[\forall i, [B]_{s_i}/\square_{s_i}]\)
Translation for choices

- if $A$ or $B$ changes the store, i.e., $\exists s' \neq s$, $\square_{s'} \in [A]_s$ or $[B]_s$:

  $$[A;B]_s \longrightarrow \lor$$

  $$[A]_s \quad [B]_s$$

- if neither $A$ nor $B$ changes the store:

  $$[A;B]_s \longrightarrow \land$$

  $$\lor$$

  $$\lor$$

  $$\land$$

  $$[A]_s \quad [B]_s$$

  $$\square_{s} \quad \square_{s}$$

  the leftmost leaf is $[A]_s[\top/\square_{s}]$ and its sibling $[B]_s[\top/\square_{s}]$
A choice that changes the $\mathcal{H}$ store

The following ClpZinc program (CLP):

\begin{verbatim}
var 1..10: x;
var 1..10: y;
:- (A = x ; A = y), A <= 5.
\end{verbatim}

can be reified into the following MiniZinc program (CSP):

\begin{verbatim}
var 1..10: x;
var 1..10: y;
var 0..1: X1;
constraint X1 = 0 -> x <= 5;
constraint X1 = 1 -> y <= 5;
solve :: seq_search([ int_search([X1], input_order, indomain_min, complete) ])
satisfy;
\end{verbatim}
A choice that does not change the $\mathcal{H}$ store

The following ClpZinc program (CLP):

\begin{verbatim}
var 1..10: x;
var 1..10: y;
:- (x = 1 ; y = 1), x <= y.
\end{verbatim}

can be reified into the following MiniZinc program (CSP):

\begin{verbatim}
var 1..10: x;
var 1..10: y;
var 0..1: X1;
constraint X1 = 0 -> x = 1;
constraint X1 = 1 -> y = 1;
constraint x <= y;
solve :: seq_search([ int_search([X1], input_order, indomain_min, complete) ])
satisfy;
\end{verbatim}
Indexicals

\[
\text{int\_search}([\_x1], \text{input\_order}, \text{min}(x), \text{complete})], \\
\text{int\_search}([\_x2], \text{input\_order}, \text{max}(x), \text{complete})
\]

And-or trees for dichotomic search with indexicals

\[
\land \\
\text{indexical\_min}(X1, x) \\
\text{indexical\_max}(X2, x) \\
\lor \\
x \leq (X1 + X2) \div 2 \\
x > (X1 + X2) \div 2 \\
\land \\
\text{indexical\_min}(X3, x) \\
\text{indexical\_max}(X4, x) \\
\lor \\
x \leq (X3 + X4) \div 2 \\
x > (X3 + X4) \div 2 \\
\land \\
\text{indexical\_min}(X5, x) \\
\text{indexical\_max}(X6, x) \\
\lor \\
x \leq (X5 + X6) \div 2
\]
Search Transformers via Meta-interpretation

Meta-interpretation

- Limited discrepancy search (LDS)
- Symmetry breaking during search (SBDS)

Symmetry breaking during search in constraint programming

\[
\text{sbds}(\text{top}, \_). \\
\text{sbds}(\text{or}(A, B), \text{Path}) :- \\
\quad ( \ A = \text{constraint}(C, A0), \\
\quad ( \ C, \text{sbds}(A, [C \mid \text{Path}]) \\
\quad ; \ \text{cut_symmetry}(C, \text{Path}), \text{sbds}(B, \text{Path})) \\
\quad ; \ A \neq \text{constraint}(\_, \_), \\
\quad (\text{sbds}(A, \text{Path}) ; \text{sbds}(B, \text{Path}))). \\
\text{sbds}(\text{constraint}(C, T), \text{Path}) :- C, \text{sbds}(T, [C \mid \text{Path}]). \\
\text{search_tree}(\text{labeling_list}(\text{queens}, 1, n), T), \\
\text{sbds}(T, []). \\
\]
Exponential speed-up with LDS

\[
\begin{align*}
\text{var } & 0..1: x; \\
\text{var } & 0..1: y; \\
\text{array}[0..n] & \text{ of var } 0..1: a;
\end{align*}
\]

\[
:\text{- int_search}(a, \text{input\_order}, \text{indomain\_min}, \text{complete}), \\
\text{lds}(((x = 0; x = 1), (y = 0; y = 1)), 0), x \neq y.
\]

\[
\text{a: } 2^n \text{ nodes to explore}
\]
Beyond And-Or Trees

State variables, persistent through backtracking.

```plaintext
annotation store(var bool: c, string: id,
                  array[int] of var int: src);
annotation retrieve(string: id,
                    array[int] of var int: target);
```

For optimization procedure, e.g. branch-and-bound.
Branch-and-bound

maximize(G, S, Min, Max) :-
   domain(I, Min, Max + 1), domain(Best, Min, Max),
   domain(Fail, 0, 1),
   domain(A, 0, 1), domain(B, 0, 1), domain(C, 0, 1),
   (Fail = 0 -> A != B \ B != C \ A != C),
   store("bb_best", [Min, 0]),
   labeling(I, Min, Max + 1),
   retrieve("bb_best", [Best, Fail]),
   ( Fail = 0, store("bb_best", [Best, 1]),
     S > Best, G, store("bb_best", [S, 0]),
     labeling(A, 0, 1), labeling(B, 0, 1)
   ; Fail = 1, I = Max + 1, S = Best, G).

minimize(G, S, Min, Max) :-
   domain(Dual, Min, Max), Dual = Max - S + Min,
   maximize(G, Dual, Min, Max).
Conclusion and perspectives

- Tree search procedures can be embedded in the constraint model.
  ⇒ solver-independent high-level search specification/modelling.

- Constraint Logic Programming programs can be compiled into Constraint Solving Problems!
  ⇒ ClpZinc: solver-independent modelling language for constraints and search.

- Constraint solver implementations can focus only on the most simple labeling search strategy.

- Opens the implementation of novel search procedures based on constraint propagation.

- Targetting other kind of solvers: MIP, SAT, local search?

- Lazy clause generation?