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

= 

<table>
<thead>
<tr>
<th>Constraint model</th>
<th>+</th>
<th>Search procedure</th>
</tr>
</thead>
</table>

- 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** \(x_n, x_{n-1}, \ldots, x_1\),
   \[ x_i: \]
   \[ \leftarrow \bullet \bullet \bullet \rightarrow \]

2. **dichotomy on** \(x_n, x_{n-1}, \ldots, x_1\),
   \[ x_i: \]
   \[ \leftarrow \bullet \bullet \bullet \rightarrow \]
   \[ \leftarrow \bullet \bullet \rightarrow \bullet \bullet \bullet \rightarrow \]

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 \( H \).

\[ \forall x, \sum_i h_i < H \quad \text{where } x_i \leq x < x_i + w_i \]

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

\[ \forall y, \sum_i w_i < W \quad \text{where } y_i \leq y < y_i + h_i \]
The Clp2Zinc Theorem

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 Constraints

Search Transformers via Meta-interpretation

Beyond And-Or Trees

Conclusion
Arithmetic constraint for Interval Splitting

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

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

where $r \in [0, s]$. Obtained by \textit{domain filtering} and \textit{constraint propagation} of the \textbf{Euclidean division} equation.
Interval Splitting: The Search Tree

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

- \( q = 0 \):
  - \( x \in [0, s] \)
- \( q = 1 \):
  - \( x \in [s, 2 \cdot s] \)
- \( q = \lceil \frac{n}{s} \rceil - 1 \):
  - \( x \in [(\lceil \frac{n}{s} \rceil - 1) \cdot s, n] \)
Arithmetic constraint for Dichotomic Search

For $x \in [0, 2^d[.\$

$x_i$: 

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

with $x_k \in \{0, 1\}$.

Obtained by domain filtering and constraint propagation of the binary decomposition of $x$. 

![Diagram of binary decomposition]

\[ x_i: \]

\[ x = \sum_{0 \leq k < d} x_k 2^k \]
Dichotomic Search: The Search Tree

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

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

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

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

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

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

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

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

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

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

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

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

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

The following ClpZinc program (CLP):

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

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

```mzn
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;
```
Choice through labeling, cont’d

...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):

```minizinc
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):

```plaintext
var 1..10: x;
:- (x <= 3 ; x >= 4, x <= 7 ; x >= 8).
```

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

```plaintext
var 1..10: x;
var 0..2: X1;
constraint X1 = 0 -> x <= 3;
constraint X1 = 1 -> x >= 4;
constraint X1 = 1 -> x <= 7;
constraint X1 = 2 -> x >= 8;
solve :: seq_search([int_search([X1], input_order, indomain_min, complete)]) satisfy;
```
Nested choices

The following ClpZinc program (CLP):

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

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

\[
\begin{align*}
\text{var } 1..10: \ x; \\
\text{var } 0..1: \ X2; \quad \text{var } 0..1: \ X1; \\
\text{constraint } X1 = 0 \rightarrow x \leq 3; \\
\text{constraint } X1 = 1 \rightarrow x \geq 4; \\
\text{constraint } X1 = 1 \land X2 = 0 \rightarrow x \leq 7; \\
\text{constraint } X1 = 1 \land X2 = 1 \rightarrow x \geq 8; \\
\text{constraint } X1 = 0 \rightarrow X2 = 0; \\
\text{solve :: seq_search([}
\hspace{1em} \text{int_search}([X1], \text{input_order, indomain_min, complete}), \\
\hspace{1em} \text{int_search}([X2], \text{input_order, indomain_min, complete})
\hspace{1em}]) \text{ satisfy};
\end{align*}
\]
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

   \[ \text{non\_overlap}([O_1, \ldots, O_n]) \]

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, Min, Max) :-
    dichotomy(X, ceil(log(2, Max - Min + 1))).
dichotomy(X, Depth) :-
    Depth > 0,
    Middle = (min(X) + max(X)) div 2,
    (X <= Middle ; X > Middle),
    dichotomy(X, Depth - 1).
dichotomy(X, 0).
var 0..5: x;
:- dichotomy(x, 0, 5).
```
Interval Splitting: The Code

\[
\text{interval\_splitting}(X, \text{Step}, \text{Min}, \text{Max}) :- \\
\quad \text{Min} + \text{Step} \leq \text{Max}, \text{NextX} = \text{min}(X) + \text{Step}, \\
\quad ( \quad \begin{align*} 
\quad & X < \text{NextX} \\
\quad & ; \\
\quad & X \geq \text{NextX}, \\
\quad & \quad \text{interval\_splitting}(X, \text{Step}, \text{Min} + \text{Step}, \text{Max}) 
\quad ) . \\
\text{interval\_splitting}(X, \text{Step}, \text{Min}, \text{Max}) :- \\
\quad \text{Min} + \text{Step} > \text{Max}. \\
\text{var \ 0..5: x} ; \\
\quad :- \text{interval\_splitting}(x, 2, 0, 5). 
\]
From CLP(\(\mathcal{H} + \mathcal{X}\)) to and/or-trees over \(\mathcal{X}\)

Translation function with environment \([\cdot]_s\) to trees with holes \(\square_s\).

\[
\begin{align*}
[\text{true}]_s & \rightarrow \hspace{1cm} \square_s \\
[\text{false}]_s & \rightarrow \hspace{1cm} \bot \\
[X = v]_s & \rightarrow \hspace{1cm} \square_s \land (X = v) \\
[c]_s & \rightarrow \hspace{1cm} \begin{array}{c} \land \\
\hspace{1cm} c \\
\hspace{1cm} \square_s \end{array}
\end{align*}
\]

where \(c\) is a constraint or a search annotation.
Translation for sequences

$[A, B]_s \rightarrow \begin{array}{c}
\begin{array}{c}
\Box_{s_1} \\
\vdots \\
[B]_{s_1}
\end{array} \\
\begin{array}{c}
\Box_{s_n} \\
\vdots \\
[B]_{s_n}
\end{array}
\end{array}

\text{all } \Box_{s_i} \text{ of } [A]_s \text{ are filled with } [B]_{s_i}

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

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

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

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

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

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

  $\lor$

  $\square s$

  $[A]_s \ [B]_s$

  $\land$

  $\square s \ [B]_s$

  $\land$

  $\square s \ [A]_s$

  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):

```clp
var 1..10: x;
var 1..10: y;
:- (A = x ; A = y), A <= 5.
```

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

```mini
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;
```
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}
int_search([x1], input_order, min(x), complete)
int_search([x2], input_order, max(x), complete)

And-or trees for dichotomic search with indexicals

indexical_max(X6, x)
indexical_min(X5, x)

indexical_max(X4, x)
indexical_min(X3, x)

indexical_max(X2, x)
indexical_min(X1, x)
Search Transformers via Meta-interpretation

Meta-interpretation

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

Symmetry breaking during search in constraint programming


\[\text{sbds}(\text{top}, \neg).\]

\[\text{sbds(or}(A, B), \text{Path}) := \]

\((A = \text{constraint}(C, A0),\]

\((C, \text{sbds}(A, [C | Path]))\]

\; \text{cut_symmetry}(C, \text{Path}), \text{sbds}(B, \text{Path}))\]

\; A \not= \text{constraint}(\neg, \neg),\]

\((\text{sbds}(A, \text{Path}); \text{sbds}(B, \text{Path})).\]

\[\text{sbds(}\text{constraint}(C, T), \text{Path}) := C, \text{sbds}(T, [C | Path]).\]

\(:= \text{search_tree(labeling_list(queens, 1, n), T),}\]

\[\text{sbds}(T, []).\]
Exponential speed-up with LDS

\[
\text{var 0..1: } x; \\
\text{var 0..1: } y; \\
\text{array[0..n] of var 0..1: } a; \\
\]

\[\text{:- int_search(a, input_order, indomain_min, complete)}, \]
\[\text{lds(((x = 0; x = 1), (y = 0; y = 1)), 0), x != y.}\]

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

State variables, persistent through backtracking.

\[ \text{annotation store(var @me: c, } \exists j X: \text{id}, \]
\[ \text{array[int] of var } j \exists: \text{src);} \]
\[ \text{annotation retrieve( } \exists j X: \text{id}, \]
\[ \text{array[int] of var } j \exists: \text{target);} \]

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

\texttt{maximize}(G, S, \text{Min}, \text{Max}) :-
\text{domain}(I, \text{Min}, \text{Max} + 1), \text{domain}(\text{Best}, \text{Min}, \text{Max}),
\text{domain}(\text{Fail}, 0, 1),
\text{domain}(A, 0, 1), \text{domain}(B, 0, 1), \text{domain}(C, 0, 1),
(\text{Fail} = 0 \rightarrow A \neq B \land B \neq C \land A \neq C),
\text{store}("bb\_best", [\text{Min}, 0]),
\text{labeling}(I, \text{Min}, \text{Max} + 1),
\text{retrieve}("bb\_best", [\text{Best}, \text{Fail}]),
( \text{Fail} = 0, \text{store}("bb\_best", [\text{Best}, 1]),
\text{S} > \text{Best}, G, \text{store}("bb\_best", [\text{S}, 0]),
\text{labeling}(A, 0, 1), \text{labeling}(B, 0, 1)
; \text{Fail} = 1, I = \text{Max} + 1, S = \text{Best}, G).

\texttt{minimize}(G, S, \text{Min}, \text{Max}) :-
\text{domain}(\text{Dual}, \text{Min}, \text{Max}), \text{Dual} = \text{Max} - S + \text{Min},
\text{maximize}(G, \text{Dual}, \text{Min}, \text{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?