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 \(x_n, x_{n-1}, \ldots, x_1\),

2. dichotomy on \(x_n, x_{n-1}, \ldots, x_1\),

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

Reified constraint: \( X = 1 \Leftrightarrow c \text{ is true.} \)

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 $x \in [0, n]$. 

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

where $r \in [0, s]$. 

Obtained by domain filtering and constraint propagation of the 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 \( x \in [0, 2^d[. \)

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

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

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

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

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

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

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

The following ClpZinc program (CLP):

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

...and even better, if we detect that constraints are opposite. The following ClpZinc program (CLP):

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

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

```mini_zinc
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):

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

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

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

\[
\text{var } 1..10: x; \\
\text{var } 0..1: X2; \text{ var } 0..1: X1; \\
\text{constraint } X1 = 0 -> x \leq 3; \\
\text{constraint } X1 = 1 -> x \geq 4; \\
\text{constraint } X1 = 1 \land X2 = 0 -> x \leq 7; \\
\text{constraint } X1 = 1 \land X2 = 1 -> x \geq 8; \\
\text{constraint } X1 = 0 -> X2 = 0; \\
\text{solve :: seq_search([} \\
\quad \text{int_search([}X1], \text{ input_order, indomain_min, complete),} \\
\quad \text{int_search([}X2], \text{ input_order, indomain_min, complete}) \\
\]) \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
   
   \[ \text{non\_overlap}([O_1, \ldots, O_n]) \]

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 \( \text{CLP}(\mathcal{H} + \mathcal{X}) \) to \( \text{CSP}(\mathcal{X}) \).

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

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

\[
dichotomy(X, \text{Min}, \text{Max}) :-
    \text{dichotomy}(X, \text{ceil}(\log(2, \text{Max} - \text{Min} + 1))).
\]
\[
dichotomy(X, \text{Depth}) :-
    \text{Depth} > 0,
    \text{Middle} = (\text{min}(X) + \text{max}(X)) \div 2,
    (X \leq \text{Middle} ; X > \text{Middle}),
    \text{dichotomy}(X, \text{Depth} - 1).
\]
\[
dichotomy(X, 0).
\]
\[
\text{var 0..5: \text{x};}
\]
\[
\text{:- dichotomy(x, 0, 5).}
\]
Interval Splitting: The Code

```prolog
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 CLP($\mathcal{H} + \mathcal{X}$) to and/or-trees over $\mathcal{X}$

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

\[
\begin{align*}
[true]_s & \quad \longrightarrow \quad \square_s \\
[false]_s & \quad \longrightarrow \quad \bot \\
[X = v]_s & \quad \longrightarrow \quad \square_s (X = v) \\
[c]_s & \quad \longrightarrow \quad \begin{array}{c}
\wedge \\
c \\
\square_s
\end{array}
\end{align*}
\]

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

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

$\Box_{s_1} \rightarrow \uparrow \rightarrow [B]_{s_1}$

$\Box_{s_n} \rightarrow \uparrow \rightarrow [B]_{s_n}$

all $\Box_{s_i}$ of $[A]_s$ 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., $\exists s' \neq s$, $\mathbf{\Box} s' \in [A]_s$ or $[B]_s$:
  
  \[
  [A;B]_s \rightarrow \bigvee
  \]
  
  \[
  [A]_s \quad [B]_s
  \]

- if neither $A$ nor $B$ changes the store:
  
  \[
  [A;B]_s \rightarrow \bigwedge
  \]
  
  \[
  \bigvee
  \]
  
  \[
  \bigwedge
  \]
  
  \[
  [A]_s \quad [B]_s
  \]
  
  \[
  \mathbf{\Box} s
  \]
  
  the leftmost leaf is $[A]_s[\top / \mathbf{\Box} s]$ and its sibling $[B]_s[\top / \mathbf{\Box} s]$
A choice that changes the $\mathcal{H}$ store

The following ClpZinc program (CLP):

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

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

```
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 $H$ store

The following ClpZinc program (CLP):

\[
\begin{align*}
\text{var 1..10: } &\ x; \\
\text{var 1..10: } &\ y; \\
\text{:- (x = 1; y = 1), x } &\leq y.
\end{align*}
\]

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

\[
\begin{align*}
\text{var 1..10: } &\ x; \\
\text{var 1..10: } &\ y; \\
\text{var 0..1: } &\ X1; \\
\text{constraint } &\ X1 = 0 \implies x = 1; \\
\text{constraint } &\ X1 = 1 \implies y = 1; \\
\text{constraint } &\ x \leq y; \\
\text{solve : : seq_search([}
\text{     \int_search([X1], input_order, indomain_min, complete)\]}) satisfy; \\
\end{align*}
\]
Indexicals

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

And-or trees for dichotomous search with indexicals

\[
\begin{align*}
\land &\quad \land &\quad x \leq (X_1 + X_2) \div 2 \\
\lor &\quad \land &\quad x > (X_1 + X_2) \div 2 \\
\land &\quad \land &\quad x \leq (X_3 + X_4) \div 2 \\
\lor &\quad \land &\quad x > (X_3 + X_4) \div 2 \\
\land &\quad \land &\quad x \leq (X_5 + X_6) \div 2 \\
\lor &\quad \land &\quad x > (X_5 + X_6) \div 2
\end{align*}
\]

\[
\begin{align*}
&\text{indexical\_min}(X_1, x) \\
&\text{indexical\_max}(X_2, x) \\
&\text{indexical\_min}(X_3, x) \\
&\text{indexical\_max}(X_4, x) \\
&\text{indexical\_min}(X_5, x) \\
&\text{indexical\_max}(X_6, x)
\end{align*}
\]
Search Transformers via Meta-interpretation

Meta-interpretation

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

Symmetry breaking during search in constraint programming

`sbsd(top, _).`
`sbsd(or(A, B), Path) :-`
    (`A = constraint(C, A0),`
     (`C, sbsd(A, [C | Path])`
      ;  cut_symmetry(C, Path), sbsd(B, Path))
     ;  A \= constraint(_, _),
     (`sbsd(A, Path) ; sbsd(B, Path))`).
`sbsd(constraint(C, T), Path) :- C, sbsd(T, [C | Path]).`
`:= search_tree(labeling_list(queens, 1, n), T),
sbsd(T, []).`
Exponential speed-up with LDS

var 0..1: x;
var 0..1: y;
array[0..n] of var 0..1: a;

:- int_search(a, input_order, indomain_min, complete),
lds(((x = 0; x = 1), (y = 0; y = 1)), 0), x != y.

a: $2^n$ nodes to explore
Beyond And-Or Trees

State variables, persistent through backtracking.

\[
\text{annotation store(var bool: } c, \text{ string: id, array[int] of var int: src);}
\]

\[
\text{annotation retrieve(string: id, array[int] of var int: target);}\]

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.

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

- Lazy clause generation?