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

$=$

Constraint model $+$ Search procedure

- relational
- high level
- MiniZinc

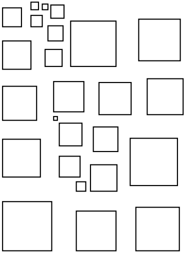
- hardly declarative
- very dependent to the solver
- low-level languages

Search procedures are crucial to solve hard combinatorial (typically, NP-complete) problems.

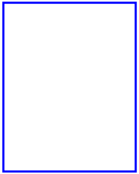
Rectangle-packing Problem (or Korf's Problem)

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?

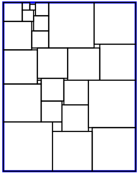
NP-complete (by reduction from bin-packing).

- Joseph Y. T. Leung, Tommy W. Tam, C. S. Wong, Gilbert H. Young, Francis Y. L. Chin. *Packing squares into a square*. Journal of Parallel and Distributed Computing, Vol. 10, No. 3. (November 1990).

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).

- Joseph Y. T. Leung, Tommy W. Tam, C. S. Wong, Gilbert H. Young, Francis Y. L. Chin. *Packing squares into a square*. Journal of Parallel and Distributed Computing, Vol. 10, No. 3. (November 1990).

A CP Strategy for the Rectangle-Packing Problem

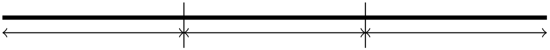
- H. Simonis, B. O'Sullivan. *Search Strategies for Rectangle Packing*. Principles and Practice of Constraint Programming, CP 2008.

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

Strategy:

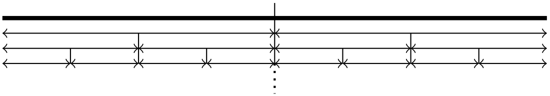
1. interval splitting on $x_n, x_{n-1}, \ldots, x_1$,

   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.

$$\bigcap_i [x_i, x_i + w_i[\times [y_i, y_i + h_i[= \varnothing$$

- 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 constraints

Constraint model $\mathcal{M}$ → Constraint model $\mathcal{M} \uplus \mathcal{M}_t$

Tree search procedure t $\underset{\text{computationally similar}}{\sim}$ Basic labeling t'

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

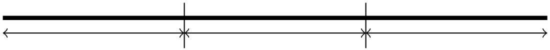
Contributions:

- A high-level language for tree search procedure (ClpZinc),
- The resulting model can be solved by any solver.

Arithmetic constraint for Interval Splitting

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

x:

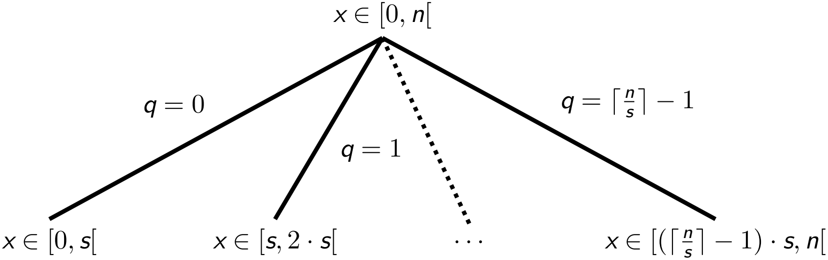


Obtained by **domain filtering** and **constraint propagation** of the **Euclidean division** equation.

$$x = s \times q + r$$

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

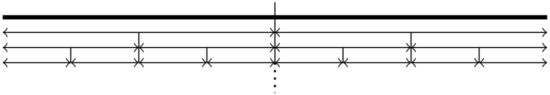
Interval Splitting: The Search Tree



Arithmetic constraint for Dichotomic Search

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

x_i:

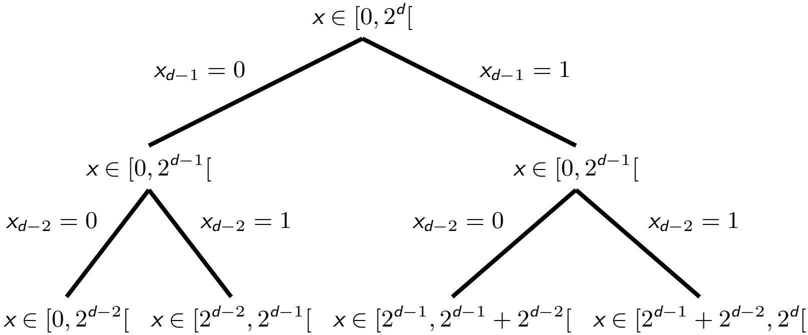


Obtained by **domain filtering** and **constraint propagation** of the **binary decomposition of x**.

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

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

Dichotomic Search: The Search Tree



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):

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

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

```
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):

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

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

```
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):

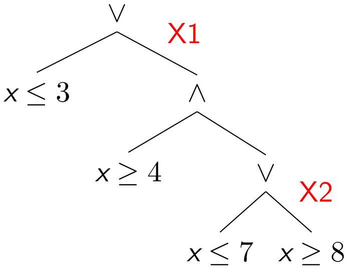
```
var 1..10: x;
:- (x <= 3 ; x >= 4, (x <= 7 ; x >= 8)).
```

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

```
var 1..10: x;
var 0..1: X2;  var 0..1: X1;
constraint X1 = 0 -> x <= 3;
constraint X1 = 1 -> x >= 4;
constraint X1 = 1 /\ X2 = 0 -> x <= 7;
constraint X1 = 1 /\ X2 = 1 -> x >= 8;
constraint X1 = 0 -> X2 = 0;
solve :: seq_search([
    int_search([X1], input_order, indomain_min, complete),
    int_search([X2], input_order, indomain_min, complete)
  ]) 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

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

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

```
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

```
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 $\mathcal{X}$

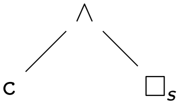
Translation function with environment $[\![\cdot]\!]_s$ to trees with holes $\square_s$.

$$[\![\texttt{true}]\!]_s \longrightarrow \square_s$$

$$[\![\texttt{false}]\!]_s \longrightarrow \bot$$

$$[\![\texttt{X = v}]\!]_s \longrightarrow \square_{s \wedge (\texttt{X = v})}$$

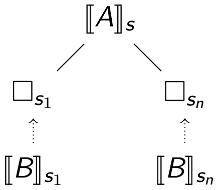
$$[\![\texttt{c}]\!]_s \longrightarrow$$



where c is a constraint
or a search annotation

Translation for sequences

$[\![\texttt{A,B}]\!]_s \longrightarrow$



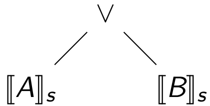
all $\Box_{s_i}$ of $[\![A]\!]_s$ are filled with $[\![B]\!]_{s_i}$

i.e., $[\![A]\!]_s[\forall i,\ [\![B]\!]_{s_i}/\Box_{s_i}]$

Translation for choices

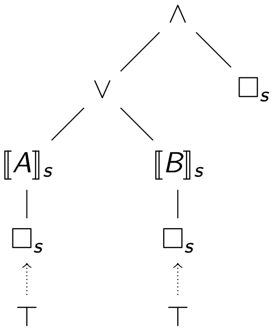
- if A or B changes the store, *i.e.*, $\exists s' \neq s$, $\square_{s'} \in [\![A]\!]_s$ or $[\![B]\!]_s$:

$[\![\texttt{A;B}]\!]_s \longrightarrow$



- if neither A nor B changes the store:

$[\![\texttt{A;B}]\!]_s \longrightarrow$



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):

```
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 $\mathcal{H}$ store

The following ClpZinc program (CLP):

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

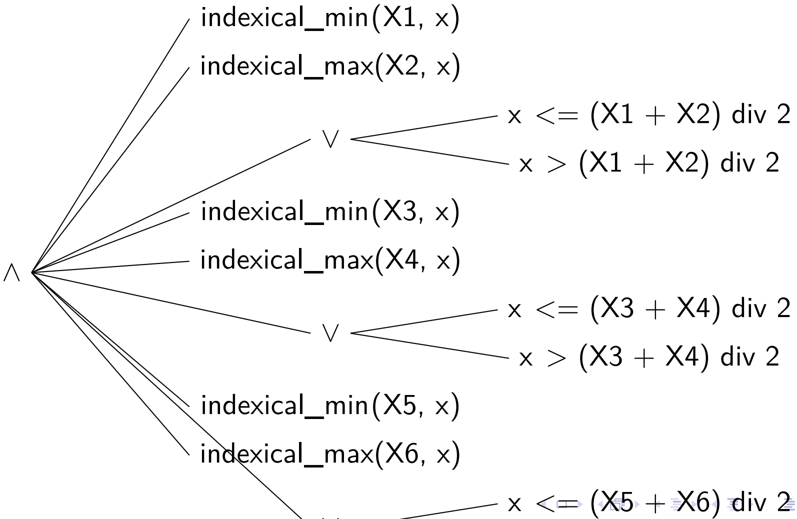
can be reified into the following MiniZinc program (CSP):

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

Indexicals

```
int_search([_x1], input_order, min(x), complete),
int_search([_x2], input_order, max(x), complete)
```

And-or trees for dichotomic search with indexicals



Search Transformers via Meta-interpretation

Meta-interpretation

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

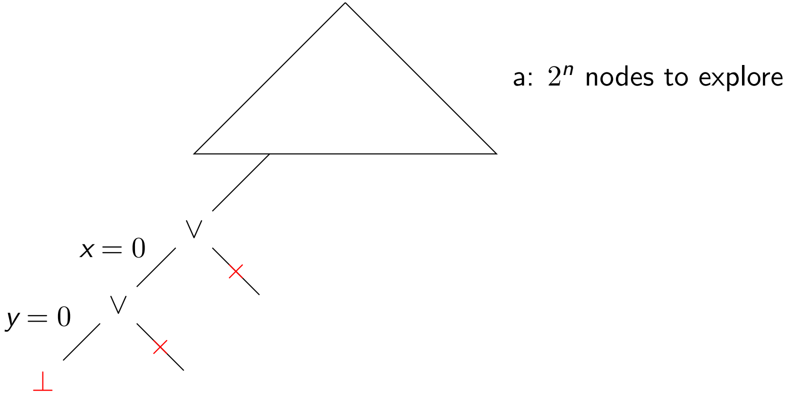
Symmetry breaking during search in constraint programmingIn Proceedings ECAI'2000, pages 599–603, 1999

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

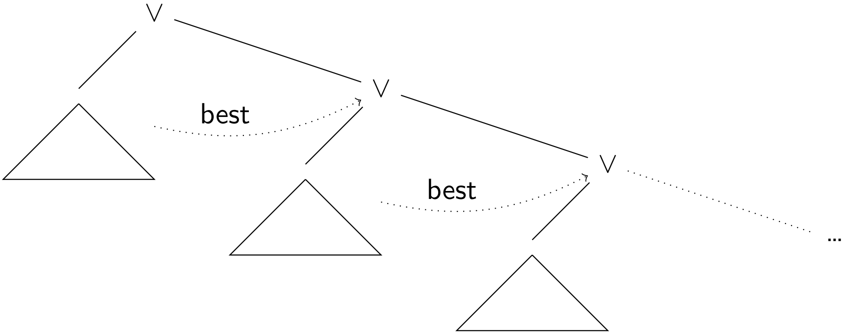


Beyond And-Or Trees

State variables, persistent through backtracking.

```
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.
  $\Rightarrow$ solver-independent high-level search specification/modelling.
- Constraint Logic Programming programs can be compiled into Constraint Solving Problems!
  $\Rightarrow$ 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?