Homework 2
ID2204
Constraint Programming

Kim Hammar
Mallu
Due Date: 7 April 2017

Sudoku

Constraints

The sudoku model uses $9 + 9 + 9$ distinct constraints:

$\forall r \in$ rows $\quad distinct(r)$
$\forall c \in$ columns $\quad distinct(c)$
$\forall s \in$ 3x3-squares $\quad distinct(s)$

Branching and Propagation

The solvability of this sudoku model depends a lot on the branching strategy and propagator strengths. For instance if we choose to go with a lower propagation constraint, which implies more search, the branching strategy is very essential. E.g when using `IPL_VAL` as propagation level for all constraints, if we choose first-fail branching heuristic the model is solved rather quickly ($<$ 1s, 214 nodes, 103 failures for sudoku instance 0), however if we choose the last-fail branching heuristic the model is not solved even solved after $>$ 2min search. A very interesting note here is that if we choose a stronger propagation level, like `IPL_DOM` the branch strategy is of less importance since some sudoku instances can even be solved without any search at all by using strong propagation, e.g no matter the branching strategy sudoku instance 0 can be solved in the same time ($<$ 1s, 1 node, 0 failures, 172 propagations) by using propagation level `IPL_DOM`.

Illustration of the difference in amount of search necessary:

Figure 1: Search tree for propagation level `IPL_DOM`

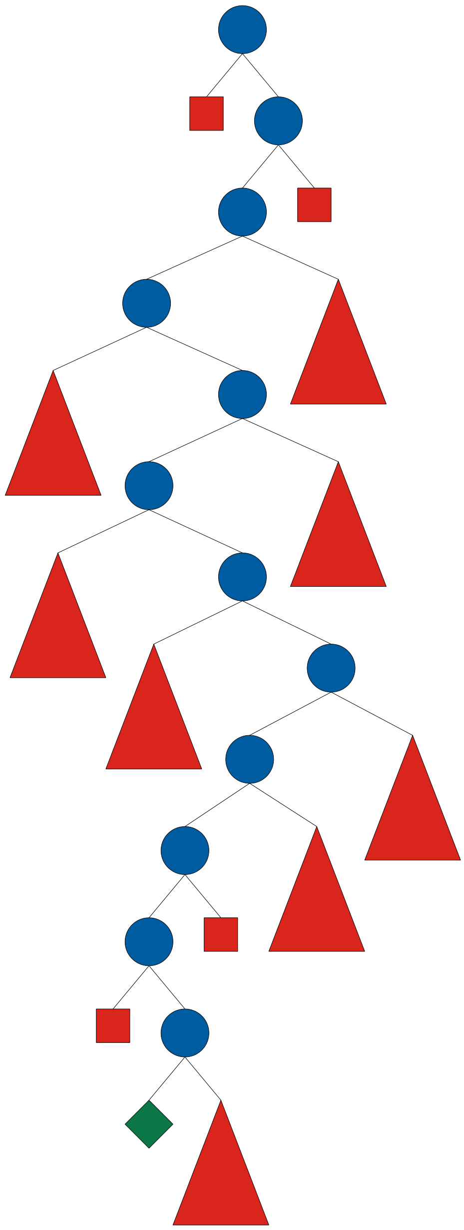

Figure 2: Search tree for propagation level `IPL_VAL` and first-fail branch heuristic

`IPL_SPEED` and `IPL_BASIC` propagation resulted in similar search tree as `IPL_VAL`.

Bounds-consistency gave propagation somewhere inbetween `IPL_DOM` and `IPL_VAL` and solved sudoku instance 0 in less than $1s$ with 23 nodes and 10 failures. With bounds-consistency it is also possible to solve sudoku instance 0 with last-fail heuristic in less than $1s$ (39 nodes, 18 failures).

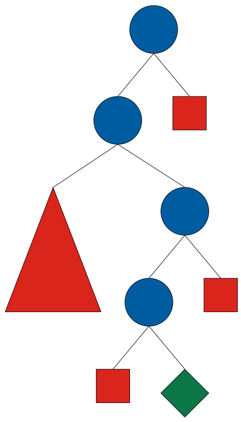

Figure 3: Search tree for propagation level `IPL_BND` and first-fail branch heuristic

To conclude, for this sudoku model stronger propagation is preferred, with `IPL_DOM` it is possible to achieve backtrack-free search and a solution in polynomial time.

n-Queens With 0/1 Variables

Modelling the problem as a $n \times n$ matrix using 0/1 variables means having one constraint per row, column, diagonal.

Formulation as a CSP

n-Queens, $n = 4$
$CSP = \langle \mathcal{V}, \mathcal{U}, \mathcal{C} \rangle$

Variables $\mathcal{V} = \{x_{1,1}, \ldots, x_{n,n}\}$ $(x_{col,row})$

Universe $\mathcal{U} = \{0, 1\}$

Constraints $\mathcal{C} = \{c_1, c_2, c_3, c_4, ...c_{18}\}$
Constraints for columns: $\{c_1, \ldots, c_4\}$
Constraints for rows:$\{c_5, \ldots, c_9\}$
Constraints for diagonals:$\{c_{10}, \ldots, c_{18}\}$

Example constraint c_1 for column 1:
$\text{var}(c_1) = \langle x_{1,1}, x_{1,2}, x_{1,3}, x_{1,4} \rangle$
$\text{sol}(c_1) = \{\langle 1, 0, 0, 0 \rangle, \langle 0, 1, 0, 0 \rangle, \langle 0, 0, 1, 0 \rangle, \langle 0, 0, 0, 1 \rangle\}$

Example constraint c_5 for row 1:
$\text{var}(c_5) = \langle x_{1,1}, x_{1,2}, x_{1,3}, x_{1,4} \rangle$
$\text{sol}(c_5) = \{\langle 1, 0, 0, 0 \rangle, \langle 0, 1, 0, 0 \rangle, \langle 0, 0, 1, 0 \rangle, \langle 0, 0, 0, 1 \rangle\}$

Example constraint c_{10} for diagonal $x_{1,4} - x_{4,1}$:
$\text{var}(c_{10}) = \langle x_{1,4}, x_{2,3}, x_{3,2}, x_{4,1} \rangle$
$\text{sol}(c_{10}) = \{\langle 1, 0, 0, 0 \rangle, \langle 0, 1, 0, 0 \rangle, \langle 0, 0, 1, 0 \rangle, \langle 0, 0, 0, 1 \rangle \langle 0, 0, 0, 0 \rangle\}$

Model

Variables

`Matrix<IntVarArray>` of size $n \times n$

Constraints

The constraints can be implemented in Gecode by asserting that the sum of the rows and columns should be 1 (no two queens can share row/column) and that for each diagonal the sum can at most be 1 (there can only be at most one queen in each diagonal).

$\forall c \in$ Matrix.columns $\quad c.sum() \equiv 1$
$\forall r \in$ Matrix.rows $\quad r.sum() \equiv 1$
$\forall d \in$ Matrix.diagonals $\quad d.sum() \leq 1$

Branching

What can you do about branching? Not that much as it seems (compared to other constraint problems). Since the variable domains are from the start very small $\{1, 2\}$, the effect of switching between for example `INT_VAR_SIZE_MIN` and `INT_VAR_SIZE_MAX` is not as evident compared to the queen-model with variable domains of larger size ($\{1, \ldots, 9\}$).

After doing some experiments mainly with `INT_VAR_NONE`, `INT_VAR_SIZE_MIN`, `INT_VAR_RND`, `INT_VAL_SIZE_MAX`, `INT_VAL_SIZE_MIN`. We have found that `INT_VAR_SIZE_MIN` (first-fail heuristic) was the most successful, choosing the value (0 or 1) heuristic was of minor importance regarding performance, but of course affected what type of solutions was found first.

The first-fail-heuristic outperformed `INT_VAR_RND` by a factor of ~ 5 and in fact `INT_VAR_NONE` and `INT_VAR_SIZE_MIN` gave similar results, fail-first slightly better only.

$n = 5$, first-fail vs random heuristic search trees:

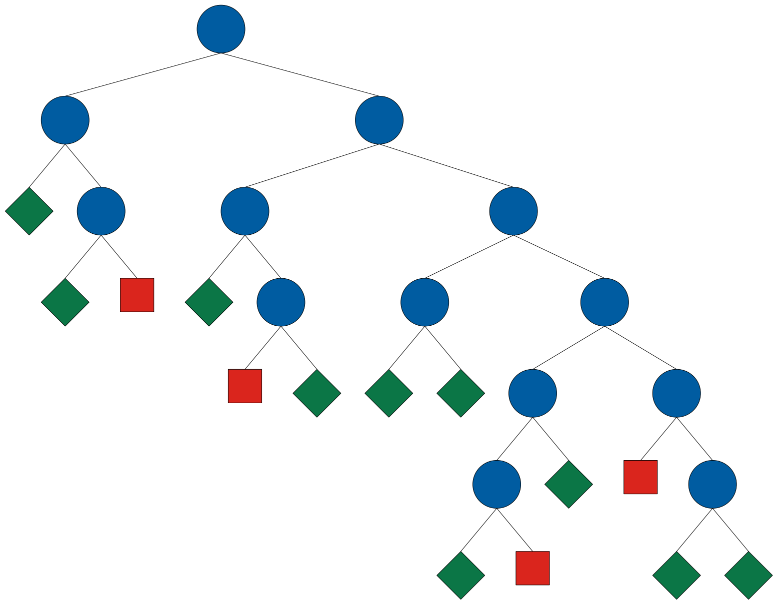

Figure 4: First-fail heuristic finding all solutions for problem instance $n = 5$

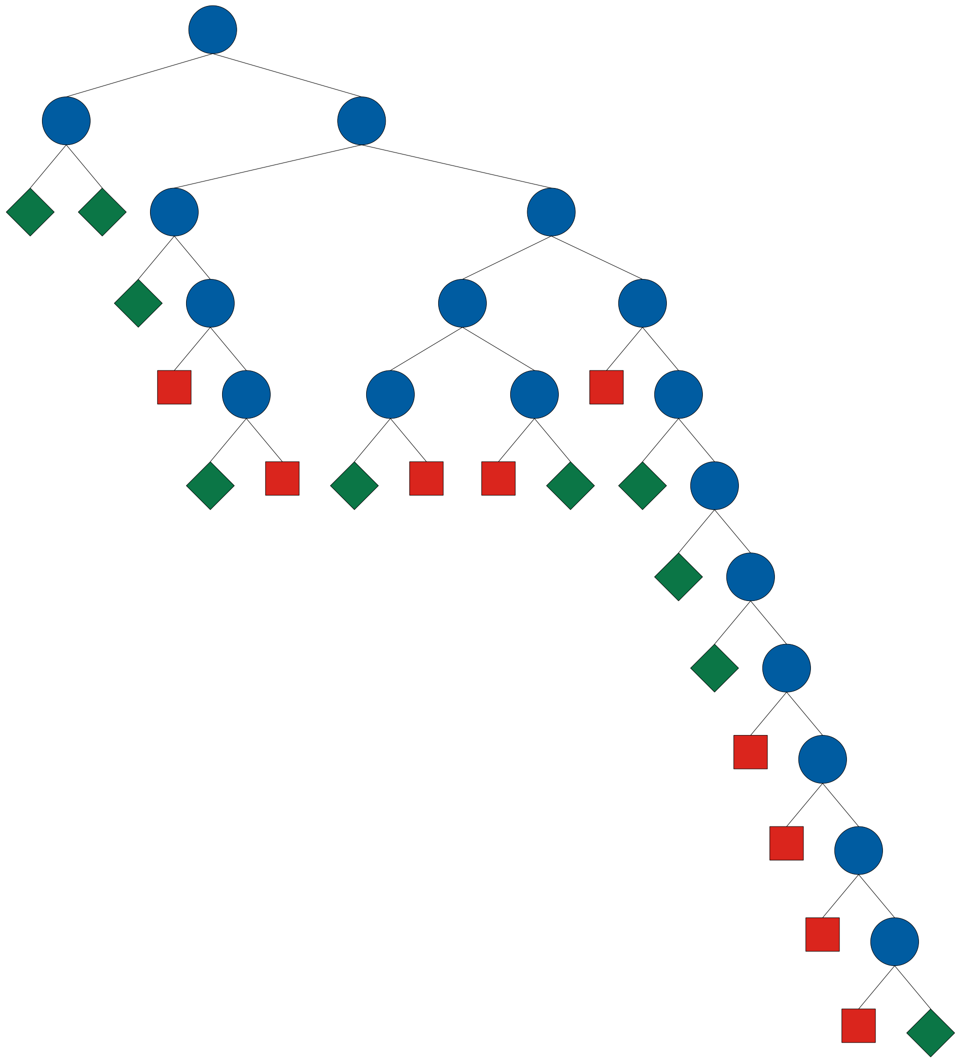

Figure 5: Random heuristic finding all solutions for problem instance $n = 5$

Pros and Cons

Pros:

- Less constraints than for example using the binary-constraints model in the example. Since propagators are ran one at a time this is in theory good from a performance perspective. Also, in some cases since propagation is not compositional, fewer larger constraints can give better propagation. (This pro however does not hold against the PROP_DISTINCT model in the example which only uses 3 constraints). Note: although our Matrix-model contains less propagators than the example-binary model, in practice our model is not more efficient.

- Very small domains for each value: $\{0, 1\}$ compared to the non-matrix approach with domains of $\{0, \ldots, n\}$, but this is a trade-off since the matrix-model contains a lot more variables (n^2 vs n) compared to the example.

- Very intuitive model and probably the model chosen by someone familiar with the n-queens problem but not with constraint-programming techniques.

Cons:

- In practice our solution with the Matrix-model is slower than the example solution.
- The matrix model is less memory-efficient.
- A lot more variables are necessary.
- The example enforces one constraint just by modelling in a smart way: one queen per column constraint is enforced by the one-dimensional array. This is not utilised in the matrix-model

How to run