Constraint Programming Tutorial 1: Magic Square

Startup

- Installation (in user space):

```
cp -fr /media/commun_mialp_eleves/temp/ppc ~/.
cd ~/ppc/facile
```

  then follow the steps indicated in the README file to build the library:

```
automake --add-missing
autoconf
./configure
make
```

- To use FaCiLe with the top-level:

```
ocaml -I ~/ppc/facile facile.cma
```

- Compilation by hand:

```
ocamlc -o prog -I ~/ppc/facile facile.cma prog.ml
ocamlopt -o prog -noassert -I ~/ppc/facile facile.cmxa prog.ml
```

- A generic *Makefile* is available in directory `ppc/tp` (where you should therefore create you constraint programs):

```
make prog.out
make prog.opt
```

  build an executable in bytecode `prog.out` or native code `prog.opt` respectively from source file `prog.ml`.

Magic Square

A magic square of order n is a $n \times n$ integer matrix such that:

- each integer is between 1 and n^2;
- all integers are pairwise different;
- the sum of each row, column and of both main diagonals are equal.

A solution of order $n = 3$:

2	7	6
9	5	1
4	3	8

1. With FaCiLe, write a constraint program that takes the order n as a command line parameter and solves this problem.
   Indications:
   - Use an "all different" constraint (`Alldiff.cstr`) with an array of all the variables.

- Function `Arith.sum_fd` returns an expression corresponding to the sum of an array of variables.To be able to use this function, arrays has to be built for lines, columns and diagonals:
  - `Array.init n f` builds a new array of size `n`, initializing each element with `f i`;
  - `Array.sub t start len` returns a subarray of array `t` starting at `start` with `len` elements.

2. Use the matching algorithm of the "all different" constraint (optional parameter `~algo` set to `Bin_matching`) with different waking events (`Fd.on_subst` or `Fd.on_refine`) and compare the respective number of backtracks and computation times.
   Indications:

   - Function `Goals.solve` has a `~control` optional parameter to execute a function at each backtrack. This function takes the number of backtracks since the start of the search as its single parameter.

     ```
     let print_bt = fun bt ->
       Printf.printf "\r%d%!" bt in
     if Goals.solve ~control:print_bt goal then
     [...]
     ```

   - Function `Sys.time: unit -> float` returns the number of seconds of processing used by the program.

3. Modify your code to generate all the solutions. For $n = 3$, 8 symmetrical solutions are obtained. Let m be the matrix corresponding to the magic square, break the symmetries by ordering the variables according to the following inequalities:

   - $m_{(1,1)} < m_{(n,n)}$
   - $m_{(1,1)} < m_{(1,n)}$
   - $m_{(1,1)} < m_{(n,1)}$
   - $m_{(n,1)} < m_{(1,n)}$

   State as comments to which symmetry corresponds each inequation.

4. Use a variable ordering strategy (*minimum domain size*) and compare the respective number of backtracks and computation time.
   Indications:

   - Use `Goals.Array.forall ~select:strategy Goals.indomain vars` to search variables array `vars` with heuristic `strategy`.
   - The minimum domain size strategy can be build with:

     ```
     let mindom =
       Goals.Array.choose_index
         (fun v1 v2 -> Fd.size v1 < Fd.size v2)
     ```

5. Let s be the sum of a column (*magic number*), express the total sum of the entire square as a function of s. Then express the total sum in another way to deduce the value of s and use the corresponding equation to post a redundant constraint. Compare the respective number of backtracks and computation time for increasing values of n.