

# Constraint Programming with ECL<sup>i</sup>PS<sup>e</sup>

Stefano Benedettini

Andrea Roli

Artificial Intelligence LM 2010

## Contents

|          |                                                     |          |
|----------|-----------------------------------------------------|----------|
| <b>1</b> | <b>ECL<sup>i</sup>PS<sup>e</sup> hints</b>          | <b>2</b> |
| <b>2</b> | <b>Constraint Satisfaction Problems: Modeling</b>   | <b>2</b> |
| 2.1      | Map-coloring . . . . .                              | 3        |
| 2.2      | Latin Square . . . . .                              | 3        |
| 2.3      | <i>N</i> -queens . . . . .                          | 3        |
| 2.4      | Cryptarithmic . . . . .                             | 3        |
| 2.5      | All-interval Series (adapted from CSPLib) . . . . . | 4        |
| 2.6      | Magic Square (adapted from CSPLib) . . . . .        | 4        |
| 2.7      | Sudoku . . . . .                                    | 4        |
| 2.8      | Futoshiki [optional] . . . . .                      | 5        |
| 2.9      | Minesweeper [advanced] . . . . .                    | 5        |
| 2.10     | Working with sets [optional] . . . . .              | 5        |
| 2.10.1   | Steiner Triplets (adapted from CSPLib) . . . . .    | 5        |
| 2.10.2   | Schur's Lemma (adapted from CSPLib) . . . . .       | 5        |
| <b>3</b> | <b>Constraint Satisfaction Problems: Searching</b>  | <b>6</b> |
| 3.1      | Squeezing the most out of your models . . . . .     | 6        |
| 3.2      | Magic Sequence (adapted from CSPLib) . . . . .      | 6        |
| 3.3      | Partial Latin Square . . . . .                      | 7        |
| <b>4</b> | <b>Constraint Optimization Problems</b>             | <b>7</b> |
| 4.1      | Maximum Clique problem . . . . .                    | 7        |
| 4.2      | Chromatic Number . . . . .                          | 7        |
| 4.3      | Golomb Rulers (adapted from CSPLib) . . . . .       | 8        |
| 4.4      | Domination Number . . . . .                         | 8        |

## Topics

- Constraint logic programming

- Constraint satisfaction problems (CSP)
- Constraint optimization problems (COP)
- Search methods for CSPs and COPs

## 1 ECL<sup>i</sup>PS<sup>e</sup> hints

- Use `tk` interface to ECL<sup>i</sup>PS<sup>e</sup> and forget about the command line: it's much more user-friendly!
- You can write solver code for an exercise on a single file or put every solver on the same file, it's up to you, but do not try to solve a problem on the ECL<sup>i</sup>PS<sup>e</sup> command line because it gets messy pretty soon.
- In order to use an ECL<sup>i</sup>PS<sup>e</sup> library you must import that library using `lib(library-Name)` in your source files; typically you will always have to write `:- lib(ic)` in the header of a source file.
- In order to use a predicate in a user provided source file, you must *compile* it: there is a convenient menu item for that.
- In order to use a user module anywhere you must import it: in a source file use `:- use_module(moduleName)`, in the GUI look for "Use module..." menu item.
- Disambiguate names using the scope operator ":" (don't worry ECL<sup>i</sup>PS<sup>e</sup> will complain about ambiguities). For instance, if you import `ic` and `ic_global` you have in scope two `alldifferent` predicates, and ECL<sup>i</sup>PS<sup>e</sup> will surely raise a compilation error.

## 2 Constraint Satisfaction Problems: Modeling

In this section you will be asked to model constraint satisfaction problems. The focus will be mainly on the modeling part and not on searching (for that see 3); that means that the majority of the problems faced can be easily solved by a naïve labeling strategy, if not by propagation alone (for example see 2.7).

You are required to use ECL<sup>i</sup>PS<sup>e</sup> for your models and are strongly encouraged to browse the on-line documentation (you can also find it on your local machine) by yourselves in order to find efficient, ready-made constraint definitions and utility predicates that will save you a lot of time and effort (for example, `lists:flatten` flattens a nested list). Remember that we will use only the `ic` library and its companions, like `ic_global`. But later 4 we will explore also `branch_and_bound`.

In the attached material you can find some further utility predicates, in the form of an ECL<sup>i</sup>PS<sup>e</sup> module, that simplify file reading and other stuff. This can be especially useful for some problems, like 2.7 or 2.9.

And now, on with exercises!

## 2.1 Map-coloring

1. Model and solve the Australia map-coloring instance on Russel and Norvig's book. Hard-code constraints in your model.
2. Come up with a more general model for solving a map-coloring problem. Users should be able to declaratively specify neighborhood relations (for example using Prolog facts). Probably you might need a Prolog structure to associate a country name with its decision variable. Encode and solve again the Australia map-coloring instance. [advanced]

## 2.2 Latin Square

A Latin Square is a  $n \times n$  table filled with  $n$  different symbols such that no symbol occurs twice in each row and column. Use integers instead of symbols without restraint.

1. Write a general model for a Latin Square of size  $n$ , where  $n$  is an input.
2. Latin Square problem has a lot of symmetries; can you add some constraints to remove some? [optional]
3. You should be able to solve quickly enough instances up to size 15 or so. For bigger instances a more clever search scheme is required (see 3.1).
4. Do you think that solutions to a Latin Square have interesting algebraic properties? What if we use a solution as “multiplication table” for a binary operator? [optional]

## 2.3 $N$ -queens

The famous  $n$ -queens problem.

1. Write a model for every  $n$ .
2. Think about symmetries in this problem and add some constraints to remove some of them.

## 2.4 Cryptarithmic

Cryptarithmic puzzles consist of mathematical equations among unknown numbers, whose digits are represented by letters. One prominent example is `SEND + MORE = MONEY`, but there are lots of them on the Internet. Write models for some of these puzzles:

1. `SEND + MORE = MONEY`
2. `TWO + TWO = FOUR`

3. LOTS + SOLD + TO + OLD = FOOLS

4. CROSS + ROADS = DANGER

5. LINDON × B = JOHNSON

## 2.5 All-interval Series (adapted from CSPLib)

An interesting problem that has application in music, is the All-interval Series. Mathematically you are to find a vector  $s = (s_1, \dots, s_n)$  where  $s_i \in \mathbb{Z}_n$  such that:

- $s$  is a permutation of  $\mathbb{Z}_n = \{0, 1, \dots, n-1\}$ ;
- the interval vector  $v = (|s_2 - s_1|, |s_3 - s_2|, \dots, |s_n - s_{n-1}|)$  is a permutation of  $\mathbb{Z}_n \setminus \{0\} = \{1, 2, \dots, n-1\}$

Write a CSP formulation for this problem.

## 2.6 Magic Square (adapted from CSPLib)

An order  $n$  magic square is a  $n \times n$  matrix containing the integers from 1 to  $n^2$ , with each row, column, main and secondary diagonals equal the same sum. This sum  $M$  is called *magic constant* and equal to:

$$M = \frac{n(n^2 + 1)}{2}$$

I strongly suggest you to use this information in your model.

Write a CSP formulation for this problem. Note that for  $n > 5$  the problem is quite intractable (at least *my* formulation on *my* machine was!)

## 2.7 Sudoku

If you don't know Sudoku you probably just awoke from a long slumber, hibernation or barbiturate-induced coma. For those of you who have been away from Earth these last few years, Sudoku is a number puzzle where a  $9 \times 9$  grid, further subdivided into 9  $3 \times 3$  square regions, is to be filled with natural numbers from 1 to 9 such that in no row, column or square region a number is repeated. Here is the Wikipedia page on Sudoku.

1. Write a general CSP formulation for this problem. Suppose that your solver is fed with a partially filled instance, like those you find on magazines; instance format is up to you: nested lists, arrays or whatever.
2. Write a solver that can interface with I/O and read an instance from a file. [optional]

## 2.8 Futoshiki [optional]

Futoshiki is a Sudoku-like number placement puzzle. A  $n \times n$  grid has to be filled with integers such that each row and column contain all the numbers in  $1, 2, \dots, n$ . In addition, some cells are constrained to contain a number that is greater than a neighboring cell content.

Write a general CSP formulation for this problem. The difficulty here is in instance format representation. Basically, you have to provide, beside grid-size  $n$ , all the hints, that is the initially placed numbers (you can freely recycle the code you wrote for Sudoku), and the constraints between cells (a pair of grid cell coordinates will do).

## 2.9 Minesweeper [advanced]

Minesweeper is the pencil-and-paper version of the famous video game, with the very same rules. The objective is to find all the mines in a rectangular grid knowing:

- the number of mines;
- that the 8 neighboring cells to a cell containing a number contain exactly that many mines;
- that a numbered cell never contains a mine.

Write a general formulation for this problem that reads an instance from a file. Constraint setup and instance reading can be independently designed: first you could design a solver that takes a hand-made structure and then you could work on the I/O or vice-versa.

## 2.10 Working with sets [optional]

These two problems involve set variables.

### 2.10.1 Steiner Triplets (adapted from CSPLib)

The ternary Steiner problem of order  $n$  consists of finding a set of  $\frac{n \cdot (n-1)}{6}$  triples of distinct integer elements in  $\{1, \dots, n\}$  such that any two triples have at most one common element.

Model and solve for  $n = 7$  using set variables.

### 2.10.2 Schur's Lemma (adapted from CSPLib)

The problem is to put  $n$  balls labelled  $\{1, \dots, n\}$  into  $k$  boxes so that for any triple of balls  $(x, y, z)$  with  $x + y = z$ , not all are in the same box.

1. Write a formulation using integer variables and solve for  $k = 3$ . (Hint: reduce it to an assignment problem, i.e., use binary variables.)

2. Write a formulation using set variables.

### 3 Constraint Satisfaction Problems: Searching

For this part of the exercise, you will learn how to use more refined and customizable search methods provided by `ECLiPSe` libraries. Our focus will mainly be on `ic:search` predicate.

#### 3.1 Squeezing the most out of your models

You will probably have noticed that some of your model, although correct, are quite slow to solve and that simple labeling does not scale well with problem size. This exercise will get you started with `ECLiPSe` built-in search capabilities by solving previously proposed problems by means of a different strategy.

The point is to find a good combination of heuristics strategies for *variable selection* and *value selection*.

A useful capability of `ic:search` is to provide the number of backtracks in the whole search process. Check this up in the documentation. Usually the fewer backtracks you encounter the faster a strategy is.

1. Find a better strategy for large ( $n > 20$ ) instances of Latin Square.
2. Find a strategy (a bit) faster than labeling for the Magic Square.
3. Find a suitable heuristic for large  $n$ -queens instances (at least  $n = 20$  solved in about a second).

#### 3.2 Magic Sequence (adapted from CSPLib)

A magic sequence of length  $n$  is a sequence of integers  $x_0, x_1, \dots, x_{n-1}$  where  $x \in [0, n-1]$ , such that  $\forall i \in [0, n-1]$ , the number  $i$  occurs exactly  $x_i$  times in the sequence. For instance,  $\langle 6, 2, 1, 0, 0, 0, 1, 0, 0, 0 \rangle$  is a magic sequence since 0 occurs 6 times in it, 1 occurs twice, etc...

1. Write a CSP model for this problem parametrized in  $n$ .
2. First try to solve it through labeling. Isn't the search taking a lot for  $n > 15$ ? If it's not, increase  $n$  until it does. Can you find a better search strategy, knowing how a solution looks like?
3. Sometimes adding redundant constraints helps propagation. Magic sequences have this property:

$$\sum_{i=0}^{n-1} i x_i = n$$

Exploit it to write a *much faster* solver.

### 3.3 Partial Latin Square

Partial Latin Square problem or Quasigroup Completion problem is a *much more* challenging problem than plain Latin Square which shows, as you know, also a phase transition. Using `partial_latin_square` ECL<sup>i</sup>PS<sup>e</sup> module attached to this document, try and solve some of the proposed instances.

Mind that they have all been generated on the phase transition so they will be hard! (Maybe too hard!)

## 4 Constraint Optimization Problems

ECL<sup>i</sup>PS<sup>e</sup> is also capable of solving constraint optimization problems. Among the many methods available, the most straightforward to use is Branch-and-Bound in `branch_and_bound` library. ECL<sup>i</sup>PS<sup>e</sup> implementation of B&B takes as input a search strategy and an objective function  $f$ , to be minimized, modeled as a variable. The solver iteratively improves on the incumbent solution  $s'$  by imposing the constraint  $f(s) < f(s')$ ; each time a better solution is found, this constraint is updated.

### 4.1 Maximum Clique problem

In graph theory, a  $k$ -clique  $\mathcal{C}$  of an undirected graph  $G$  is a subset of the vertices of  $G$  which has cardinality  $k$  and such that for every two vertices there exists an edge connecting the two in  $G$ . Therefore  $\mathcal{C}$  induces a complete graph in  $G$  (it has  $\binom{k}{2}$  edges).

Browse `ai_util` ECL<sup>i</sup>PS<sup>e</sup> module attached to this document, look for `read_graph` and `adjacency_matrix` predicates and understand what they do. You can find some graph descriptions in a companion file of this exercises.

1. Model this problem as a COP.
2. Successfully solve it on some graphs.

### 4.2 Chromatic Number

The chromatic number  $\chi(G)$  of a graph  $G$  is the minimum number of color needed to color each vertex in  $G$  such that no two adjacent vertices have the same color.

Utilize again the predicates in `ai_util` module and the same graph descriptions you used for the Maximum Clique.

1. Model and solve this problem as a COP.
2. What is the relationship between this problem and the Maximum Clique problem?

### 4.3 Golomb Rulers (adapted from CSPLib)

A Golomb ruler may be defined as a set of  $m$  integers  $0 = a_1 < a_2 < \dots < a_m$  such that the  $\frac{m \cdot (m-1)}{2}$  differences  $a_j - a_i, 1 \leq i < j \leq m$  are distinct. Such a ruler is said to contain  $m$  marks and is of length  $a_m$ . The objective is to find optimal (minimum length) or near optimal rulers.

1. Model this problem as a COP and solve it.
2. If you use the vanilla `labeling` predicate chances are that you loop forever. Can you figure out why? Hint: it has something to do with value selection strategy.
3. This problem is particularly difficult for  $m > 8$ . Try to remove the following symmetry and see if it helps:

$$a_2 - a_1 < a_m - a_{m-1}$$

that is to say that the first difference is less than the last.

### 4.4 Domination Number

The domination number of a  $n \times n$  chess board is the minimum number of queens to place in order to attack or occupy every square.

1. Model this problem for any given  $n$  as a COP.
2. Demonstrate that the domination number for a  $8 \times 8$  chessboard is 5.
3. Can you find domination number for other chess pieces? Try refactor your solver in order to be able to easily switch between pieces. Find the domination number for kings, bishops and rooks on a chessboard of your choice. [optional]