XCSP3 Competition 2018
Proceedings

Christophe Lecoutre and Olivier Roussel
CRIL CNRS, UMR 8188
University of Artois, France
{lecoutre,roussel}@cril.fr

December 3, 2018
This document represent the proceedings of the XCSP3 Competition 2018. The website containing all detailed results is available at:

http://www.cril.fr/XCSP18/
Contents

1 About the Selection of Problems in 2018 5

2 Problems and Models 9
   2.1 Auction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
   2.2 BACP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
   2.3 BIBD . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
   2.4 Car Sequencing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
   2.5 Coloured Queens . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
   2.6 Crosswords (Satisfaction) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
   2.7 Crosswords (Optimization) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
   2.8 Dubois . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
   2.9 Eternity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
   2.10 FAPP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
   2.11 FRB . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
   2.12 Golomb Ruler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
   2.13 Graceful Graph . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
   2.14 Graph Coloring . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
   2.15 Haystacks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
   2.16 Knapsack . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
   2.17 Langford . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
   2.18 Low Autocorrelation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
   2.19 Magic Hexagon . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
   2.20 Magic Square . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
   2.21 Mario . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
   2.22 Mistery Shopper . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
   2.23 Nurse Rostering . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
   2.24 Peacable Armies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
   2.25 Pizza Voucher . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
   2.26 Pseudo-Boolean . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
   2.27 Quadratic Assignment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
   2.28 Quasigroup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
   2.29 RCPS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
   2.30 RLFAP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
   2.31 Social Golfers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
   2.32 Sports Scheduling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
   2.33 Steel Mill Slab . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
   2.34 Still Life . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
   2.35 Strip Packing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
   2.36 Subgraph Isomorphism . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
   2.37 Sum Coloring . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
2.38 TAL ................................................................. 56
2.39 Template Design .............................................. 58
2.40 Traveling Tournament ........................................ 60
2.41 Travelling Salesman ......................................... 62

3 Solvers ..................................................................... 65
   BTD ........................................................................ 65
   BTD_12 ................................................................. 68
   Choco ................................................................. 70
   Concrete .............................................................. 73
   CoSoCo .............................................................. 78
   macht ................................................................. 80
   Mistral ............................................................... 82
   NACRE ............................................................... 85
   OscaR ................................................................. 86
   PicatSAT ........................................................... 88
   Sat4j .................................................................. 91
   sCOP ................................................................ 93

4 Results ..................................................................... 95
   4.1 Rankings .......................................................... 95
   4.2 A few Comments ............................................. 96
      4.2.1 Standard Track – CSP – Sequential .............. 96
      4.2.2 Standard Track – COP – Sequential .............. 99
Chapter 1

About the Selection of Problems in 2018

Remember that the complete description, Version 3.0.5, of the format (XCSP3) used to represent combinatorial constrained problems can be found in [3]. For the 2018 competition, we have limited XCSP3 to its kernel, called XCSP3-core. This means that the scope of XCSP3 is restricted to:

- integer variables,
- CSP and COP problems,
- a set of 21 popular (global) constraints for Standard tracks:
  - generic constraints: intension and extension
  - language-based constraints: regular and mdd
  - comparison constraints: allDifferent, allEqual, ordered and lex
  - counting/summing constraints: sum, count, nValues and cardinality
  - connection constraints: maximum, minimum, element and channel
  - packing/scheduling constraints: noOverlap and cumulative
  - circuit, instantiation and slide

and a small set of constraints for Mini-solver tracks.

For the 2018 competition, 41 problems have been selected. They are succinctly presented in Table 1.1. For each problem, the type of optimization is indicated (if any), as well as the involved constraints. At this point, do note that making a good selection of problems/instances is a difficult task. In our opinion, important criteria for a good selection are:

- the novelty of problems, avoiding constraint solvers to overfit already published problems;
- the diversity of constraints, trying to represent all of the most popular constraints (those from XCSP3-core) while paying attention to not over-representing some of them (in particular, second class citizens);
- the scaling up of problems.
### Table 1.1: Selected Problems for the 2018 Competition. New problems are indicated in italic font. VAR means that a variable must be optimized. For RLFAP, the type of objective differs depending on instances. When extension is followed by (*), it means that short tables are considered.
Novelty. More than one third of the problems are new (15 out of 41, i.e. 36.5%). They are Auction, BACP, Crosswords Design, Eternity, FAPP, Mistery Shopper, Nurse Rostering, Peacable Armies, Pizza Voucher, RLFAP, Steel Mill Slab, Sum Coloring, TAL, Template Design, and Traveling Tournament. Some of these new problems are quite challenging; in particular, Crosswords design (an optimization problem with a total freedom on the position of black cells), FAPP (the original optimization instances from the ROADEF challenge), Nurse Rostering (an optimization problem involving many types of constraints), RLFAP (the original optimization instances from the “Centre d’Electronique de l’Armement”) and TAL (an optimization problem of natural language processing). It is important to note that the optimization FAPP and RLFAP instances, mentioned here, are far more difficult that the simplified satisfaction versions published in XCSP 2.1 some years ago.

Diversity. On the one hand, 5 constraints from XCSP3-core were not involved in 2018. They are mdd, allEqual, nValues, minimum and maximum. Very certainly, we shall try to foster mdd in next editions because of the growing interest [5, 18, 6, 17, 22] for that constraint, but note that regular is quite related to mdd. The constraint allEqual is basically an ease of modeling, because it trivially corresponds to a set of binary equality constraints. A related form of the constraint nValues is indirectly represented in some RLFAP instances where the type of the objective is NVALUES. Similarly, a related form of maximum is indirectly represented in 3 problems where the type of the objective is MAXIMUM. On the other hand, in many problems, one can observe the presence of intension and extension. The frequent occurrence of intension is quite natural since in many problems a few primitives (e.g., like $x < y$) are required. The frequent occurrence of extension can be explained by the usefulness of that constraint. Sometimes, ordinary and short tables simply happen to be simple and natural choices for dealing with tricky situations. This is the case when no known (global) constraint exists or when converting a logical combination of (small) constraints into a table is needed for filtering efficiency reasons. Basically, table constraints offer the user a direct way to handle disjunction (a choice between tuples), and this is clearly emphasized with smart tables [15], which could be introduced in next editions (possibly, in a special track). Another argument showing the importance of universal structures like tables, and also MDDs (Multi-valued Decision Diagrams), is the rising of tabulation techniques, i.e., the the process of converting sub-problems into tables (or MDDs), by hand, by means of heuristics [1] or by annotations [7].

Scaling up. It is always interesting to see how constraint solvers behave when the instances of a problem become harder and harder. This is what we call the scaling behavior of solvers. For most of the problems in the 2018 competition, we have selected series of instances with regular increasing difficulty. For example, selected instances for Crosswords Design follow an increasing order (size of the grid) from 4 to 15. Similarly, selected instances for Eternity also follow an increasing order (size of the puzzle) from 4 to 15.

Selection. This year, the selection of problems and instances has been performed by Christophe Lecoutre. As a consequence, the solver AbsCon didn’t enter the competition.
CHAPTER 1. ABOUT THE SELECTION OF PROBLEMS IN 2018
Chapter 2

Problems and Models

In the next sections, you will find all the models that have been developed for generating the XCSP3 instances. All these models are written in MCSP3 1.1 [14], which is the new version of MCSP3, officially released in December 2018 (with a full detailed documentation).

2.1 Auction

This is Problem 063 on CSPLib, called Winner Determination Problem (Combinatorial Auction).

Description (from Patrick Prosser on CSPLib)

“There is a bunch of people bidding for things. A bid has a value, and the bid is for a set of items. If we have two bids, call them A and B, and there is an intersection on the items they bid for, then we can accept bid A or bid B, but we cannot accept both of them. However, if A and B are bids on disjoint sets of items then these two bids are compatible with each other, and we might accept both. The problem then is to accept compatible bids such that we maximize the sum of the values of those bids.”

Data

As an illustration of data specifying an instance of this problem, we have:

```json
{
    "bids": [
        { "value": 10, "items": [1, 2] },
        { "value": 20, "items": [1, 3] },
        { "value": 30, "items": [2, 4] },
        { "value": 40, "items": [2, 3, 4] },
        { "value": 14, "items": [1] }
    ]
}
```

Model

The MCSP3 model used for the competition is:
class Auction implements ProblemAPI {
    Bid[] bids;

class Bid {
    int value;
    int[] items;
}

public void model() {
    int[] allItems = singleValuesFrom(bids, bid -> bid.items); // distinct sorted items
    int[] bidValues = valuesFrom(bids, bid -> bid.value);
    int nBids = bids.length, nItems = allItems.length;

    Var[] b = array("b", size(nBids), dom(0, 1),
        "b[i] is 1 iff the ith bid is selected");

    forall(range(nItems), i -> {
        int[] itemBids = select(range(nBids), j -> contains(bids[j].items, allItems[i]));
        if (itemBids.length > 1) {
            Var[] scope = select(b, itemBids);
            if (modelVariant("cnt"))
                atMost1(scope, takingValue(1));
            if (modelVariant("sum"))
                sum(scope, LE, 1);
        }
    }).note("avoiding intersection of bids");

    maximize(SUM, b, weightedBy(bidValues))
        .note("maximizing summed value of selected bids");
}
}

The model is rather elementary, involving 0/1 variables, and either the global constraint count (atMost1) with model variant 'cnt', or the global constraint sum with model variant 'sum'. To observe the efficiency of solvers with respect to these two global constraints, two series of 10 instances have been selected for the competition. Also, note that only the model variant 'sum' is compatible with the restrictions imposed for the mini-track.

2.2 BACP

This is Problem 030 on CSPLib, called Balanced Academic Curriculum Problem (BACP).

Description (from Brahim Hnich, Zeynep Kiziltan and Toby Walsh on CSPLib)

“The BACP is to design a balanced academic curriculum by assigning periods to courses in a way that the academic load of each period is balanced, i.e., as similar as possible. An academic curriculum is defined by a set of courses and a set of prerequisite relationships among them. Courses must be assigned within a maximum number of academic periods. Each course has associated a number of credits or units that represent the academic effort required to successfully follow it. The curriculum must obey the following regulations:

- Minimum academic load: a minimum number of academic credits per period is required to consider a student as full time.
- Maximum academic load: a maximum number of academic credits per period is allowed in order to avoid overload.
- Minimum number of courses: a minimum number of courses per period is required to consider a student as full time.
• Maximum number of courses: a maximum number of courses per period is allowed in order to avoid overload.

The goal is to assign a period to every course in a way that the minimum and maximum academic load for each period, the minimum and maximum number of courses for each period, and the prerequisite relationships are satisfied. An optimal balanced curriculum minimizes the maximum academic load for all periods.

Data

As an illustration of data specifying an instance of this problem, we have:

```json
{
    "nPeriods": 5,
    "minCredits": 6,
    "maxCredits": 15,
    "minCourses": 2,
    "maxCourses": 6,
    "credits": [2, 3, 2, 4, 1, 3, 3, 3, 3, 3, 3, 2, 3, 3, 3],
    "prerequisites": [[6, 0], [7, 5], [10, 4], [10, 5], [11, 10], [13, 8], [14, 8], [15, 9]]
}
```

Model

The MCSP3 model used for the competition is:

```java
class Bacp implements ProblemAPI {
    int nPeriods;
    int minCredits, maxCredits;
    int minCourses, maxCourses;
    int[] credits;
    int[][] prerequisites;

    private int[][] channelingTable(int c) {
        int[][] tuples = new int[nPeriods][nPeriods + 1];
        for (int p = 0; p < nPeriods; p++) {
            tuples[p][p] = credits[c];
            tuples[p][nPeriods] = p;
        }
        return tuples;
    }

    public void model() {
        int nCourses = credits.length, nPrerequisites = prerequisites.length;
        Var[] s = array("s", size(nCourses), dom(range(nPeriods)),
            "s[c] is the period (schedule) for course c");
        Var[] co = array("co", size(nPeriods), dom(range(minCourses, maxCourses + 1)),
            "co[p] is the number of courses at period p");
        Var[] cr = array("cr", size(nPeriods), dom(range(minCredits, maxCredits + 1)),
            "cr[p] is the number of credits at period p");
        Var[][] cp = array("cp", size(nCourses, nPeriods), (c, p) -> dom(0, credits[c]),
            "cp[c][p] is 0 if course c is not planned at period p, the number of credits for c otherwise");

       forall(range(nCourses), c -> extension(vars(cp[c], s[c]), channelingTable(c)))
            .note("channeling between arrays cp and s");
        forall(range(nPeriods), p -> count(s, takingValue(p), EQ, co[p]))
            .note("counting the number of courses in each period");
        forall(range(nPeriods), p -> sum(columnOf(cp, p), EQ, cr[p]))
            .note("counting the number of credits in each period");
    }
}
```
CHAPTER 2. PROBLEMS AND MODELS

This model involves 4 arrays of variables and 4 types of constraints: extension, count, sum and intension (primitive lessThan). Actually, two series 'm1' and 'm2' of 12 instances each have been selected. The series 'm1' corresponds to the model described above whereas 'm2' (obtained after some reformulations) is compatible with the restrictions imposed for the mini-track. Decision variables are indicated, except for the instances (XCSP3 files) whose name contains 'nodv' (no decision variables).

2.3 BIBD

This is Problem 028 on CSPLib, called Balanced Incomplete Block Designs (BIBD).

Description (from Steven Prestwich on CSPLib)

"BIBD generation is described in most standard textbooks on combinatorics. A BIBD is defined as an arrangement of \( v \) distinct objects into \( b \) blocks such that each block contains exactly \( k \) distinct objects, each object occurs in exactly \( r \) different blocks, and every two distinct objects occur together in exactly \( \lambda \) blocks. Another way of defining a BIBD is in terms of its incidence matrix, which is a \( v \) by \( b \) binary matrix with exactly \( r \) ones per row, \( k \) ones per column, and with a scalar product of \( \lambda \) between any pair of distinct rows. A BIBD is therefore specified by its parameters \((v, b, r, k, \lambda)\)"

Data

As an illustration of data specifying an instance of this problem, we have \((v = 7, b = 7, r = 3, k = 3, \lambda = 1)\).

Model

The MCSP3 model used for the competition is:
This model involves 1 array of variables and 2 types of constraints: \texttt{sum} and \texttt{lexMatrix}, the latter being used to break some variable symmetries. Two series 'sum' and 'sc' of 6 instances each have been selected. The series 'sum' corresponds to the model variant described above whereas 'sc' is a model variant obtained by introducing auxiliary variables.

\section*{2.4 Car Sequencing}

This is Problem 001 on CSPLib.

\textbf{Description} (from Barbara Smith on CSPLib)

“A number of cars are to be produced; they are not identical, because different options are available as variants on the basic model. The assembly line has different stations which install the various options (air-conditioning, sunroof, etc.). These stations have been designed to handle at most a certain percentage of the cars passing along the assembly line. Furthermore, the cars requiring a certain option must not be bunched together, otherwise the station will not be able to cope. Consequently, the cars must be arranged in a sequence so that the capacity of each station is never exceeded. For instance, if a particular station can only cope with at most half of the cars passing along the line, the sequence must be built so that at most 1 car in any 2 requires that option.”

\textbf{Data}

As an illustration of data specifying an instance of this problem, we have:

\begin{verbatim}
{
  "carClasses": [
    { "demand": 1, "options": [1, 0, 1, 1, 0] },
    { "demand": 1, "options": [0, 0, 0, 1, 0] },
    { "demand": 2, "options": [0, 1, 0, 0, 1] },
    { "demand": 2, "options": [0, 1, 0, 1, 0] },
    { "demand": 2, "options": [1, 0, 0, 0, 1] },
    { "demand": 2, "options": [1, 1, 0, 0, 0] }
  ],
  "optionLimits": [
    { "num": 1, "den": 2 },
    { "num": 2, "den": 3 },
    { "num": 1, "den": 3 },
    { "num": 2, "den": 5 },
    { "num": 1, "den": 5 }
  ]
}
\end{verbatim}

\textbf{Model}

The MCSP3 model used for the competition is:
class CarSequencing implements ProblemAPI {
    CarClass[] classes;
    OptionLimit[] limits;

    class CarClass {
        int demand;
        int[] options;
    }

class OptionLimit {
    int num;
    int den;
}

private Table channelingTable() {
    Table table = table();
    for (int i = 0; i < classes.length; i++)
        table.add(i, classes[i].options); // indexing car class options
    return table;
}

public void model() {
    int[] demands = valuesFrom(classes, cla -> cla.demand);
    int nCars = sumOf(demands), nOptions = limits.length, nClasses = classes.length;
    Range allClasses = range(nClasses);
    Var[] c = array("c", size(nCars), dom(allClasses),
        "c[i] is the class of the ith assembled car");
    Var[][] o = array("o", size(nCars, nOptions), dom(0, 1),
        "o[i][k] is 1 if the ith assembled car has option k");

cardinality(c, allClasses, occurExactly(demands))
    .note("building the right numbers of cars per class");

    forall(range(nCars), i ->
        extension(vars(c[i], o[i]), channelingTable())))
    .note("linking cars and options");

    forall(range(nOptions).range(nCars), (k, i) -> {
        if (i <= nCars - limits[k].den) {
            Var[] scp = select(columnOf(o, k), range(i, i + limits[k].den));
            sum(scp, LE, limits[k].num);
        }
    }).note("constraints about option frequencies");

    forall(range(nOptions).range(nCars), (k, i) -> {
        // i stands for the number of blocks set to the maximal capacity
        int nOptionOccurrences = sumOf(valuesFrom(classes, cla -> cla.options[k] * cla.demand));
        int nOptionsRemainingToSet = nOptionOccurrences - i * limits[k].num;
        int nOptionsPossibleToSet = nCars - i * limits[k].den;
        if (nOptionsRemainingToSet > 0 && nOptionsPossibleToSet > 0) {
            Var[] scp = select(columnOf(o, k), range(nOptionsPossibleToSet));
            sum(scp, GE, nOptionsRemainingToSet);
        }
    }).tag(REDUndant_CONSTRAints);
}
2.5. COLOURED QUEENS

2.5 Coloured Queens

Description

The queens graph is a graph with $n \times n$ nodes corresponding to the squares of a chessboard. There is an edge between nodes iff they are on the same row, column, or diagonal, i.e., if two queens on those squares would attack each other. The coloring problem is to color the queens graph with $n$ colors. See [13].

Data

As an illustration of data specifying an instance of this problem, we simply have $n = 8$.

Model

The MCSP3 model used for the competition is:

```java
class ColouredQueens implements ProblemAPI {
    int n;

    public void model() {
        Var[][] x = array("x", size(n, n), dom(range(n)),
            "x[i][j] is the color at row i and column j");
        Var[][] dn = diagonalsDown(x), up = diagonalsUp(x); // precomputing scopes
        allDifferentMatrix(x)
            .note("different colors on rows and columns");
        forall(range(dn.length), i -> allDifferent(dn[i])
            .note("different colors on downward diagonals");)
        forall(range(up.length), i -> allDifferent(up[i])
            .note("different colors on upward diagonals");)
    }
}
```

This model only involves 1 array of variables and 2 types of constraints: `allDifferent` and `allDifferentMatrix`. A series of 12 instances has been selected for the competition.

2.6 Crosswords (Satisfaction)

This problem has already been used in previous XCSP competitions, because it notably permits to compare filtering algorithms on large table constraints.

Description

“Given a grid with imposed black cells (spots) and a dictionary, the problem is to fulfill the grid with the words contained in the dictionary.”

Data

As an illustration of data specifying an instance of this problem, we have:

```json
{
    "spots": [[0,1,0,0,0], [0,0,0,0,0], [0,0,1,0,0], [0,0,0,0,0], [0,0,0,0,1]],
    "dictFileName": "ogd"
}
```
Model
The MCSP3 model used for the competition is:

class Crossword implements ProblemAPI {
    int[][] spots;
    String dictFileName;

    private Map<Integer, List<int[]>> loadWords() {
        Map<Integer, List<int[]>> words = new HashMap<>();
        readFileLines(dictFileName).forEach(w ->
            words.computeIfAbsent(w.length(), k -> new ArrayList<>()).add(Utilities.wordAsIntArray(w))
        );
        return words;
    }

    private class Hole {
        int row, col, size;
        boolean horizontal;

        Hole(int row, int col, int size, boolean horizontal) {
            this.row = horizontal ? row : col;
            this.col = horizontal ? col : row;
            this.size = size;
            this.horizontal = horizontal;
        }

        Var[] scope(Var[][] x) {
            return variablesFrom(range(size), i -> horizontal ? x[row][col + i] : x[row + i][col]);
        }
    }

    private List<Hole> findHoles(int[][] t, boolean untransposed) {
        int nRows = t.length, nCols = t[0].length;
        List<Hole> list = new ArrayList<>();
        for (int i = 0; i < nRows; i++) {
            int start = -1;
            for (int j = 0; j < nCols; j++)
                if (t[i][j] == 1) { // if spot (black cell)
                    if (start != -1 && j - start >= 2)
                        list.add(new Hole(i, start, j - start, untransposed));
                    start = -1;
                } else {
                    if (start == -1)
                        start = j;
                    else if (j == nCols - 1 && nCols - start >= 2)
                        list.add(new Hole(i, start, nCols - start, untransposed));
                }
        }
        return list;
    }

    private Hole[] findHoles() {
        List<Hole> list = findHoles(spots, true);
        list.addAll(findHoles(transpose(spots), false));
        return list.toArray(new Hole[0]);
    }

    public void model() {
        Map<Integer, List<int[]>> words = loadWords();
        Hole[] holes = findHoles();
        int nRows = spots.length, nCols = spots[0].length, nHoles = holes.length;

        Var[][] x = array("x", size(nRows, nCols), (i, j) -> dom(range(26)).when(spots[i][j] == 0),
            "x[i][j] is the letter, number from 0 to 25, at row i and column j (when no spot)");
    }
}
This satisfaction problem only involves 1 array of variables and 1 type of constraints: `extension` (ordinary table constraints). For clarity, we use an auxiliary class `Hole`. A series of 13 instances, with only blank grids, has been selected for the competition.

### 2.7 Crosswords (Optimization)

This problem is the subject of a regular Romanian challenge, and has been studied in XX.

**Description**

“Given a main dictionary containing ordinary words, and a second dictionary containing thematic words, the objective is to fill up a grid with words of both dictionary. Each word from the thematic word has a value (benefit) equal to its length. The objective is to maximize the overall value. The problem is difficult because black cells are not imposed, i.e., can be put anywhere in the grid (but no adjacency of black cells is authorized).”

**Data**

As an illustration of data specifying an instance of this problem, we have:

```json
{
  "n": 10,
  "nMaxWords": 5,
  "mainDict": "mainDictRomanian.txt",
  "thematicDict": "thematicDictRomanian2017.txt"
}
```

**Model**

The MCSP3 model used for the competition is:

```java
class CrosswordDesign implements ProblemAPI {
    int n; // size of the grid (number of rows and number of columns)
    int nMaxWords; // maximum number of words that can be put on a same row or column
    String mainDict, thematicDict;

    String[] words; // words of the two merged dictionaries

    int[] wordsPoints; // value of each word

    private void loadWords() {
        words = Stream.concat(readFileLines(mainDict), readFileLines(thematicDict))).toArray(String[]::new);
        wordsPoints = new int[words.length];
        List<String> list = Arrays.asList(words);
        readFileLines(thematicDict).forEach(w -> {
            int pos = list.indexOf(w);
            if (pos != -1)
                wordsPoints[pos] = w.length(); // thematic words have some value (their lengths)
        });
    }
}
```
private Table shortTable(int k)
{
  boolean lastWord = k == nMaxWords - 1;
  List<int[]> list = new ArrayList<>();
  if (k != 0)
    list.addAll(range(n + 4).map(i -> i == 0 || i == 1 || i == 3 ? -1 : i == 2 ? 0 : STAR));
  int[] possiblePositions = k == 0 ? vals(0, 1) : vals(range(2 * k, n));
  for (int p : possiblePositions)
    for (int i = 0; i < words.length; i++)
      if (bp = p + words[i].length(); // position of the black point, just after the word
          if (bp <= n - 1) {
            int[] tuple = new int[n + 4]; // position of the black point, just after the word
            tuple[0] = p;
            tuple[1] = i;
            tuple[2] = wordsPoints[i];
            if (lastWord && bp < n - 2)
              continue;
            for (int j = 4; j < tuple.length; j++)
              if (j - 4 == p - 1 || j - 4 == bp)
                tuple[j] = 26; // black points
              else if (p <= j - 4 && j - 4 < p + words[i].length())
                tuple[j] = words[i].charAt(j - 4 - p) - 97;
              else
                tuple[j] = STAR;
            list.add(tuple);
          }
      }
  return table().add(list);
}

private int[] positionValues(int k) {
  return k == 0 ? vals(0, 1) : k == nMaxWords ? vals(-1) : vals(range(-1, n));
}

public void model() {
  loadWords();

  int nWords = words.length;
  Var[][] x = array("x", size(n, n), dom(range(27)),
      "x[i][j] is the (number for) letter at row i and col j; 26 stands for a black point");
  Var[][] r = array("r", size(n, nMaxWords), dom(range(-1, nWords)),
      "r[i][k] is the (index of) kth word at row i; -1 means no word");
  Var[][] c = array("c", size(n, nMaxWords), dom(range(-1, nWords)),
      "c[j][k] is the (index of) kth word at col j; -1 means no word");
  Var[][] pr = array("pr", size(n, nMaxWords + 1), (i, k) -> dom(positionValues(k)),
      "pr[i][k] is the position (index of col) of the kth word at row i; -1 means no word");
  Var[][] pc = array("pc", size(n, nMaxWords + 1), (j, k) -> dom(positionValues(k)),
      "pc[j][k] is the position (index of row) of the kth word at col j; -1 means no word");
  Var[][] br = array("br", size(n, nMaxWords), dom(range(n + 1)),
      "br[i][k] is the benefit of the kth word at row i");
  Var[][] bc = array("bc", size(n, nMaxWords), dom(range(n + 1)),
      "bc[j][k] is the benefit of the kth word at col j");

  forall(range(n).range(nMaxWords), (i, k) ->
    extension(vars(pr[i][k], r[i][k], br[i][k], pr[i][k+1], x[i]), shortTable(k)))
    .note("putting words on rows");

  forall(range(n).range(nMaxWords), (j, k) ->
    extension(vars(pc[j][k], c[j][k], bc[j][k], pc[j][k+1], columnOf(x, j)), shortTable(k)))
    .note("putting words on columns");
This optimization problem involves 7 arrays of variables and simply the constraint `extension`. However, do note that such constraints are built with large short tables (i.e., tables involving '*', denoted by STAR in the code). The auxiliary methods `loadWords()` and `shortTable()` are respectively useful for loading the dictionaries and computing the short tables. A series of 13 instances has been selected for the competition.

### 2.8 Dubois

This problem has been conceived by Olivier Dubois, and submitted to the second DIMACS Implementation Challenge. Dubois's generator produces contradictory 3-SAT instances that seem very difficult to be solved by any general method.

#### Description

“Given an integer $n$, called the degree, Dubois's process allows us to construct a 3-SAT contradictory instance with $3 \times n$ variables and $2 \times n$ clauses, each of them having 3 literals.”

#### Data

As an illustration of data specifying an instance of this problem, we simply have $n = 10$.

#### Model

The MCSP3 model used for the competition is:

```java
class Dubois implements ProblemAPI {
  int n;

  public void model() {
    Table table1 = table("(0,0,1)(0,1,0)(1,0,0)(1,1,1)"), table2 = table("(0,0,0)(0,1,1)(1,0,1)(1,1,0)");
    Var[] x = array("x", size(3 * n), dom(0, 1))
      .note("x[i] is the Boolean value (0/1) of the ith variable of Dubois's sequence");
    extension(vars(x[2*n - 2], x[2*n - 1], x[0]), table1);
    forAll(range(n - 2), i -> extension(vars(x[i], x[2*n + i], x[i + 1]), table1));
    extension(vars(x[2 * n - 2], x[2 * n - 1], x[2 * n - 3]), table2);
  }
}
```

This model involves 1 array of variables and 1 type of constraints: `extension`. A series of 12 instances has been selected for the competition.

### 2.9 Eternity

Eternity II is a famous edge-matching puzzle, released in July 2007 by TOMY, with a 2 million dollars prize for the first submitted solution. See, for example, [2]. Here, we are interested
in instances derived from the original problem by the BeCool team of the UCL ("Université Catholique de Louvain") who proposed them for the competition.

**Description**

"On a board of size $n \times m$, you have to put square tiles (pieces) that are described by four colors (one for each direction: top, right, bottom and left). All adjacent tiles on the board must have matching colors along their common edge. All edges must have color '0' on the border of the board."

**Data**

As an illustration of data specifying an instance of this problem, we have:

```json
{
  "n": 3,
  "m": 3,
  "pieces": [[0,0,1,1], [0,0,1,2], [0,0,2,1], [0,0,2,2], [0,1,3,2], [0,1,4,1],
              [0,2,3,1], [0,2,4,2], [3,3,4,4]]
}
```

**Model**

The MCSP3 model used for the competition is:

```java
class Eternity implements ProblemAPI {
    int n, m;
    int[][] pieces;

    private Table piecesTable() {
        Table table = table();
        for (int i = 0; i < n * m; i++)
            for (int r = 0; r < 4; r++) // handling rotation
                table.add(i, pieces[i][r % 4], pieces[i][(r+1) % 4], pieces[i][(r+2) % 4], pieces[i][(r+3) % 4]);
        return table;
    }

    public void model() {
        int maxValue = maxOf(valuesIn(pieces)); // max possible value on pieces

        Var[][] id = array("id", size(n, m), dom(range(n * m)),
                          "id[i][j] is the id of the piece at row i and column j");
        Var[][] top = array("top", size(n, m), dom(range(0, maxValue)),
                           "top[i][j] is the value at the top of the piece put at row i and column j");
        Var[][] left = array("left", size(n, m), dom(range(0, maxValue)),
                             "left[i][j] is the value at the left of the piece put at row i and column j");
        Var[][] bot = array("bot", size(m), dom(range(0, maxValue)),
                            "bot[j] is the value at the bottom of the piece put at the bottommost row and column j");
        Var[] right = array("right", size(n), dom(range(0, maxValue)),
                            "right[i] is the value at the right of the piece put at the row i and the rightmost column");

        allDifferent(id)
            .note("all pieces must be placed (only once)");

        forall(range(n).range(m), (i, j) -> {
            Var lr = j < m - 1 ? left[i][j + 1] : right[j],
                lb = i < n - 1 ? top[i + 1][j] : bot[j];
            extension(vars(id[i][j], top[i][j], lr, lb, left[i][j], piecesTable()));
        }).note("pieces must be valid (i.e. correspond to those given initially, possibly after rotation)");
}
```
2.10. FAPP

This model involves 5 arrays of variables and 3 types of constraints: \texttt{allDifferent}, \texttt{extension} and \texttt{intension} (primitive \texttt{equal}). Note that we could have stored and reused the table, instead of creating it systematically. A series of 15 instances has been selected for the competition.

2.10 FAPP

The frequency assignment problem with polarization constraints (FAPP) is an optimization problem\footnote{This is an extended subject of the CALMA European project} that was part of the ROADEF’2001 challenge\footnote{See \url{http://uma.ensta.fr/conf/roadef-2001-challenge/}}. In this problem, there are constraints concerning frequencies and polarizations of radio links. Progressive relaxation of these constraints is explored: the relaxation level is between 0 (no relaxation) and 10 (the maximum relaxation). Whereas we used simplified CSP instances of this problem in previous XCSP competitions, do note here that we have considered the original COP instances.

Description

The description is rather complex. Hence, we refer the reader to:


Data

As an illustration of data specifying an instance of this problem, we have:

```json
{
    "domains": {
        "0": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
        "1": [25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40],
        "2": [55, 56, 57, 63, 64, 65],
    },
    "routes": [
        { "domain": 1, "polarization": -1 },
        { "domain": 1, "polarization": 0 },
        ...
    ],
    "hards": [
        { "route1": 1, "route2": 2, "frequency": true, "equality": true, "gap": 36 },
        { "route1": 0, "route2": 1, "frequency": true, "equality": true, "gap": 0 },
        ...
    ],
    "softs": [
        { "route1": 0, "route2": 2, "eqRelaxations": [46, 44, 42, 40, 40, 35, 35, 35, 30, 20],
        { "route1": 0, "route2": 8, "eqRelaxations": [30, 29, 28, 27, 26, 25, 24, 23, 21, 20],
            "neRelaxations": [29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19] },
```
CHAPTER 2. PROBLEMS AND MODELS

```java
{ "route1": 1, "route2": 3, "eqRelaxations": [33,32,30,29,28,27,22,17,12,7],
  "neRelaxations": [29,28,27,27,27,26,25,20,15,10,5] },
...
}
```

Model

The MCSP3 model used for the competition is:

```java
class Fapp implements ProblemAPI {

    Map<Integer, int[]> domains;
    Route[] routes;
    HardCtr[] hards;
    SoftCtr[] softs;

    class Route {
        int domain, polarization;

        int[] polarizationValues() {
            return polarization == 0 ? vals(0, 1) : polarization == 1 ? vals(1) : vals(0);
        }
    }

    class HardCtr {
        int route1, route2;
        boolean frequency, equality;
        int gap;
    }

    class SoftCtr {
        int route1, route2;
        int[] eqRelaxations;
        int[] neRelaxations;
    }

    private boolean softLink(int i, int j) {
        return firstFrom(softs, c -> c.route1 == i && c.route2 == j || c.route1 == j && c.route2 == i) != null;
    }

    private int[] distances(int i, int j) {
        int[] dom1 = domains.get(routes[i].domain), dom2 = domains.get(routes[j].domain);
        return IntStream.of(dom1).flatMap(f1 -> IntStream.of(dom2).map(f2 -> Math.abs(f1 - f2))).toArray();
    }

    private CtrEntity imperativeConstraint(Var[] f, Var[] p, HardCtr c) {
        int i = c.route1, j = c.route2;
        if (c.frequency) {
            if (c.gap == 0)
                return c.equality ? equal(f[i], f[j]) : different(f[i], f[j]);
            else
                return c.equality ? equal(dist(f[i], f[j]), c.gap) : different(dist(f[i], f[j]), c.gap);
        }
        return c.equality ? equal(p[i], p[j]) : different(p[i], p[j]);
    }

    private Table relaxTable(SoftCtr c) {
        Table table = table();
        Set<Integer> set = new HashSet<>();
        int i = c.route1, j = c.route2;
        for (int fi : domains.get(routes[i].domain))
```
for (int fj : domains.get(routes[j].domain)) {
    int dist = Math.abs(fi - fj);
    if (set.contains(dist))
        continue; // because already encountered
    for (int pol = 0; pol < 4; pol++) {
        int pi = pol < 2 ? 0 : 1;
        int pj = pol == 1 || pol == 3 ? 1 : 0;
        if (routes[i].polarization == 1 && pi == 0 || routes[j].polarization == 1 && pj == 0)
            continue;
        if (routes[i].polarization == -1 && pi == 1 || routes[j].polarization == -1 && pj == 1)
            continue;
        int[] t = pi == pj ? c.eqRelaxations : c.neRelaxations;
        for (int k = 0; k <= 11; k++) {
            if (k == 11 || dist >= t[k]) { // for k=11, we suppose t[k] = 0
                int sum = IntStream.range(0, k - 1).map(l -> dist >= t[l] ? 0 : 1).sum();
                table.add(dist, pi, pj, k, k == 0 || dist >= t[k - 1] ? 0 : 1, k <= 1 ? 0 : sum);
            }
        }
        set.add(dist);
    }
}
return table;

public void model() {
    int n = routes.length, nHards = hards == null ? 0 : hards.length, nSofts = softs.length;
    Var[] f = array("f", size(n), i -> dom(domains.get(routes[i].domain)),
        "f[i] is the frequency of the ith radio-link");
    Var[] p = array("p", size(n), i -> dom(routes[i].polarizationValues()),
        "p[i] is the polarization of the ith radio-link");
    Var[][] d = array("d", size(n, n), (i, j) -> dom(distances(i, j)).when(i < j && softLink(i,j)),
        "d[i][j] is the distance between the ith and the jth frequencies, for i < j when a soft link exists");
    Var[] v1 = array("v1", size(nSofts), dom(0, 1),
        "v1[q] is 1 iff the qth pair of radio constraints is violated when relaxing another level");
    Var[] v2 = array("v2", size(nSofts), dom(range(11)),
        "v2[q] is the number of times the qth pair of radio constraints is violated when relaxing more than one level");
    Var k = var("k", dom(range(12)),
        "k is the relaxation level to be optimized");

   forall(range(n).range(n), (i, j) -> {
        if (i < j && softLink(i,j))
            equal(d[i][j], dist(f[i], f[j]));
    }).note("computing intermediary distances");

   forall(range(nHards), q -> imperativeConstraint(f, p, hards[q]))
        .note("imperative constraints");

   forall(range(nSofts), q -> {
        int i = softs[q].route1, j = softs[q].route2;
        extension(vars(i < j ? d[i][j] : d[j][i], p[i], p[j], k, v1[q], v2[q]), relaxTable(softs[q]));
    }).note("relaxable radioelectric compatibility constraints");

    int[] coeffs = vals(10 * nSofts * nSofts, repeat(10 * nSofts, nSofts), repeat(1, nSofts));
    minimize(SUM, vars(k, v1, v2), weightedBy(coeffs))
        .note("minimizing sophisticated relaxation cost");
}

This model involves 5 arrays of variables (as well as the stand-alone variable k) and two types of constraints: extension and intension. A cache could be used for avoiding creating similar tables, and also for avoiding checking several times whether a given pair (i,j) is subject to a soft link. Two series 'm2s' and 'ext' of respectively 18 and 10 instances have been selected. The series 'm2s' corresponds to the model described above whereas the series 'ext' (obtained
2.11 FRB

This problem has been already used in previous XCSP competitions. Hence, we very succinctly introduce it.

**Description**

These instances are randomly generated using Model RB [23], while guaranteeing satisfiability.

This satisfaction problem only involves (ordinary) table constraints. A series of 16 instances has been selected. Using model RB, some forced binary CSP instances have been generated by choosing \( k = 2, \alpha = 0.8, r = 0.8 \) and \( n \) varying from 40 to 59. Each such instance is prefixed by frb-n.

2.12 Golomb Ruler

This is Problem 006 on CSPLib, called Golomb Ruler.

**Description** (from Peter van Beek on CSPLib)

"The problem is to find the ruler with the smallest length where we can put \( n \) marks such that the distance between any two pairs of marks is distinct."

**Data**

As an illustration of data specifying an instance of this problem, we simply have \( n = 8 \).

**Model**

The MCSP3 model used for the competition is:

```java
class GolombRuler implements ProblemAPI {
    int n;

    public void model() {
        int rulerLength = n * n + 1; // a trivial upper-bound
        Var[] x = array("x", size(n), dom(range(rulerLength)),
                        "x[i] is the position of the ith tick");
        Var[][] y = array("y", size(n, n), (i, j) -> dom(range(1, rulerLength)).when(i < j),
                        "y[i][j] is the distance between x[i] and x[j], for i < j");
        allDifferent(y)
            .note("all distances are different");
        forall(range(n), i -> forall(range(i + 1, n), j -> equal(x[j], add(x[i], y[i][j]))))
            .note("computing distances");
        minimize(x[n - 1])
            .note("minimizing the position of the rightmost tick");
    }
}
```
2.13. **GRACEFUL GRAPH**

This model involves 2 arrays of variables and 2 types of constraints: `allDifferent` and `intension (equal)`. A series of 10 + 3 instances has been chosen (\(n\) varying from 7 to 16). Decision variables are indicated, except for the 3 instances (XCSP3 files) whose name contains 'nodv' (no decision variables).

## 2.13 Graceful Graph

This is Problem 053 on CSPLib, called Graceful Graph. See, for example, [21].

**Description** (from Karen Petrie on CSPLib)

“A labelling \(f\) of the nodes of a graph with \(q\) edges is graceful if \(f\) assigns each node a unique label from \(\{0, 1, \ldots, q\}\) and when each edge \((x, y)\) is labelled with \(|f(x) - f(y)|\), the edge labels are all different. (Hence, the edge labels are a permutation of \(1, 2, \ldots, q\).)”

We focused on graphs of the form \(K_k \times P_p\) that consist of \(p\) copies of a clique \(K\) of size \(k\) with corresponding nodes of the cliques also forming the nodes of a path of length \(p\).

**Data**

As an illustration of data specifying an instance of this problem, we have \((k = 5, p = 2)\).

**Model**

The MCSP3 model used for the competition is:

```java
class GracefulGraph implements ProblemAPI {
    int k; // size of each clique K (number of nodes)
    int p; // size of each path P (or equivalently, number of cliques)

    public void model() {
        int nEdges = ((k * (k - 1)) * p) / 2 + k * (p - 1);

        Var[][] cn = array("cn", size(p, k), dom(range(nEdges + 1)),
        "cn[i][j] is the color of the jth node of the ith clique");
        Var[][] ce = array("ce", size(p, k, k), (i, j1, j2) -> dom(range(1, nEdges + 1)).when(j1 < j2),
        "ce[i][j1][j2] is the color of the edge (j1, j2) of the ith clique, for j1 < j2");
        Var[][] cp = array("cp", size(p - 1, k), dom(range(1, nEdges + 1)),
        "cp[i][j] is the color of the jth edge of the ith path");

        allDifferent(cn)
        .note("all nodes are colored differently");
        allDifferent(vars(ce, cp))
        .note("all edges are colored differently");

        block(() -> {
            forall(range(p).range(k), (i, j1) -> forall(range(j1 + 1, k), j2 ->
                equal(ce[i][j1][j2], dist(cn[i][j1], cn[i][j2])));
            forall(range(p - 1).range(k), (i, j) -> equal(cp[i][j], dist(cn[i][j], cn[i + 1][j])));
        }).note("computing colors of edges from colors of nodes");
    }
}
```

This model involves 3 arrays of variables and 2 types of constraints: `allDifferent` and `intension (equal)`. A series of 11 instances has been selected.
2.14 Graph Coloring

This well-known problem has been already used in previous XCSP competitions.

Description

“Given a graph \( G = (V, E) \), the objective is to find the minimum number of colors such that it is possible to color each node of \( G \) while ensuring that no two adjacent nodes share the same color.”

Model

The MCSP3 model used for the competition is:

```java
class Coloring implements ProblemAPI {
    int nNodes, nColors;
    int[][] edges;

    public void model() {
        int nEdges = edges.length;
        Var[] x = array("x", size(nNodes), dom(range(nColors)),
                        "x[i] is the color assigned to the ith node of the graph");
        forall(range(nEdges), i -> different(x[edges[i][0]], x[edges[i][1]]))
            .note("all adjacent nodes must be colored differently");
        minimize(MAXIMUM, x)
            .note("minimizing the maximum used color index (and, consequently, the number of colors"));
    }
}
```

This model only involves 1 array of variables and 1 type of constraint: intension (different). A series of 11 instances has been selected for the competition.

2.15 Haystacks

This problem, introduced by Marc Van Dongen, has been already used in previous XCSP competitions.

Description (from Marc Van Dongen)

“The problem instance of order \( p \) has \( p \times p \) variables with domain \( \{0,\ldots,p-1\} \). The constraint graph is highly regular, consisting of \( p \) clusters: one central cluster and \( p-1 \) outer clusters, each one being a \( p \)-clique. The instances are designed so that if the variables in the central cluster are instantiated, only one of the outer clusters contains an inconsistency: this cluster is the haystack. The task is to find the haystack and decide that it is unsatisfiable, thereby providing a proof that the current instantiation of the variables in the central cluster is inconsistent.”

A series of 10 instances has been selected for the competition.

2.16 Knapsack

This is Problem 133 on CSPLib, called Knapsack.
2.17. **LANGFORD**

**Description**

“Given a set of items, each with a weight and a value, determine which items to include in a collection so that the total weight is less than or equal to a given capacity and the total value is as large as possible.”

**Data**

As an illustration of data specifying an instance of this problem, we have:

```json
{
    "capacity": 10,
    "items": [
        { "weight": 2, "value": 54 },
        { "weight": 2, "value": 92 },
        { "weight": 1, "value": 62 },
        { "weight": 2, "value": 20 },
        { "weight": 2, "value": 55 }
    ]
}
```

**Model**

The MCSP3 model used for the competition is:

```java
class Knapsack implements ProblemAPI {
    int capacity;
    Item[] items;

class Item {
    int weight;
    int value;
}

public void model() {
    int[] weights = valuesFrom(items, item -> item.weight);
    int[] values = valuesFrom(items, item -> item.value);
    int nItems = items.length;

    Var[] x = array("x", size(nItems), dom(0, 1),
                    "x[i] is 1 iff the ith item is selected");

    sum(x, weightedBy(weights), LE, capacity).
    .note("the capacity of the knapsack must not be exceeded");

    maximize(SUM, x, weightedBy(values))
    .note("maximizing summed up value (benefit)");
}
```

This model only involves 1 array of variables and 1 type of constraint: **sum**. A series of 14 instances has been selected for the competition.

**2.17 Langford**

This is Problem 024 on CSPLib, called Langford’s number problem.
CHAPTER 2. PROBLEMS AND MODELS

Description (from Toby Walsh on CSPLib)

“Given two integers \( k \) and \( n \), the problem \( L(k, n) \) is to arrange \( k \) sets of numbers 1 to \( n \), so that each appearance of the number \( m \) is \( m \) numbers on from the last.”

Data

Here, we focus on the model proposed in [10] for \( k = 2 \). The MCSP3 model used for the competition is:

```java
class LangfordBin implements ProblemAPI {
    int n;
    public void model() {
        Var[] v = array("v", size(2 * n), dom(range(1, n + 1)),
            "v[i] is the ith value of the Langford vector");
        Var[] p = array("p", size(2 * n), dom(range(2 * n)),
            "p[j] is the first (resp., second) position of 1+j/2 in v if j is even (resp., odd)");
        forall(range(n), i -> element(v, at(p[2 * i]), takingValue(i + 1)))
            .note("computing the position of the 1st occurrence of i");
        forall(range(n), i -> element(v, at(p[2 * i + 1]), takingValue(i + 1)))
            .note("computing the position of the 2nd occurrence of i");
        forall(range(n), i -> equal(p[2 * i], add(i + 2, p[2 * i + 1])))
            .note("the distance between two occurrences of i must be respected");
    }
}
```

This model involves 2 arrays of variables and 2 types of constraints: element and intension (equal). A series of 11 instances has been generated for the competition, by varying \( n \) from 6 to 16.

2.18 Low Autocorrelation

This is Problem 005 on CSPLib, called Low Autocorrelation Binary Sequences.

Description (from Toby Walsh on CSPLib)

“The objective is to construct a binary sequence \( S_i \) of length \( n \) that minimizes the autocorrelations between bits. Each bit in the sequence takes the value +1 or −1. With non-periodic (or open) boundary conditions, the \( k \)th autocorrelation, \( C_k \) is defined to be \( \sum_{i=0}^{n-k-1} S_i \times S_{i+k} \). The aim is to minimize the sum of the squares of these autocorrelations, i.e., to minimize \( E = \sum_{k=1}^{n} C_k^2 \).

Data

As an illustration of data specifying an instance of this problem, we have \( n = 10 \).

Model

The MCSP3 model used for the competition is:

```java
class LowAutocorrelation implements ProblemAPI {
    int n;
    public void model() {
        Var[] x = array("x", size(n), dom(-1, 1),
            "x[i] is the ith value of the binary sequence");
        int[] c = array("c", size(n), dom(-1, 1),
            "c[i] is the autocorrelation for the sequence");
        forall(range(n), i -> element(x, at(i), takingValue(1)))
            .note("computing the value of the binary sequence");
        foreach(range(n), i) {
            int sum = 0;
            for(int j = 0; j < n; j++) {
                sum += x[i] * x[j];
            }
            c[i] = sum;
        }
        int E = 0;
        foreach(range(n), i) {
            E += c[i] * c[i];
        }
        .note("computing the autocorrelation");
        .note("computing the energy of the sequence");
    }
}
```
This model involves 4 arrays of variables and 2 types of constraints: sum and intension (equal). A series of 14 instances has been generated for the competition.

2.19 Magic Hexagon

This is Problem 023 on CSPLib, called Magic Hexagon.

Description

“A magic hexagon consists of the numbers 1 to 19 arranged in a hexagonal pattern such that all diagonals sum to 38.”

The description is given here for order $n = 3$ (the length of the first row of the hexagon) and starting value $s = 1$ (the first value of the sequence of numbers).

Model

The MCSP3 model used for the competition is:

```java
class MagicHexagon implements ProblemAPI {
  int n; // order
  int s; // start

  private Var[][] scopeForDiagonal(Var[][] x, int i, boolean right) {
    int d = x.length;
    int v1 = right ? Math.max(0, d / 2 - i) : Math.max(0, i - d / 2);
    Range r = range(d - Math.abs(d / 2 - i));
    return variablesFrom(r, j -> x[j + v1][i - Math.max(0, right ? v1 - j : j - v1)]);
  }

  public void model() {
    int gap = 3 * n * n - 3 * n + 1;
    int magic = sumOf(range(s, s + gap)) / (2 * n - 1);
    int d = n + n - 1; // longest diameter

    Var[][] x = array("x", size(d, d), (i, j) -> dom(range(s, s + gap)).when(j < d - Math.abs(d/2 - 1)),
      "x represents the hexagon; on row x[i], only the first n - |n/2 - i| cells are useful.");

    allDifferent(x)
      .note("all values must be different");
  }
}
```
This model involves 1 array of variables and 3 types of constraints: allDifferent, sum and intension (lessThan). The intensional constraints are here for breaking a few symmetries. A series of 11 instances has been generated for the competition.

2.20 Magic Square

This is Problem 019 on CSPLib, called Magic Square.

Description

“A magic square of order \( n \) is a \( n \) by \( n \) matrix containing the numbers 1 to \( n^2 \), where each row, column and main diagonal sum up to the same value.”

Data

As an illustration of data specifying an instance of this problem, we have:

```json
{
  "n": 9,
  "clues": [
    [0, 0, 0, 31, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 12, 0],
    ...
  ]
}
```

When there is no clue (pre-set value) at all, we have:

```json
{
  "n": 10,
  "clues": null
}
```

or, equivalently:

```json
{
  "n": 10,
  "clues": "null"
}
```
Model

The MCSP3 model used for the competition is:

class MagicSquare implements ProblemAPI {
    int n;
    int[][] clues;

    public void model() {
        int magic = n * (n * n + 1) / 2;

        Var[][] x = array("x", size(n, n), dom(range(1, n * n + 1)),
                       "x[i][j] is the value at row i and column j of the magic square");

        allDifferent(x).
        .note("all values must be different");

        forall(range(n), i -> sum(x[i], EQ, magic)).
        .note("all rows sum up to the magic value");

        forall(range(n), j -> sum(columnOf(x, j), EQ, magic)).
        .note("all columns sum up to the magic value");

        block(() -> {
            sum(diagonalDown(x), EQ, magic);
            sum(diagonalUp(x), EQ, magic);
        }).
        .note("the two (main) diagonals sum up to the magic value");

        instantiation(x, takingValues(clues), onlyOn((i, j) -> clues[i][j] != 0)).
        .tag(CLUES).
        .note("respecting specified clues (if any)");
    }
}

This model involves 1 array of variables and 3 types of constraints: allDifferent, sum and instantiation. A series of 13 instances has been generated, by varying n from 4 to 16. We didn’t use any clues (and so, the constraint instantiation is simply discarded at compilation).

2.21 Mario

This is a problem proposed by Amaury Ollagnier and Jean-Guillaume Fages at the 2013 MiniZinc Challenge.

Description (from Amaury Ollagnier and Jean-Guillaume Fages)

“This models a routing problem based on a little example of Mario’s day. Mario is an Italian Plumber and his work is mainly to find gold in the plumbing of all the houses of the neighborhood. Mario is moving in the city using his kart that has a specified amount of fuel. Mario starts his day of work from his house and always ends to his friend Luigi’s house to have the supper. The problem here is to plan the best path for Mario in order to earn the more money with the amount of fuel of his kart !

Data

As an illustration of data specifying an instance of this problem, we have:

{
    "marioHouse": 0,
    "luigiHouse": 1,
"fuelLimit": 2000,  
"houses": [  
  {  
    "fuelConsumption": [0, 221, 274, 808, 13, 677, 670, 943, 969, 13, 18, 217],  
    "gold": 0  
  },  
  {  
    "fuelConsumption": [0, 0, 702, 83, 813, 679, 906, 335, 529, 719, 528, 451],  
    "gold": 10  
  },  
  ...  
]  
}

Model

The MCSP3 model used for the competition is:

```java
class Mario implements ProblemAPI {
    int marioHouse, luigiHouse;
    int fuelLimit;
    House[] houses;

    class House {
        int[] fuel;
        int gold;
    }

    public void model() {
        int nHouses = houses.length;

        Var[] s = array("s", size(nHouses), dom(range(nHouses)),
                    "s[i] is the house succeeding to the ith house (itself if not part of the route)");
        Var[] f = array("f", size(nHouses), i -> dom(houses[i].fuel),
                        "f[i] is the fuel consumed at each step (from house i to its successor)");
        Var[] g = array("g", size(nHouses), i -> dom(0, houses[i].gold),
                        "g[i] is the gold earned at house i");

        forall(range(nHouses), i -> extension(vars(s[i], f[i]), indexing(houses[i].fuel)))
            .note("fuel consumption at each step");

        sum(f, LE, fuelLimit)
            .note("we cannot consume more than the available fuel");

        forall(range(nHouses), i -> {
            if (i != marioHouse && i != luigiHouse)
                equivalence(eq(s[i], i), eq(g[i], 0));
        }).note("gold earned at each house");

        circuit(s)
            .note("Mario must make a complete tour");

        equal(s[luigiHouse], marioHouse)
            .note("Mario house is just after Luigi house");

        maximize(SUM, g)
            .note("maximizing collected gold");
    }
}
```

This model involves 3 arrays of variables and 4 types of constraints: `circuit, sum, extension`
and intension (equivalence and equal). A series of 10 instances has been selected for the competition.

### 2.22 Mistery Shopper

This is Problem 004 on CSPLib, called Mistery Shopper.

**Description (from Jim Ho Man Lee on CSPLib)**

“A well-known cosmetic company wants to evaluate the performance of their sales people, who are stationed at the company's counters at various department stores in different geographical locations. For this purpose, the company has hired some secret agents to disguise themselves as shoppers to visit the sales people. The visits must be scheduled in such a way that each sales person must be visited by shoppers of different varieties and that the visits should be spaced out roughly evenly. Also, shoppers should visit sales people in different geographic locations.”

More details can be found on CSPLib.

**Data**

As an illustration of data specifying an instance of this problem, we have:

```java
{
    "visitorGroups" : [4, 4, 4],
    "visiteeGroups" : [3, 2, 4]
}
```

**Model**

The MCSP3 model used for the competition is:

```java
class MisteryShopper implements ProblemAPI {
    int[] visitorGroups; // visitorGroups[i] gives the size of the ith visitor group
    int[] visiteeGroups; // visiteeGroups[i] gives the size of the ith visitee group

    private Table numberPer(int[] t) { // numbering persons over all groups (sizes) of t
        Table table = table();
        for (int cnt = 0, i = 0; i < t.length; i++)
            for (int j = 0; j < t[i]; j++)
                table.add(i, cnt++);
        return table;
    }

    public void model() {
        int nVisitors = sumOf(visitorGroups), nVisitees = sumOf(visiteeGroups);
        int n = nVisitors, nDummyVisitees = nVisitors - nVisitees;
        if (nDummyVisitees > 0)
            visiteeGroups = addInt(visiteeGroups, nDummyVisitees); // dummy group added
        int nVisitorGroups = visitorGroups.length, nVisiteegroups = visiteeGroups.length;
        int nWeeks = nVisitorGroups;

        Var[][] vr = array("vr", size(n, nWeeks), dom(range(n)),
            "vr[i][w] is the visitor for the ith visitee at week w");
        Var[][] ve = array("ve", size(n, nWeeks), dom(range(n)),
            "ve[i][w] is the visitee for the ith visitor at week w");
        Var[][] gvr = array("gvr", size(n, nWeeks), dom(range(nVisitorGroups)),
            "gvr[i][w] is the group number of the visitor for the ith visitee at week w");
```
CHAPTER 2. PROBLEMS AND MODELS

"gvr[i][w] is the visitor group for the ith visitee at week w"

Var[](gve) = array("gve", size(n, nWeeks), dom(range(nVisiteeGroups)),
"gve[i][w] is the visitee group for the ith visitor at week w"
)

forall(range(nWeeks), w -> allDifferent(columnOf(vr, w)))
.note("each week, all visitors must be different")
forall(range(nWeeks), w -> allDifferent(columnOf(ve, w)))
.note("each week, all visitees must be different")
forall(range(n), i -> allDifferent(gvr[i]))
.note("the visitor groups must be different for each visitee")
forall(range(n), i -> allDifferent(gve[i]))
.note("the visitee groups must be different for each visitor")
forall(range(nWeeks), w -> channel(columnOf(vr, w), columnOf(ve, w)))
.note("channeling arrays vr and ve, each week")
forall(range(n).range(nWeeks), (i,w) -> extension(vars(gvr[i][w],vr[i][w]), numberPer(visitorGroups)))
.note("linking a visitor with its group")
forall(range(n).range(nWeeks), (i,w) -> extension(vars(gve[i][w],ve[i][w]), numberPer(visiteeGroups)))
.note("linking a visitee with its group")

block(() -> {
  lexMatrix(vr, INCREASING);
  if (nDummyVisitees > 0)
    forall(range(nWeeks), w -> strictlyIncreasing(select(columnOf(vr, w), range(nVisitees, n))));
}).tag(SYMMETRY_BREAKING);
}
}

This model involves 4 arrays of variables and 5 types of constraints: channel, allDifferent, lexMatrix, extension and intension (strictlyIncreasing). Note that we could have stored and reused tables instead of systematically building them. There is a block for breaking some symmetries. A series of 10 instances has been generated for the competition.

2.23 Nurse Rostering

This is a realistic employee shift scheduling Problem (see, for example, [16]).

Description

The description is rather complex. Hence, we refer the reader to:

Data

As an illustration of data specifying an instance of this problem, we have:

```json
{
  "nDays": 14,
  "shifts": [ { "id": "D", "length": 480, "forbiddenFollowingShifts": "null" } ],
  "staffs": [
    { "id": "A",
      "maxShifts": [14],
      "minTotalMinutes": 3360,"maxTotalMinutes": 4320,
      "minConsecutiveShifts": 2,"maxConsecutiveShifts": 5,
      "minConsecutiveDaysOff": 1, "daysOff": [0],
      "onRequests": [ ["A", 2], ["B", 3]]
  ]
}
```
2.23. NURSE ROSTERING

```
{
    "day": 2, "shift": "D", "weight": 2 },
{
    "day": 3, "shift": "D", "weight": 2}
],
"offRequests": "null"
},
...
],
"covers": [
[
    { "requirement": 3, "weightIfUnder": 100, "weightIfOver": 1 } ],
[
    { "requirement": 5, "weightIfUnder": 100, "weightIfOver": 1 } ],
...
]
}

Model

The MCSP3 model used for the competition is:

class NurseRostering implements ProblemAPI {
    int nDays;
    Shift[] shifts;
    Staff[] staffs;
    Cover[][] covers;

class Shift {
    String id = "off"; // value for the dummy shift
    int length;
    String[] forbiddenFollowingShifts;
}

class Request {
    int day;
    String shift;
    int weight;
}

class Staff {
    String id;
    int[] maxShifts;
    int minTotalMinutes, maxTotalMinutes;
    int minConsecutiveShifts, maxConsecutiveShifts;
    int minConsecutiveDaysOff, maxWeekends;
    int[] daysOff;
    Request[] onRequests, offRequests;
}

class Cover {
    int requirement, weightIfUnder, weightIfOver;

    int costFor(int i) {
        return i <= requirement ? (requirement - i) * weightIfUnder : (i - requirement) * weightIfOver;
    }
}

private Request onRequest(int person, int day) {
    return firstFrom(staffs[person].onRequests, request -> request.day == day);
}

private Request offRequest(int person, int day) {
    return firstFrom(staffs[person].offRequests, request -> request.day == day);
}
```
private int shiftPos(String s) {
    return firstFrom(range(shifts.length), i -> shifts[i].id.equals(s));
}

private int[] costsFor(int day, int shift) {
    int[] t = new int[staffs.length + 1];
    if (shift != shifts.length - 1) // if not 'off'
        for (int i = 0; i < t.length; i++)
            t[i] = covers[day][shift].costFor(i);
    return t;
}

private Automaton automatonMinConsec(int nShifts, int k, boolean forShifts) {
    Range rangeOff = range(nShifts - 1, nShifts); // a range with only one value ('off')
    Range rangeNotOff = range(nShifts - 1); // a range with all other values
    Range r1 = forShifts ? rangeOff : rangeNotOff, r2 = forShifts ? rangeNotOff : rangeOff;
    Transitions transitions = transitions();
    transitions.add("q0", r1, "q1").add("q0", r2, "q" + (k + 1)).add("q1", r1, "q" + (k + 1));
    for (int i = 1; i <= k; i++)
        transitions.add("q" + i, r2, "q" + (i + 1));
    transitions.add("q" + (k + 1), range(nShifts), "q" + (k + 1));
    return automaton("q0", transitions, finalState("q" + (k + 1)));
}

private Table rotationTable() {
    Table table = table(NEGATIVE); // a negative table (i.e., involving conflicts)
    for (Shift shift1 : shifts)
        if (shift1.forbiddenFollowingShifts != null)
            for (String shift2 : shift1.forbiddenFollowingShifts)
                table.add(shiftPos(shift1.id), shiftPos(shift2));
    return table;
}

private void buildDummyShift() {
    shifts = addObject(shifts, new Shift()); // we append first a dummy off shift
    for (Staff staff : staffs)
        staff.maxShifts = addInt(staff.maxShifts, nDays); // we append no limit (nDays) for the dummy shift
}

public void model() {
    buildDummyShift();
    int nShifts = shifts.length, nStaffs = staffs.length;
    int off = nShifts - 1; // value for 'off'
    Var[][] x = array("x", size(nDays, nStaffs), dom(range(nShifts)),
        "x[d][p] is the shift at day d for person p (one shift denoting 'off')");
    Var[][] ps = array("ps", size(nStaffs, nShifts), (p, s) -> dom(range(staffs[p].maxShifts[s] + 1)),
        "ps[p][s] is the number of days such that person p works with shift s");
    Var[][] da = array("ds", size(nDays, nShifts), dom(range(nShifts + 1)),
        "ds[d][s] is the number of persons working on day d with shift s");
    Var[][] wk = array("wk", size(nStaffs, nWeeks), dom(0, 1),
        "wk[p][w] is 1 iff the week-end w is worked by person p");
    Var[][] cn = array("cn", size(nStaffs, nDays), (p, d) -> onRequest(p, d) != null ? dom(0, onRequest(p, d).weight) : null,
        "cn[p][d] is the cost of not satisfying the on-request (if it exists) of person p on day d");
    Var[][] cf = array("cf", size(nStaffs, nDays), (p, d) ->
        offRequest(p, d) != null ? dom(0, offRequest(p, d).weight) : null,
        "cf[p][d] is the cost of not satisfying the off-request (if it exists) of person p on day d");
    Var[][] cc = array("cc", size(nDays, nShifts), (d, s) -> dom(costs[d, s]),
        "cc[d][s] is the cost of not satisfying cover for shift s on day d");

    instantiation(select(x, (d, p) -> contains(staffs[p].daysOff, d)), takingValue(off))
        .note("guaranteeing days off for staff");
    forall(range(nStaffs).range(nShifts), (p, s) -> exactly(columnOf(x, p), takingValue(s), ps[p][s]))
This model involves 7 arrays of variables and 7 types of constraints: regular, slide, count (exactly and atLeast), sum, instantiation, intension (implication and equivalence) and extension. Note how data are structured: we use 4 classes to describe them. You can
easily follow the structure of the automatas that are built when `automatonMinConsec()` is called. A series of 20 instances has been selected from http://www.schedulingbenchmarks.org.

### 2.24 Peacable Armies

This is Problem 110 on CSPLib, called Peaceably Co-existing Armies of Queens.

**Description** (from Ozgur Akgun on CSPLib)

“In the Armies of queens problem, we are required to place two equal-sized armies of black and white queens on a chessboard so that the white queens do not attack the black queens (and necessarily vice versa) and to find the maximum size of two such armies. Also see [20].”

**Data**

As an illustration of data specifying an instance of this problem, we have \( n = 10 \).

**Model**

The MCSP3 model(s) used for the competition is:

```java
class PeacableArmies implements ProblemAPI {
    int n; // order

    public void model() {
        if (modelVariant("m1")) {
            Var[][] b = array("b", size(n, n), dom(0, 1),
            "b[i][j] is 1 if a black queen is in the cell at row i and column j");
            Var[][] w = array("w", size(n, n), dom(0, 1),
            "w[i][j] is 1 if a white queen is in the cell at row i and column j");

            forall(range(n).range(n).range(n).range(n), (i1, j1, i2, j2) -> {
                if (i1 == i2 && j1 == j2)
                    lessEqual(add(b[i1][j1], w[i1][j1]), 1);
                else if (i1 < i2 || (i1 == i2 && j1 < j2))
                    if (i1 == i2 || j1 == j2 || Math.abs(i1 - i2) == Math.abs(j1 - j2)) {
                        lessEqual(add(b[i1][j1], w[i2][j2]), 1);
                        lessEqual(add(w[i1][j1], b[i2][j2]), 1);
                    }
                }).note("no two opponent queens can attack each other");

            int[] coeffs = range(n * n + 2).map(i -> i < n * n ? 1 : -1);
            sum(vars(b, w), weightedBy(coeffs), EQ, 0)
                .note("ensuring the same numbers of black and white queens");

            maximize(SUM, b)
                .note("maximizing the number of black queens (and consequently, the size of the armies)");
        }

        if (modelVariant("m2")) {
            Var[][] x = array("x", size(n, n), dom(0, 1, 2),
            "x[i][j] is 1 or 2 if a black or white queen is at row i and column j. It is 0 otherwise.");
            Var nb = var("nb", dom(range(n * n / 2)),
            "nb is the number of black queens");
            Var nw = var("nw", dom(range(n * n / 2)),
            "nw is the number of white queens");

            forall(range(n).range(n).range(n).range(n), (i1, j1, i2, j2) -> {
```
if (i1 < i2 || (i1 == i2 && j1 < j2))
    if (i1 == i2 || j1 == j2 || Math.abs(i1 - i2) == Math.abs(j1 - j2))
        different(add(x[i1][j1], x[i2][j2]), 3);
});

Note: “No two opponent queens can attack each other”;

count(vars(x), takingValue(1), EQ, nb).note("counting the number of black queens");
count(vars(x), takingValue(2), EQ, nw).note("counting the number of white queens");
equal(nb, nw).note("ensuring equal-sized armies");

maximize(nb)
    .note("maximizing the number of black queens (and consequently, the size of the armies)");

Following [20], two model variants, called ‘m1’ and ‘m2’ have been written. The first variant model involves 2 arrays of variables and 2 types of constraints: sum and intension (lessEqual). The second variant model involves 1 array of variables, 2 stand-alone variables and 2 types of constraints: count and intension (equal and different). A series of $2 \times 7$ instances has been generated for the competition.

2.25 Pizza Voucher

This is a problem introduced in the Minizinc challenge 2015 under the name “freepizza”.

Description

“You are given a list of pizzas (actually, their prices) to get, and a set of vouchers. Each voucher can be used to get pizzas for free. For example a voucher 2/1 (2 being the ‘pay’ part and 1 the ‘free’ part) indicates that you need to buy 2 pizzas to get another one free (whose price must be inferior). You want to optimally use the vouchers so as to get all the pizzas with minimal cost.”

Data

As an illustration of data specifying an instance of this problem, we have:

```json
{
    "pizzaPrices": [50, 60, 90, 70, 80, 100, 20, 30, 40, 10],
    "vouchers": [
        { "payPart": 1, "freePart": 2 },
        { "payPart": 2, "freePart": 3 },
        ...
    ]
}
```

Model

The MCSP3 model used for the competition is:

```java
class PizzaVoucher implements ProblemAPI {
    int[] pizzaPrices;
    Voucher[] vouchers;

class Voucher {
    int payPart;
```
CHAPTER 2. PROBLEMS AND MODELS

```java
int freePart;
}

public void model() {
    int nPizzas = pizzaPrices.length, nVouchers = vouchers.length;

    Var[] v = array("v", size(nPizzas), dom(rangeClosed(-nVouchers, nVouchers)),
    "v[i] is the voucher used for getting the ith pizza. 0 means that no voucher is used. A negative
    (resp., positive) value i means that the ith pizza contributes to the the pay (resp., free) part of voucher [i].");
    Var[] np = array("np", size(nVouchers), i -> dom(0, vouchers[i].payPart),
    "np[i] is the number of paid pizzas wrt the ith voucher");
    Var[] nf = array("nf", size(nVouchers), i -> dom(range(vouchers[i].freePart + 1)),
    "nf[i] is the number of free pizzas wrt the ith voucher");
    Var[] pp = array("pp", size(nPizzas), i -> dom(0, pizzaPrices[i]),
    "pp[i] is the price paid for the ith pizza");

    forall(range(nVouchers), i -> count(v, takingValue(-i - 1), EQ, np[i])).note("counting paid pizzas");
    forall(range(nVouchers), i -> count(v, takingValue(i + 1), EQ, nf[i])).note("counting free pizzas");
    forall(range(nVouchers), i -> equivalence(eq(nf[i], 0), ne(np[i], vouchers[i].payPart))).note("A voucher, if used, must contribute to have at least one free pizza.");
    forall(range(nPizzas).range(nPizzas), (i, j) -> {
        if (i != j && pizzaPrices[i] < pizzaPrices[j])
            disjunction(ge(v[i], v[j]), ne(v[i], neg(v[j])));
    }).note("A free pizza got with a voucher must be cheaper than any pizza paid wrt this voucher");

    minimize(SUM, pp).note("minimizing summed up price paid for pizzas");
    decisionVariables(v);
}
```

This model involves 4 arrays of variables and 2 types of constraints: count and intension (equivalence, implication and disjunction). A series of 10 original instances has been generated for the competition.

2.26 Pseudo-Boolean

This problem has been already used in previous XCSP competitions.

Description

Pseudo-Boolean problems generalize SAT problems by allowing linear constraints and, possibly, a linear objective function.

Data

As an illustration of data specifying an instance of this problem, we have:

```json
{
    "n": 144,
    "e": 704,
    "ctras": [
        { "coeffs": [1,1,1,1,1,1], "nums": [1,17,33,49,65,81], "op": "+", "limit": 1
        },
```
The MCSP3 model used for the competition is:

class PseudoBoolean implements ProblemAPI {
    int n, e;
    LinearCtr[] ctrs;
    LinearObj obj;

    class LinearCtr {
        int[] coeffs;
        int[] nums;
        String op;
        int limit;
    }

    class LinearObj {
        int[] coeffs;
        int[] nums;
    }

    public void model() {
        Var[] x = array("x", size(n), dom(0, 1), "x\[\] is the Boolean value (0/1) of the ith variable");

        forall(range(e), i -> {
            Var[] scp = variablesFrom(ctrs[i].nums, num -> x[num]);
            sum(scp, weightedBy(ctrs[i].coeffs), TypeConditionOperatorRel.valueFor(ctrs[i].op), ctrs[i].limit)
                .note("respecting each linear constraint");
        });

        if (obj != null) {
            Var[] scp = variablesFrom(obj.nums, num -> x[num]);
            minimize(SUM, scp, weightedBy(obj.coeffs))
                .note("minimizing the linear objective");
        }
    }
}

This problem involves 1 array of variables and 1 type of constraint: sum. Two series of 13 instances have been selected: one for CSP (model variant 'dec') and the other for COP (model 'opt').

### 2.27 Quadratic Assignment

The Quadratic Assignment Problem (QAP) is one of the fundamental combinatorial optimization problems in the branch of optimization. See, for example, QAPLIB.
CHAPTER 2. PROBLEMS AND MODELS

Description (from WikiPedia)

“There are a set of \( n \) facilities and a set of \( n \) locations. For each pair of locations, a distance is specified and for each pair of facilities a weight or flow is specified (e.g., the amount of supplies transported between the two facilities). The problem is to assign all facilities to different locations with the goal of minimizing the sum of the distances multiplied by the corresponding flows.”

Data

As an illustration of data specifying an instance of this problem, we have:

```
{
    "weights": [
        [0, 90, 10, 23, 43, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [90, 0, 0, 0, 0, 88, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        ...,
    ],
    "distances": [
        [0, 36, 54, 26, 59, 72, 9, 34, 79, 17, 46, 95],
        [36, 0, 73, 35, 90, 58, 30, 78, 35, 44, 79, 36],
        ...,
    ]
}
```

Model

The MCSP3 model used for the competition is:

```java
class QuadraticAssignment implements ProblemAPI {
    int[][] weights; // facility weights
    int[][] distances; // location distances

    private Table channelingTable() {
        Table table = table();
        for (int i = 0; i < distances.length; i++)
            for (int j = 0; j < distances.length; j++)
                if (i != j)
                    table.add(i, j, distances[i][j]);
        return table;
    }

    public void model() {
        int n = weights.length;
        Var[] x = array("x", size(n), dom(range(n)),
            "x[i] is the location assigned to the ith facility");
        Var[][] d = array("d", size(n, n), (i, j) -> dom(distances).when(i < j & weights[i][j] != 0),
            "d[i][j] is the distance between the locations assigned to the ith and jth facilities");
        allDifferent(x);
        .note("all locations must be different");
        forall(range(n).range(n), (i, j) -> {
            if (i < j & weights[i][j] > 0)
                extension(vars(x[i], x[j], d[i][j]), channelingTable());
        }).note("computing the distances");
        minimize(SUM, d, weightedBy(weights), onlyOn((i, j) -> i < j & weights[i][j] != 0))
    }
}
```
2.28 Quasigroup

This model involves 2 arrays of variables and 2 types of constraints: allDifferent and extension. A series of 19 instances has been selected for the competition.

2.28 Quasigroup

This is Problem 003 on CSPLib, called Quasigroup Existence.

Description (from Toby Walsh on CSPLib)

An order \( n \) quasigroup is a Latin square of size \( n \). That is, a \( n \times n \) multiplication table in which each element occurs once in every row and column. A quasigroup can be specified by a set and a binary multiplication operator, \( * \) defined over this set. Quasigroup existence problems determine the existence or non-existence of quasigroups of a given size with additional properties. For example:

- QG3: quasigroups for which \((a * b) * (b * a) = a\)
- QG7: quasigroups for which \((b * a) * b = a * (b * a)\)

For each of these problems, we may additionally demand that the quasigroup is idempotent. That is, \( a * a = a \) for every element \( a \).

Data

As an illustration of data specifying an instance of this problem, we have \( n = 6 \).

Model

The MCSP3 model(s) used for the competition is:

class QuasiGroup implements ProblemAPI {
  int n;

  public void model() {
    Var[][] x = array("x", size(n, n), dom(range(n)),
    "x[i][j] is the value at row i and column j of the quasigroup");

    allDifferentMatrix(x)
    .note("ensuring a Latin square");

    instantiation(diagonalDown(x), takingValues(range(n))).tag("idempotence");
    .note("enforcing x[i][i] = i");

    if (modelVariant("qg3")) {
      Var[][] y = array("y", size(n, n), dom(range(n * n)),
      "y[i][j] is the value at row i and column j of the quasigroup");

      forall(range(n).range(n), (i, j) -> {
        if (i != j) {
          element(vars(x), at(y[i][j]), takingValue(i));
          equal(y[i][j], add(mul(x[i][j], n), x[j][i]));
        }
      });
    }
    if (modelVariant("qg7")) {
      Var[][] y = array("y", size(n, n), dom(range(n)));
    }
  }
}
Two variants of the problem are described here. Both involve 2 arrays of variables and 3 types of constraints: allDifferentMatrix, instantiation and element. The second variant, 'qg7', also involves another type of constraint: intension (equal). Note the presence of the tag 'idempotence', which easily allows us to activate or deactivate the constraint instantiation at parsing time. A series of $2 \times 8$ instances has been generated, for problems QG3 and QG7.

### 2.29 RCPSP

This is Problem 061 on CSPLib, called Resource-Constrained Project Scheduling Problem (RCPSP). See also PSPLIB.

**Description** (from Peter Nightingale and Emir Demirovi on CSPLib)

“The resource-constrained project scheduling problem is a classical well-known problem in operations research. A number of activities are to be scheduled. Each activity has a duration and cannot be interrupted. There are a set of precedence relations between pairs of activities which state that the second activity must start after the first has finished. There are a set of renewable resources. Each resource has a maximum capacity and at any given time slot no more than this amount can be in use. Each activity has a demand (possibly zero) on each resource. The dummy source and sink activities have zero demand on all resources. The problem is usually stated as an optimisation problem where the makespan (i.e. the completion time of the sink activity) is minimized.”

**Data**

As an illustration of data specifying an instance of this problem, we have:

```json
{
  "horizon": 158,
  "resourceCapacities": [12, 13, 4, 12],
  "jobs": [
    { "duration": 0, "successors": [1, 2, 3], "requiredQuantities": [0, 0, 0, 0] },
    { "duration": 8, "successors": [5, 10, 14], "requiredQuantities": [4, 0, 0, 0] },
    ...
    { "duration": 0, "successors": [], "requiredQuantities": [0, 0, 0, 0] }
  ]
}
```

**Model**

The MCSP3 model used for the competition is:
class Rcpsp implements ProblemAPI {
    int horizon;
    int[] resourceCapacities;
    Job[] jobs;

class Job {
    int duration;
    int[] successors;
    int[] requiredQuantities;
}

public void model() {
    int nJobs = jobs.length;

    Var[] s = array("s", size(nJobs), i -> i == 0 ? dom(0) : dom(range(horizon)),
               "s[i] is the starting time of the ith job");

    forall(range(nJobs).range(nJobs), (i, j) -> {
        if (j < jobs[i].successors.length)
            lessEqual(add(s[i], jobs[i].duration), s[jobs[i].successors[j]]);
    }).note("precedence constraints");

    forall(range(resourceCapacities.length), j -> {
        int[] indexes = select(range(nJobs), i -> jobs[i].requiredQuantities[j] > 0);
        Var[] origins = variablesFrom(indexes, index -> s[index]);
        int[] lengths = valuesFrom(indexes, index -> jobs[index].duration);
        int[] heights = valuesFrom(indexes, index -> jobs[index].requiredQuantities[j]);
        cumulative(origins, lengths, heights, resourceCapacities[j]);
    }).note("resource constraints");

    minimize(s[nJobs - 1])
           .note("minimizing the makespan");
}
}

This model involves 1 array of variables and 2 types of constraints: cumulative and intension (lessEqual). A series of 16 instances has been selected for the competition.

### 2.30 RLFAP

When radio communication links are assigned the same or closely related frequencies, there is a potential for interference. Consider a radio communication network, defined by a set of radio links. The radio link frequency assignment problem [4] is to assign, from limited spectral resources, a frequency to each of these links in such a way that all the links may operate together without noticeable interference. Moreover, the assignment has to comply to certain regulations and physical constraints of the transmitters. Among all such assignments, one will naturally prefer those which make good use of the available spectrum, trying to save the spectral resources for a later extension of the network. Whereas we used simplified CSP instances of this problem in previous XCSP competitions, do note here that we have considered the original COP instances.

**Description**

The description is rather complex. Hence, we refer the reader to [4].

**Data**

As an illustration of data specifying an instance of this problem, we have:
The MCSP3 model used for the competition is:

```java
class Rlfap implements ProblemAPI {
    Map<Integer, int[]> domains;
    RlfapVar[] vars;
    RlfapCtr[] ctrs;
    int[] interferenceCosts;
    int[] mobilityCosts;

    class RlfapVar {
        int number;
        int domain;
        Integer value;
        Integer mobility;
    }

    class RlfapCtr {
        int x;
        int y;
        boolean equality;
        int limit;
        int weight;
    }

    private int index(int num) {
        return firstFrom(range(vars.length), i -> vars[i].number == num);
    }

    public void model() {
        int n = vars.length, e = ctrs.length;
        Var[] f = array("f", size(n), i -> dom(domains.get(vars[i].domain)));
        "f[i] is the frequency of the ith radio link"

        int[] indexes = select(range(n), i -> vars[i].value != null);
        Var[] fixedVars = variablesFrom(indexes, index -> mapVars.get(vars[index].number));
        "managing pre-assigned frequencies"

        instantiation(fixedVars, fixedVals).note("managing pre-assigned frequencies");
    }
}
```
Here, for simplicity, we only provide code for criteria SPAN and CARD. The model involves
1 array of variables and 2 types of constraints: instantiation and intension (equal and
greaterThan). Note that instead of the auxiliary method index(), we could have used a map.
The complete series of 25 instances, 11 CELAR (scen) and 14 GRAPH, has been selected for
the competition. Also, the good old series ‘scen11’ of CSP instances has been selected.

2.31 Social Golfers

This is Problem 010 on CSPLib, and called the Social Golfers Problem.

Description (from Warwick Harvey on CSPLib)

“The coordinator of a local golf club has come to you with the following problem.
In their club, there are 32 social golfers, each of whom play golf once a week, and
always in groups of 4. They would like you to come up with a schedule of play for
these golfers, to last as many weeks as possible, such that no golfer plays in the
same group as any other golfer on more than one occasion. The problem can easily
be generalized to that of scheduling m groups of n golfers over p weeks, such that no
golfer plays in the same group as any other golfer twice (i.e. maximum socialisation
is achieved).

Data

As an illustration of data specifying an instance of this problem, we have (nGroups = 8, groupSize =
4, nWeeks = 6):

Model

The MCSP3 model used for the competition is:

```java
class SocialGolfers implements ProblemAPI {
    int nGroups, groupSize, nWeeks;

    public void model() {
        int nPlayers = nGroups * groupSize;
        Range allGroups = range(nGroups);

        Var[][] x = array("x", size(nWeeks, nPlayers), dom(allGroups),
                        "x[w][p] is the group in which player p plays in week w");
```
CHAPTER 2. PROBLEMS AND MODELS

forall(range(nWeeks), w -> cardinality(x[w], allGroups, occursEachExactly(groupSize)));

 Особенности размера групп

forall(range(nWeeks).range(nWeeks).range(nPlayers).range(nPlayers), (w1, w2, p1, p2) -> {
    if (w1 < w2 && p1 < p2)
        disjunction(ne(x[w1][p1], x[w1][p2]), ne(x[w2][p1], x[w2][p2]));
}).(note("ensuring that two players don’t meet each other more than one time"));

block(() -> {
    instantiation(x[0], takingValues(range(nPlayers).map(p -> p / groupSize)));
    forall(range(groupSize), k -> instantiation(select(columnOf(x, k), w -> w > 0), takingValue(k)));
    lexMatrix(x, INCREASING);
}).tag(SYMMETRY_BREAKING);

This model involves 1 array of variables and 4 types of constraints: cardinality, lexMatrix, instantiation and intension (disjunction). Note the presence of a block for breaking some symmetries. A series of 12 instances has been selected for the competition.

2.32 Sports Scheduling

This is Problem 026 on CSPLib, called the Sports Tournament Scheduling.

Description (from Toby Walsh on CSPLib)

“The problem is to schedule a tournament of n teams over n/1 weeks, with each week divided into n/2 periods, and each period divided into two slots. The first team in each slot plays at home, whilst the second plays the first team away. A tournament must satisfy the following three constraints: every team plays once a week; every team plays at most twice in the same period over the tournament; every team plays every other team.”

Data

As an illustration of data specifying an instance of this problem, we have n = 10.

Model

The MCSP3 model used for the competition is:

class SportsScheduling implements ProblemAPI {
    int nTeams;

    private int matchNumber(int team1, int team2) {
        int nPossibleMatches = (nTeams - 1) * nTeams / 2;
        return nPossibleMatches - ((nTeams - team1) * (nTeams - team1 - 1)) / 2 + (team2 - team1 - 1);
    }

    private Table matches() {
        Table table = table();
        for (int team1 = 0; team1 < nTeams; team1++)
            for (int team2 = team1 + 1; team2 < nTeams; team2++)
                table.add(team1, team2, matchNumber(team1, team2));
        return table;
    }

    public void model() {

int nWeeks = nTeams - 1, nPeriods = nTeams / 2, nPossibleMatches = (nTeams - 1) * nTeams / 2;
Range allTeams = range(nTeams);

Var[1][1] h = array("h", size(nPeriods, nWeeks), dom(allTeams),
    "h[p][w] is the number of the home opponent");
Var[1][1] a = array("a", size(nPeriods, nWeeks), dom(allTeams),
    "a[p][w] is the number of the away opponent");
Var[1][1] m = array("m", size(nPeriods, nWeeks), dom(range(nPossibleMatches)),
    "m[p][w] is the number of the match");

allDifferent(m)
    .note("all matches are different (no team can play twice against another team)");
forall(range(nPeriods).range(nWeeks), (p, w) -> extension(vars(h[p][w], a[p][w], m[p][w]), matchs()))
    .note("linking variables through ternary table constraints");
forall(range(nWeeks), w -> allDifferent(vars(columnOf(h, w), columnOf(a, w))))
    .note("each week, all teams are different (each team plays each week");
forall(range(nPeriods), p -> cardinality(vars(h[p], a[p]), allTeams, occursEachBetween(1, 2)))
    .note("each team plays at most two times in each period");

block(() -> {
    instantiation(columnOf(m, 0), takingValues(range(nPeriods).map(p -> matchNumber(2 * p, 2 * p + 1))))
        .note("the first week is set : 0 vs 1, 2 vs 3, 4 vs 5, etc.");
    forall(range(nWeeks), w -> exactly1(columnOf(m, w), takingValue(matchNumber(0, w + 1))))
        .note("the match 0 versus t (with t strictly greater than 0) appears at week t − 1");
}).tag(SYMMETRY_BREAKING);

block(() -> {
    Var[1] hd = array("hd", size(nPeriods), dom(range(nTeams)),
        "hd[p] is the number of the home opponent for the dummy match of the period");
    Var[1] ad = array("ad", size(nPeriods), dom(range(nTeams)),
        "ad[p] is the number of the away opponent for the dummy match of the period");

    allDifferent(vars(hd, ad))
        .note("all teams are different in the dummy week");
    forall(range(nPeriods), p -> cardinality(vars(h[p], hd[p], ad[p], a[p]), allTeams, occursEachExactly(2)))
        .note("Each team plays two times in each period");
    forall(range(nPeriods), p -> lessThan(hd[p], ad[p]))
        .tag(SYMMETRY_BREAKING);
}).note("handling dummy week (variables and constraints") .tag("dummyWeek");
}

This model involves 3 + 2 arrays of variables and 6 types of constraints: cardinality, allDifferent, count (exactly1), instantiation, extension and intension (lessThan).
Note that we could have used a cache for the table built by matchs(). Also, the presence of the tag ‘dummyWeek’ allows us to easily activate or deactivate this part of the model, at parsing time. A series of 10 instances has been selected for the competition.

2.33 Steel Mill Slab

This is Problem 038 on CSPLib, called Steel Mill Slab Design.

Description (from Ian Miguel on CSPLib)

"Steel is produced by casting molten iron into slabs. A steel mill can produce a finite number of slab sizes. An order has two properties, a colour corresponding to the route required through the steel mill and a weight. Given input orders, the problem is to assign the orders to slabs, the number and size of which are also to be determined, such that the total weight of steel produced is minimized. This assignment is subject to two further constraints:
• Capacity constraints: The total weight of orders assigned to a slab cannot exceed the slab capacity.
• Colour constraints: Each slab can contain at most \( p \) of \( k \) total colours (\( p \) is usually 2).

The colour constraints arise because it is expensive to cut up slabs in order to send them to different parts of the mill.

Data
As an illustration of data specifying an instance of this problem, we have:

```

{  
  "slabCapacities": [5, 7, 9, 11, 15, 18],  
  "orders": [  
   { "size": 1, "color": 2 },  
   { "size": 3, "color": 1 },  
   { "size": 2, "color": 1 },  
   ...  
  ]
}
```

Model
The MCSP3 model(s) used for the competition is:

```java
class SteelMillSlab implements ProblemAPI {
  int[] slabCapacities;
  Order[] orders;

  class Order {
    int size;
    int color;
  }

  // Repartition of orders in groups according to colors
  private Stream<int[]> colRep(int[] allColors) {
    return IntStream.of(allColors).mapToObj(c -> range(orders.length).select(i -> orders[i].color == c));
  }

  public void model() {
    slabCapacities = singleValuesIn(0, slabCapacities); // distinct sorted capacities (including 0)
    int maxCapacity = slabCapacities[slabCapacities.length - 1];
    int[] possibleLosses = range(maxCapacity+1).map(i -> minOf(select(slabCapacities, v -> v >= i)) - i);
    int[] sizes = valuesFrom(orders, order -> order.size);
    int totalSize = sumOf(sizes);
    int[] allColors = singleValuesFrom(orders, order -> order.color);
    int nOrders = orders.length, nSlabs = orders.length, nColors = allColors.length;

    Var[] sb = array("sb", size(nOrders), dom(range(nSlabs)), "sb[o] is the slab used to produce order o");
    Var[] ld = array("ld", size(nSlabs), dom(range(maxCapacity + 1)), "ld[s] is the load of slab s");
    Var[] ls = array("ls", size(nSlabs), dom(possibleLosses), "ls[s] is the loss of slab s");

    if (modelVariant("m1")) {
     forall(range(nSlabs), s -> sum(treesFrom(sb, x -> eq(x, s)), weightedBy(sizes), EQ, ld[s]));
      note("computing (and checking) the load of each slab");
    }
}
```
2.34. STILL LIFE

Two model variants, 'm1' and 'm2', have been considered. The first model variant involves 3 arrays of variables and 4 types of constraints: sum (over trees), extension, ordered (decreasing) and intension (lessEqual). The second model variant involves 3 + 2 arrays of variables and 4 types of constraints: sum, extension, ordered (decreasing) and intension (equivalence, implication and lessEqual). Note that there is a redundant constraint and a block of symmetry-breaking constraints. The series 'm2s' corresponds to the model 'm2' without the redundant and symmetry-breaking constraints. A series of 6 + 6 + 5 instances has been selected for the main track. For the mini-track, instances from model 'm2s' have been slightly reformulated.

2.34 Still Life

This is Problem 032 on CSPLib, called Maximum density still life. This problem arises from the Game of Life, invented by John Horton Conway in the 1960s and popularized by Martin Gardner in his Scientific American columns.
Description (from Barbara Smith on CSPLib)

“Life is played on a squared board. Each square of the board is a cell, which at any time during the game is either alive or dead. A cell has eight neighbours. The configuration of live and dead cells at time $t$ leads to a new configuration at time $t + 1$ according to the rules of the game:

- if a cell has exactly three living neighbours at time $t$, it is alive at time $t + 1$
- if a cell has exactly two living neighbours at time $t$ it is in the same state at time $t + 1$ as it was at time $t$
- otherwise, the cell is dead at time $t + 1$

A stable pattern, or still-life, is not changed by these rules. Hence, every cell that has exactly three live neighbours is alive, and every cell that has fewer than two or more than three live neighbours is dead. What is the densest possible still-life, i.e. the pattern with the largest number of live cells, that can be fitted into the board?

Data

As an illustration of data specifying an instance of this problem, we have $n = 8$.

Model

The MCSP3 model used for the competition is:

```java
class StillLife implements ProblemAPI {
    int n;

    private Predicate<int[]> p = x -> {
        return (x[4] != 1 || s1 >= 2) && (x[4] != 1 || s1 <= 3) && (x[4] != 0 || s1 != 3)
            && (x[4] != 0 || s3 > 0 || x[9] >= 2);
    };

    public void model() {
        Var[][] x = array("x", size(n + 2, n + 2), dom(0, 1),
            "x[i][j] is 1 iff the cell at row i and column j is alive (note that there is a border"));
        Var[][] w = array("w", size(n + 2, n + 2), dom(0, 1, 2),
            "w[i][j] is the wastage for the cell at row i and column j");
        Var[] ws = array("ws", size(n + 2), dom(range(2 * (n + 2) * (n + 2) + 1)),
            "ws[i] is the wastage sum for cells at row i");
        Var z = var("z", dom(range(n * n + 1)),
            "z is the number of alive cells");

        block(() -> {
            instantiation(x[0], takingValue(0));
            instantiation(x[n + 1], takingValue(0));
            instantiation(columnOf(x, 0), takingValue(0));
            instantiation(columnOf(x, n + 1), takingValue(0));
        }).note("cells at the border are assumed to be dead");

        block(() -> {
            Table conflicts = table(NEGATIVE).add(1, 1, 1);
        });
    }
}
```
This model involves 3 arrays of variables, 1 stand-alone variable and 4 types of constraints: instantiation, extension, sum and intension (equal and greaterEqual). This model is in the spirit of the 'wastage' model from the Minizinc challenge 2012. Interestingly, note how we manage both Still life and wastage constraints with table constraints. To build them, we filter tuples from a Cartesian product by using a predicate (field \( p \) which is a lambda function not being considered as a piece of data by means of the annotation @NotData) for the competition. A series of 13 instances has been selected.

### 2.35 Strip Packing

This is the Two-Dimensional Strip Packing Problem (TDSP), as introduced, for example, in [11]. See also the OR-library.

#### Description

“In the Two-Dimensional Strip Packing Problem (TDSP), one has to pack a set of rectangular items into a rectangular strip.”

#### Data

As an illustration of data specifying an instance of this problem, we have:

```json
{
  "container": { "width": 20, "height": 20 },
  "rectangles": [
    { "width": 2, "height": 12 },
    { "width": 7, "height": 12 }
  ]
}
```
CHAPTER 2. PROBLEMS AND MODELS

Model

The MCSP3 model used for the competition is:

class StripPacking implements ProblemAPI {

    Rectangle container;
    Rectangle[] items;

    class Rectangle {
        int width;
        int height;
    }

    public void model() {
        int nItems = items.length;

        Var[] x = array("x", size(nItems), dom(range(container.width)),
        "x[i] is the x-coordinate of the ith rectangle");
        Var[] y = array("y", size(nItems), dom(range(container.height)),
        "y[i] is the y-coordinate of the ith rectangle");
        Var[] w = array("w", size(nItems), i -> dom(items[i].width, items[i].height),
        "w[i] is the width of the ith rectangle");
        Var[] h = array("h", size(nItems), i -> dom(items[i].width, items[i].height),
        "h[i] is the height of the ith rectangle");
        Var[] r = array("r", size(nItems), dom(0, 1),
        "r[i] is 1 iff the ith rectangle is rotated by 90 degrees");

        forall(range(nItems), i -> lessEqual(add(x[i], w[i]), container.width))
        .note("horizontal control");
        forall(range(nItems), i -> lessEqual(add(y[i], h[i]), container.height))
        .note("vertical control");
        forall(range(nItems), i -> {
            Table table = table();
            table.add(0, items[i].width, items[i].height).add(1, items[i].height, items[i].width);
            extension(vars(r[i], w[i], h[i]), table);
        }).note("managing rotation");
        noOverlap(transpose(x, y), transpose(w, h))
        .note("no overlapping between rectangles");
    }
}

This model involves 5 arrays of variables, and 3 types of constraints: noOverlap, extension and intension (lessEqual). A series of 12 instances has been selected for the competition.

2.36 Subgraph Isomorphism

Description

"In theoretical computer science, the subgraph isomorphism problem is a computational task in which two graphs G and H are given as input, and one must determine whether G contains a subgraph that is isomorphic to H."

Data

As an illustration of data specifying an instance of this problem, we have:
2.36. SUBGRAPH ISOMORPHISM

{ 
"nPattNodes": 180,  
"nTarNodes": 200, 
"patternEdges": [ [0,1], [0,3], [0,17], ... ], 
"targetEdges": [ [0,34], [0,65], [0,129], ... ] 
}

Model

The MCSP3 model used for the competition is:

```java
class Subisomorphism implements ProblemAPI {
    int nPattNodes, nTarNodes;
    int[][] patternEdges, targetEdges;

    private int[] selfLoops(int[][] edges) {
        return Stream.of(edges).filter(t -> t[0] == t[1]).mapToLong(t -> t[0]).toArray();
    }

    private int degree(int[][] edges, int node) {
        return (int) Stream.of(edges).filter(t -> t[0] == node || t[1] == node).count();
    }

    private Table bothWayTable() {
        Table table = table().add(targetEdges);
        table.add(Stream.of(targetEdges).map(t -> tuple(t[1], t[0]))); // reversed tuples
        return table;
    }

    public void model() {
        int[] pLoops = selfLoops(patternEdges);
        int[] tLoops = selfLoops(targetEdges);
        int[] pDegrees = range(nPattNodes).map(i -> degree(patternEdges, i));
        int[] tDegrees = range(nTarNodes).map(i -> degree(targetEdges, i));
        int l = pLoops.length, e = patternEdges.length;

        Var[] x = array("x", size(nPattNodes), dom(range(nTarNodes)), "x[i] is the node from the target graph to which the ith node of the pattern graph is mapped."");
        allDifferent(x)
            .note("ensuring injectivity");
        forall(range(l), i -> extension(x[pLoops[i]], tLoops))
            .note("being careful of self-loops");
        forall(range(e), i -> extension(vars(x[patternEdges[i][0]], x[patternEdges[i][1]]), bothWayTable()))
            .note("preserving edges");
        forall(range(nPattNodes), i -> {
            int[] conflicts = range(nTarNodes).select(j -> tDegrees[j] < pDegrees[i]);
            if (conflicts.length > 0)
                extension(x[i], conflicts, NEGATIVE);
        }).tag(REDUndANT_CONSTRAINTS);
    }
}
```

This model involves 1 array of variables and 2 types of constraints: allDifferent and extension. Note that we could have used a cache for the table built by bothWayTable(). There is a block with redundant unary constraints. A series of 11 instances has been selected for the competition.
2.37 Sum Coloring

Description

“In graph theory, a sum coloring of a graph is a labeling of its nodes by positive integers, with no two adjacent nodes having equal labels, that minimizes the sum of the labels.”

Data

As an illustration of data specifying an instance of this problem, we have:

```
{
    "nNodes": 30,
    "edges": [ [0,1], [0,3], [0,10], [0,12], ... ]
}
```

Model

The MCSP3 model used for the competition is:

```java
class SumColoring implements ProblemAPI {
    int nNodes;
    int[][] edges;

    public void model() {
        int nEdges = edges.length;

        Var[] c = array("c", size(nNodes), dom(range(nNodes)),
                        "c[i] is the color assigned to the ith node");

        forall(range(nEdges), i -> different(c[edges[i][0]], c[edges[i][1]]))
            .note("two adjacent nodes must be colored differently");

        minimize(SUM, c)
            .note("minimizing the sum of colors assigned to nodes");
    }
}
```

This model only involves 1 array of variables and 1 type of constraint: intension (different).

A series of 14 instances has been selected for the competition.

2.38 TAL

TAL is a problem of natural language processing.

Description (from work by Rémi Coletta and Jean-Philippe Prost)

The description is rather complex. Hence, we refer the reader to [12].

Data

As an illustration of data specifying an instance of this problem, we have:

```
{
    "maxArity": 7,
```
class Tal implements ProblemAPI {
    int maxArity;
    int maxHeight;
    int[] sentence;
    int[][][] grammar; // grammar[i] gives the grammar tuples of arity i
    String[] tokens;
    int[] costs;

    private void predicate(Var[][] l, Var[][] a, int i, int j, int[] lengths) {
        Range r = range(i != 0 && j == lengths[i] - 1 ? 1 : 0, Math.min(j + 1, maxArity));
        intension(xor(eq(l[i][j], 0), treesFrom(r, k -> ge(a[i + 1][j - k], k + 1))));
    }

    private Table tableFor(int vectorLength) {
        int arity = vectorLength + 2;
        Table table = table().add(range(arity).map(k -> k == arity - 1 ? 0 : STAR));
        for (int i = 0; i < vectorLength; i++)
            for (int j = 1; j < tokens.length + 1; j++) {
                int[] tuple = repeat(STAR, arity);
                tuple[0] = i;
                tuple[i + 1] = j;
                tuple[arity - 1] = j;
                table.add(tuple);
            }
        return table;
    }

    public void model() {
        int nWords = sentence.length, nLevels = nWords * 2, nTokens = tokens.length;
        int[] lengths = valuesFrom(range(nLevels), i -> i==0 ? nWords : nWords - (int) Math.floor((i+1)/2)+1);

        Var[][] c = array("C", size(nLevels, nWords), (i, j) -> (i == 0 || i % 2 == 1) && j < lengths[i] ?
            dom(costs), "c[i][j] is the cost of the jth word at the ith level");
        Var[][] l = array("L", size(nLevels, nWords), (i, j) -> j < lengths[i] ?
            dom(range(nTokens + 1)), "l[i][j] is the label of the jth word at the ith level");
        Var[][] a = array("a", size(nLevels, nWords), (i, j) -> i % 2 == 1 && j < lengths[i] ?
            dom(range(maxArity + 1)), "a[i][j] is the arity of the jth word at the ith level");
        Var[][] x = array("x", size(nLevels, nWords), (i, j) -> 0 < i && i % 2 == 0 && j < lengths[i] ?
            dom(range(lengths[i])), "x[i][j] is the index of the jth word at the ith level");
        Var[] s = array("s", size(nLevels - 2), i -> dom(range(lengths[i + 1])));

        forall(range(1, nLevels-1), i -> exactly(select(l[i][j], j -> j<lengths[i]), takingValue(0), s[i - 1]));
        forall(range(1, nLevels - 1, 2), i -> equal(s[i - 1], s[i]));

        forall(range(nWords), j -> equal(c[0][j], 0))
            .note("on row 0, costs are 0");
        forall(range(nWords), j -> equal(l[1][j], sentence[j]))
            .note("on row 0, the jth label is the jth word of the sentence");
        forall(range(1, nLevels), i -> greaterThan(l[i][0], 0))
            .note("on column 0, labels are 0");
    }
}
This model involves 5 arrays of variables and 3 types of constraints: count (exactly), extension and intension (equal, greaterThan, implication and equivalence). A series of 10 instances has been selected for the competition.

### 2.39 Template Design

This is Problem 002 on CSPLib, called Template Design. See also [19].

**Description (from Barbara Smith on CSPLib)**

“This problem arises from a colour printing firm which produces a variety of products from thin board, including cartons for human and animal food and magazine inserts. Food products, for example, are often marketed as a basic brand with several variations (typically flavours). Packaging for such variations usually has the same overall design, in particular the same size and shape, but differs in a small proportion of the text displayed and/or in colour. For instance, two variations of a cat food carton may differ only in that on one is printed Chicken Flavour on a blue background whereas the other has Rabbit Flavour printed on a green background. A typical order is for a variety of quantities of several design variations. Because each variation is identical in dimension, we know in advance exactly how many items can be printed on each mother sheet of board, whose dimensions are
2.39. TEMPLATE DESIGN

largely determined by the dimensions of the printing machinery. Each mother sheet is printed from a template, consisting of a thin aluminium sheet on which the design for several of the variations is etched. The problem is to decide, firstly, how many distinct templates to produce, and secondly, which variations, and how many copies of each, to include on each template.”

Data

As an illustration of data specifying an instance of this problem, we have:

```
{  
    "nSlots": 9,  
    "demands": [250, 255, 260, 500, 500, 800, 1100]  
}
```

Model

The MCSP3 model(s) used for the competition is:

```java
class TemplateDesign implements ProblemAPI {
    int nSlots;
    int[] demands;

    private int lb(int v) {
        return (int) Math.ceil(demands[v] * 0.95);
    }

    private int ub(int v) {
        return (int) Math.floor(demands[v] * 1.1);
    }

    public void model() {
        int maxDemand = maxOf(demands), nVariations = demands.length, nTemplates = nVariations;
        Var[][] d = array("d", size(nTemplates, nVariations), dom(range(nSlots + 1)),  
            "d[t][v] is the number of occurrences of variation v on template t");
        Var[] p = array("p", size(nTemplates), dom(range(maxDemand + 1)),  
            "p[t] is the number of printings of template t");
        Var[] u = array("u", size(nTemplates), dom(0, 1),  
            "u[t] is 1 iff the template t is used");

        forall(range(nTemplates), t -> sum(d[t], EQ, nSlots))  
            .note("all slots of all templates are used");
        forall(range(nTemplates), t -> equivalence(eq(u[t], 1), gt(p[t], 0)))  
            .note("if a template is used, it is printed at least once");

        if (modelVariant("m1")) {
            Var[][] pv = array("pv", size(nTemplates, nVariations), (t, v) -> dom(range(ub(v))),  
                "pv[t][v] is the number of printings of variation v by using template t");
            forall(range(nTemplates).range(nVariations), (t, v) -> equal(mul(p[t], d[t][v]), pv[t][v]))  
                .note("linking variables of arrays p and pv");
            forall(range(nVariations), v -> sum(columnOf(pv, v), IN, range(lb(v), ub(v) + 1)))  
                .note("respecting printing bounds for each variation v");
        }

        if (modelVariant("m2"))
            forall(range(nVariations), v -> sum(p, weightedBy(columnOf(d, v)), IN, range(lb(v), ub(v) + 1)))  
                .note("respecting printing bounds for each variation v");
    }
}
```
Two model variants, 'm1' and 'm2', have been considered. These model variants involve 3(+1) arrays of variables, and 3 types of constraints: sum, ordered (decreasing) and intension (equivalence and equal). The model variant 'm1' introduces some auxiliary variables in order to post a basic form of sum. Note that there is a block for breaking a few symmetries. A series of 3×5 instances has been selected for the competition: 5 instances for models 'm1', 'm2' and 'm1s' which is 'm1' without the symmetry-breaking constraints.

### 2.40 Traveling Tournament

This problem is related to Problem 068 on CSPLib. Many relevant information can be found at http://mat.gsia.cmu.edu/TOURN/. See also [9].

#### Description

"The Traveling Tournament Problem (TTP) is defined as follows. A double round robin tournament is played by an even number of teams. Each team has its own venue at its home city. All teams are initially at their home cities, to where they return after their last away game. The distance from the home city of a team to that of another team is known beforehand. Whenever a team plays two consecutive away games, it travels directly from the venue of the first opponent to that of the second. The problem calls for a schedule such that no team plays more than (two or) three consecutive home games or more than (two or) three consecutive away games, there are no consecutive games involving the same pair of teams, and the total distance traveled by the teams during the tournament is minimized."

#### Data

As an illustration of data specifying an instance of this problem, we have:

```
{
  "distances": [
    [0, 10, 15, 34],
    [10, 0, 22, 32],
    [15, 22, 0, 47],
    [34, 32, 47, 0]
  ]
}
```

#### Model

The MCSP3 model used for the competition is:

```java
class TravelingTournament implements ProblemAPI {
  int[][] distances;
```
private Table tableEnd(int i) {
    Table table = table().add(1, STAR, 0); // when playing at home, travel distance is 0
    for (int j = 0; j < distances.length; j++)
        if (j != i)
            table.add(0, j, distances[i][j]);
    return table;
}

private Table tableOther(int i) {
    int nTeams = distances.length;
    Table table = table().add(1, 1, STAR, STAR, 0);
    for (int j = 0; j < nTeams; j++)
        if (j != i) {
            table.add(0, 1, j, STAR, distances[i][j]);
            table.add(1, 0, STAR, j, distances[i][j]);
        }
    for (int j1 = 0; j1 < nTeams; j1++)
        for (int j2 = 0; j2 < nTeams; j2++)
            if (j1 != i && j2 != i && j1 != j2)
                table.add(0, 0, j1, j2, distances[j1][j2]);
    return table;
}

private Automaton automat() {
    String transitions = "(q,0,q01)(q,1,q11)(q01,0,q02)(q01,1,q11)(q11,0,q01)(q11,1,q12)(q02,1,q11)(q12,0,q01)";
    if (modelVariant("a2"))
        return automaton("q", transitions, finalStates("q01", "q02", "q11", "q12"));
    transitions = transitions + "(q02,0,q03)(q12,1,q13)(q03,1,q11)(q13,0,q01)";
    return automaton("q", transitions, finalStates("q01", "q02", "q03", "q11", "q12", "q13"));
}

private int[] allTeamsExcept(int i) {
    return range(distances.length).select(j -> j != i);
}

public void model() {
    int nTeams = distances.length, nRounds = nTeams * 2 - 2;
    Var[][] o = array("o", size(nTeams, nRounds), dom(range(nTeams)),
        "o[i][k] is the opponent (team) of the ith team at the kth round");
    Var[][] h = array("h", size(nTeams, nRounds), dom(0, 1),
        "h[i][k] is 1 iff the ith team plays at home at the kth round");
    Var[][] a = array("a", size(nTeams, nRounds), dom(0, 1),
        "a[i][k] is 0 iff the ith team plays away at the kth round");
    Var[][] t = array("t", size(nTeams, nRounds + 1), dom(distances),
        "t[i][k] is the travelled distance by the ith team at the kth round; an additional round for returning at home.");
    forall(range(nTeams), i -> cardinality(o[i], allTeamsExcept(i), CLOSED, occursEachExactly(2)))
        .note("each team must play exactly two times against each other team");
    forall(range(nTeams).range(nRounds), (i, k) -> element(columnOf(o, k), at(o[i][k]), takingValue(i)))
        .note("if team i plays against j at round k, then team j plays against i at round k");
    forall(range(nTeams).range(nRounds), (i, k) -> equal(h[i][k], not(a[i][k])))
        .note("playing home at round k iff not playing away at round k");
    forall(range(nTeams).range(nRounds), (i, k) -> element(columnOf(h, k), at(o[i][k]), takingValue(a[i][k])))
        .note("channeling the three arrays");
    forall(range(nTeams).range(nRounds).range(nRounds), (i, k1, k2) -> {
        if (k1 + 1 < k2)
            implication(eq(o[i][k1], o[i][k2]), ne(h[i][k1], h[i][k2]));
    }).note("playing against the same team must be done once at home and once away");
    forall(range(nTeams), i -> regular(h[i], automat()))
        .note("verifying the number of consecutive games at home, and consecutive games away");
CHAPTER 2. PROBLEMS AND MODELS

This model involves 4 arrays of variables and 6 types of constraints: cardinality, regular, element, allDifferent, extension and intension (equal, implication and lessThan). Note that we could have used a cache for the tables built by tableEnd() and tableOther() as well as for the automaton built by automat(). A series of 14 instances has been selected for the competition.

2.41 Travelling Salesman

This is the famous Travelling Salesman Problem (TSP). See for example TSPLIB.

Description

“Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?”

Data

As an illustration of data specifying an instance of this problem, we have:

```json
{
   "distances": [
       [0, 5, 6, 6, 6],
       [5, 0, 9, 8, 4],
       [6, 9, 0, 1, 7],
       [6, 8, 1, 0, 6],
       [6, 4, 7, 6, 0]
   ]
}
```

Model

The MCSP3 model used for the competition is:

```java
class TravellingSalesman implements ProblemAPI {
    int[][] distances;

    private Table distTable() {
        Table table = table();
        for (int i = 0; i < distances.length; i++)
```
for (int j = 0; j < distances.length; j++)
    if (i != j)
        table.add(i, j, distances[i][j]);
return table;
}

public void model() {
    int nCities = distances.length;
    Var[] c = array("c", size(nCities), dom(range(nCities)),
            "c[i] is the ith city of the tour");
    Var[] d = array("d", size(nCities), dom(distances),
            "d[i] is the distance between the cities i and i+1");

    allDifferent(c)
        .note("visiting each city only once");
    forall(range(nCities), i -> extension(vars(c[i], c[(i + 1) % nCities], d[i]), distTable()))
        .note("computing the distance between any two successive cities in the tour");

    minimize(SUM, d)
        .note("minimizing the total distance");
}

This model involves two arrays of variables and two types of constraints: allDifferent and extension. Of course, for large number of cities, the table built by the auxiliary method distTable() should be stored and reused, instead of being systematically computed. A series of 12 instances has been generated for the competition.
Chapter 3

Solvers

In this chapter, we introduce the solvers and teams having participated to the XCSP3 Competition 2018. When names of solvers are given in italic font, it means that a short description of these solvers are given by authors in the following pages.

- **BTD, miniBTD** (Philippe Jegou, Helene Kanso and Cyril Terrioux)
- **BTD$_{12}$, miniBTD$_{12}$** (Philippe Jegou, Djamal Habet, Helene Kanso and Cyril Terrioux)
- **Choco-solver** (Charles Prud’homme and Jean-Guillaume Fages)
- **Concrete** (Julien Vion)
- **CoSoCo** (Gilles Audemard)
- GG’s minicp (Arnaud Gellens and Simon Gustin)
- **macht, minimacht** (Djamal Habet and Cyril Terrioux)
- MiniCPFever (Victor Joos and Antoine Vanderschueren)
- **Mistral-2.0** (Emmanuel Hebrard and Mohamed Siala)
- **NACRE** (Gal Glorian)
- **OscaR** (OscaR Team)
- **PicatSAT** (Neng-Fa Zhou and Hakan Kjellerstrand)
- **Sat4j-CSP** (Daniel Le Berre and Emmanuel Lonca)
- **scop** (Takehide Soh, Daniel Le Berre, Mutsunori Banbara, Naoyuki Tamura)
- slowpoke (Alexandre Gerlache and vincent vandervilt)
- Solver of Xavier Schul and Yvhan Smal (Xavier Schul and Yvhan Smal)
- SuperSolver (Florian Stevenart Meeus and Jean-Baptiste Macq)
- The dodo solver (Alexandre Dubray)
BTD and miniBTD

Philippe Jégou¹, Hélène Kanso¹,², and Cyril Terrioux¹

¹ Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France
{philippe.jegou, cyril.terrioux}@lis-lab.fr
² Effat University, Jeddah, Saudi Arabia
hkanso@effatuniversity.edu.sa

1 Solver description

BTD and miniBTD are written in C++ and both implement the algorithm BTD-MAC+RST+Merge [2]. This algorithm exploits the structure of CSP instances thanks to the notion of tree-decomposition [3].

For the competition, we have made the following choices:

– the tree-decompositions are computed thanks to the heuristic H₅-TD-WT [2]
– dom/wdeg [1] is exploited for ordering the variables inside a cluster,
– lexico is used as value ordering heuristic,
– generalized arc-consistency is enforced by a propagation-based system exploiting events,
– restarts are performed according to a geometric restart policy based on the number of backtrack with an initial cutoff set to 100 and an increasing factor set to 1.1,
– the first root cluster is the cluster having the maximum ratio number of constraints to its size minus one and then, at each restart, the selected root cluster is one which maximizes the sum of the weights of the constraints whose scope intersects the cluster.

Note that, at now, BTD only takes into account the following constraints:

– intension,
– extension,
– allDifferent (the element <except> is not supported),
– allEqual,
– ordered,
– sum,
– maximum (the variant <arg_max> is not supported),
– minimum (the variant <arg_min> is not supported),
– element,
– channel,
– noOverlap (only the one dimensional form),
– instantiation.
2 Command line

BTD and miniBTD can be launched thanks to the following command line:

```
SOLVER 3 TIMELIMIT BENCHNAME
```

where:

– SOLVER is the path to the executable BTD or miniBTD,
– TIMELIMIT is the number of seconds allowed for solving the instance,
– BENCHNAME is the name of the XML file representing the instance we want to solve.

Acknowledgments

This work has been partially funded by the french "Agence nationale de la Recherche", reference ANR-16-C40-0028.

References

BTD_{12} and miniBTD_{12}

Djamal Habet\textsuperscript{1}, Philippe Jégou\textsuperscript{1}, Hélène Kanso\textsuperscript{1,2}, and Cyril Terrioux\textsuperscript{1}

\textsuperscript{1} Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France
\{djamal.habet, philippe.jegou, cyril.terrioux\}@lis-lab.fr
\textsuperscript{2} Effat University, Jeddah, Saudi Arabia
hkanso@effatuniversity.edu.sa

1 Solver description

BTD_{12} and miniBTD_{12} are written in C++ and both implement the algorithm BTD-MAC+RST+Merge \cite{1}. This algorithm exploits the structure of CSP instances thanks to the notion of tree-decomposition \cite{3}.

For the competition, we have made the following choices:

\begin{itemize}
  \item the tree-decompositions are computed thanks to the heuristic H\textsubscript{5}-TD-WT \cite{1}.
  \item an heuristic based on the exponential recency weighted average \cite{2,4} is exploited for ordering the variables inside a cluster,
  \item \textit{lexico} is used as value ordering heuristic,
  \item generalized arc-consistency is enforced by a propagation-based system exploiting events,
  \item restarts are performed according to a geometric restart policy based on the number of backtrack with an initial cutoff set to 100 and an increasing factor set to 1.1,
  \item the first root cluster is the cluster having the maximum ratio number of constraints to its size minus one and then, at each restart, the selected root cluster is one which maximizes the sum of the weights of the constraints whose scope intersects the cluster.
\end{itemize}

Note that, at now, BTD_{12} only takes into account the following constraints:

\begin{itemize}
  \item intension,
  \item extension,
  \item allDifferent (the element \texttt{<except>} is not supported),
  \item allEqual,
  \item ordered,
  \item sum,
  \item maximum (the variant \texttt{<arg_max>} is not supported),
  \item minimum (the variant \texttt{<arg_min>} is not supported),
  \item element,
  \item channel,
  \item noOverlap (only the one dimensional form),
  \item instantiation.
\end{itemize}
2 Command line

BTD$_{12}$ and miniBTD$_{12}$ can be launched thanks to the following command line:

\[ \text{SOLVER } 12 \text{ TIMELIMIT BENCHNAME} \]

where:

– \text{SOLVER} is the path to the executable BTD or miniBTD,
– \text{TIMELIMIT} is the number of seconds allowed for solving the instance,
– \text{BENCHNAME} is the name of the XML file representing the instance we want to solve.

Acknowledgments

This work has been partially funded by the french "Agence nationale de la Recherche", reference ANR-16-C40-0028.

References

Choco solver 4 : a Free Open-Source Java Library for Constraint Programming

Charles Prud’homme\textsuperscript{1} and Jean-Guillaume Fages\textsuperscript{2}

\textsuperscript{1} IMT-Atlantique, France,  
Charles.Prudhomme@imt-atlantique.fr
\textsuperscript{2} COSLING S.A.S., France,  
jg.fages@cosling.com

Abstract. Choco solver is a Free Open-Source Java library dedicated to Constraint Programming. The user models its problem in a declarative way by stating the set of constraints that need to be satisfied in every solution. Then, the problem is solved by alternating constraint filtering algorithms with a search mechanism.

Keywords: constraint programming, solver

1 Introduction

Choco solver already has a long history : the first line of code was written in 1999 \cite{11}. Since then, the code has been frequently re-engineered and released, up to version 4.0.8, the last current released \cite{12}. It contains numerous variables, many (global) constraints and search procedures, to provide wide modeling perspectives.

Choco solver is used by the academy for teaching and research and by the industry to solve real-world problems, such as program verification, data center management, timetabling, scheduling and routing.

Several useful extra features are also available such as an extension that deals with graph variables, parsers to XCSP3 and FlatZinc or a minimalist profiler.

2 A Modeling API

Choco solver comes with the commonly used types of variables: integer variables with either bounded domain or enumerated one, boolean variables and set variables. Views \cite{14} but also arithmetical, relational and logical expressions are supported.

Up to 100 constraints –and more than 150 propagators– are provided : from classic ones, such as arithmetical constraints, to must-have global constraints, such as allDifferent or cumulative, and include less common even though useful ones, such as tree. One can pick some existing propagators to compose a new constraint or create its own one in a straightforward way by implementing a filtering algorithm and a satisfaction checker.
The library supports natively real variables and constraints also, and relies on Ibex [3] to solve the continuous part of the problem [4]. Graph variables and constraints on them can be declared by adding a dependency to choco-graph [6].

3 Resolution Toolbox

Choco solver has been carefully designed to offer wide range of resolution configurations and good resolution performances. Backtrackable primitives and structures are based on trailing. The propagation engine deals with seven priority levels and manage either fine or coarse grain events which enables to get efficient incremental constraint propagators.

The search algorithm relies on three components Propagate, Learn and Move depicted in [9]. Such a generic search algorithm is then instantiated to DFS, LDS [8], DDS [16], HBFS [1] or LNS [15].

The search process can also be greatly improved by various built-in search strategies such as DomWDeg [2], ABS [10], IBS [13], BIVS [5], first-fail [7], etc., and by creating a problem-adapted search strategy.

One can solve a problem in many ways: checking satisfaction, finding one or all solutions, optimizing one or more objectives and solving on one or more thread, or simply propagating. The search process itself is monitorable and extensible.

4 The code and the dev team

Structurally, Choco solver is made of 573 Java classes which represents roughly 53k source code lines. The source code is hosted on GitHub under a BSD 4-clause licence. The project is mainly developed and maintained by Charles Prud’homme and Jean-Guillaume Fages, they can count on attentive contributors.

Tutorials, Javadoc and a user guide can be referred to, as long as a Google group.

References


Concrete 3.9.2: A CSP solving software & API

Julien Vion

http://github.com/concrete-cp/concrete

1 Features

Concrete is a CSP constraint solver written in Scala 2.12 [15]. We always try to use up-to-date dependencies. Concrete is a pretty standard CP solver, which solves CSP instances using depth-first search and AC or weaker variants for propagation. The two main specific aspects of Concrete are:

- the use of persistent data structures [16] for managing domain states and some constraint states. We use bit vectors copied on-the-fly, hash tries, trees with a high branching factor, and red-black trees. For the state of many constraints, semi-persistent data structures (mainly sparse sets [4]) or backtrack-stable data (watched literals [6] or residues [9]) are preferred.

- the use of the companion project CSPOM 2.25 [19], a solver-independent modeling assistant able to perform automatic reformulation such as constraint aggregation. CSPOM is able to parse problems written in FlatZinc [13], XCSP3 [1], the legacy XCSP2 format or its own Java and Scala DSL (yet to be documented).

Concrete can solve models defined with signed 32-bit integers. Domains are internally represented using either intervals, bit vectors, or red-black trees depending on domain density. Singleton and boolean domains are handled specifically. CSPOM represents domains with interval trees and supports infinite domains, but they must be fully defined during the compilation phase in order to be processed in Concrete. This allows to infer some variable domains, e.g. for auxiliary variables or variables defined by a constraint. Set variables are currently not supported.

The main loop of Concrete is a tail-recursive DFS. It allows to enumerate solutions or to search for an optimal solution. If used correctly, it is able to add constraints dynamically between solutions.

Concrete natively supports the following constraints:

- Extension (list of allowed or forbidden tuples). An optimized algorithm should be automatically selected for binary constraints (AC3-bit+rm) [11] or MDD [20].

- Linear \((a \cdot x + b \cdot y + \ldots \{= / < / \leq / \neq\} k)\). Bound consistency (except for \(\neq\)) [7] or full domain consistency for ternary constraints (using residues).
• Absolute value \((x = |y|)\). Bound or domain consistency (using residues).
• Distance \((x = |y - z|)\). Bound or domain consistency (using residues).
• All-different with 2-consistency or bound consistency [12].
• Cardinality (AtLeast/AtMost)
• Bin-packing [17]
• Channel \((x(i) = j \iff x(j) = i)\)
• Boolean clauses and XOR (using watched literals)
• Cumulative using profile and energetic reasoning
• Rectangle packing (diffN) using quad-trees and energetic reasoning
• Quadratic \((x = y \cdot z, x = y^2)\). Bound or domain consistency (using residues)
• Integer division and modulo. Bound or domain consistency (using residues)
• Element / Member (using watched literals and residues)
• Inverse \((x(i) = j \Rightarrow y(j) = i)\)
• Lex-Leq
• Lex-Neq
• Min/Max
• Subcircuit with defined starting point (uses Dijkstra shortest path algorithm)
• Regular and Sliding-Sum via MDD decomposition
• Generic reification (for any constraint \(C\), a boolean variable \(b\) can be defined s.t. \(b \iff C\))

All FlatZinc constraints are supported, and other documented MiniZinc constraints are supported \(\textit{via}\) provided decomposition or reformulation. All XCSP3 constraints selected for the 2018 competition are supported \(\textit{via}\) (trivial) decomposition or reformulation. Some XCSP3 constraints may not be supported.
2 Search strategies

Concrete solves CSP/COP using a binary depth-first tree search [8]. The default variable ordering heuristic is \texttt{dom/wdeg} [2] with incremental computation and random tiebreaking. The default value heuristic chooses the best known value first [22], then applies BBS, an heuristic that uses singleton assignments to find the value that maximizes potential solution quality [5]. Ties are broken randomly, with priority given to domain bounds. Sometimes, a random value is selected to improve diversity in search. Search is restarted periodically (with a geometric growth) to reduce long tails of search time [Gomes2000].

Propagation queue is managed using a coarse-grained constraint-oriented propagation scheme [3] with dynamic and constraint-specific propagation ordering heuristic [21]. Constraint entailment is managed when it can be detected easily.

3 Present and near-future of Concrete

Feedback from the competition allowed us to improve Concrete in many ways in late 2017. Bugs have been fixed, some heuristics have been improved. For 2018, we implemented new constraints selected for the competition (mainly Subcircuit), BBS, and tuned some parameters.

We recently reimplemented nogood recording from restarts [10], which was available in older versions of Concrete for binary nogoods only, but was dropped off a few years ago. A full nogood-managing global constraint is now available with innovating tricks.

Common subexpression elimination (ACCSE) [14] for CSPOM is now fast and stable.

License. Concrete is free and open-source software, released under the terms of the GNU LGPL 3.0 license [18]. Concrete is © Julien Vion, CNRS and Univ. Polytechnique des Hauts de France.

References


CoSoCo is a constraint solver written in C++. The main goal is to build a simple, but efficient constraint solver. Indeed, the main part of the solver contains less than 2,000 lines of code. CoSoCo will be available on bitbucket as soon as possible. CoSoCo recognizes XCSP3 [2] by using the C++ parser that can be downloaded at https://github.com/xcsp3team/XCSP3-CPP-Parser. It can deal with almost all XCSP3 Core constraints. The part related to all constraint propagators contains around 4,500 lines of codes.

This is the second participation of CoSoCo to XCSP competitions. Unfortunately, no improvements have been done this year, just few new additional constraints are supported.

CoSoCo performs backtrack search, enforcing (generalized) arc consistency at each node (when possible). The main components are:

- \textit{dom/wdeg} [1] as variable ordering heuristic;
- \textit{lexico} as value ordering heuristic;
- \textit{lc}(1), last-conflict reasoning with a collecting parameter \( k \) set to 1, as described in [4];
- a geometric restart policy;
- a variable-oriented propagation scheme [5], where a queue \( Q \) records all variables with recently reduced domains (see Chapter 4 in [3]).

Acknowledgements

This work would not have taken place without Christophe Lecoutre. I would like to thank him very warmly for his support.

References

macht and minimacht

Djamal Habet and Cyril Terrioux
Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France
\{djamal.habet, cyril.terrioux\}@lis-lab.fr

1 Solver description

macht and minimacht are written in C++ and both implement the algorithm MAC+RST+NG [1].
For the competition, we have made the following choices:
- an heuristic based on the exponential recency weighted average [2,3] is exploited for ordering the variables,
- \textit{lexic} is used as value ordering heuristic,
- generalized arc-consistency is enforced by a propagation-based system exploiting events,
- restarts are performed according to a geometric restart policy based on the number of backtrack with an initial cutoff set to 100 and an increasing factor set to 1.1,

Note that, at now, macht only takes into account the following constraints:
- intension,
- extension,
- allDifferent (the element \texttt{<except>} is not supported),
- allEqual,
- ordered,
- sum,
- maximum (the variant \texttt{<arg\_max>} is not supported),
- minimum (the variant \texttt{<arg\_min>} is not supported),
- element,
- channel,
- noOverlap (only the one dimensional form),
- instantiation.

2 Command line

macht and minimacht can be launched thanks to the following command line:

\texttt{SOLVER TIMELIMIT BENCHNAME}

where:
- \texttt{SOLVER} is the path to the executable macht or minimacht,
- \texttt{TIMELIMIT} is the number of seconds allowed for solving the instance,
- \texttt{BENCHNAME} is the name of the XML file representing the instance we want to solve.
Acknowledgments

This work has been partially funded by the French "Agence nationale de la Recherche", reference ANR-16-C40-0028.

References

Mistral 2.0
XCSP3 Competition 2017

Emmanuel Hebrard\textsuperscript{1} and Mohamed Siala\textsuperscript{2}

\textsuperscript{1} LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France
hebrard@laas.fr
\textsuperscript{2} Insight Centre for Data Analytics
Department of Computer Science, University College Cork, Ireland
mohamed.siala@insight-centre.org

Mistral is an open source constraint programming library written in C++ and available on GitHub (https://github.com/ehebrard/Mistral-2.0). It implements a modelling API, however, it can also read instance files in XCSP3 or FlatZinc format. Moreover, it is fully interfaced with Numberjack \cite{6} which provides a Python API for modelling and solving combinatorial optimization problems using several back-end solvers.

Solver Engine

The solver engine relies on standard mechanisms, using a priority constraint queue and a domain event stack to implement the propagation procedure. Moreover, it supports dynamic type change for variables: domains are initially implemented using intervals or Boolean types whenever possible, and can be changed to (bit)sets during search when a propagator requires it. The backtracking mechanism is implemented using a trail in a standard way.

Propagators

Several classic global constraints are implemented, such as LexOrdering \cite{4}, bound consistency propagator for AllDifferent \cite{9} and Gcc \cite{10}. Moreover, less standard constraints were implemented within the context of research articles on constraint propagation, such as the AtMostSeqCard constraint for car-sequencing \cite{12} or a VertexCover constraint \cite{3} to reason about cliques, independant set or vertex covers.

Search Strategy

The search heuristic used for the XCSP3 competition is based on Last Conflit \cite{8}, using a variant of Weighted Degree \cite{2} as default strategy: in the case of failure
raised by a propagator of a global constraint, an explanation of the conflict is computed and only the weight of the variables participating in the conflict is incremented. This heuristic is fully described in [7]. Moreover, given the next variable \( x \) to branch on, the solver chooses the value that was assigned to \( x \) in the best solution found so far, if possible, or the minimum value in the domain of \( x \) otherwise.

**Applications of Mistral**

Mistral was used to implement a state-of-the-art method for disjunctive scheduling which improved several best known results on classic benchmarks [5]. More recently, some clause learning methods were implemented in Mistral, still improving the results on disjunctive scheduling [1] and car-sequencing problems [11]. These methods were not used within the context of the XCSP3 competition.

**References**


NACRE
XCSP3 2018 Competition
Gaël Glorian
CRIL-CNRS, UMR 8188
Université d’Artois, F-62307 Lens France
glorian@cril.fr
http://www.cril.univ-artois.fr/~glorian/

NACRE (Nogood And Clause Reasoning Engine) is a constraint solver written in C++. The main purpose of this solver is to experiment nogood recording (with a clause reasoning engine) in Constraint Programming (CP). In particular, the data structures of the solver have been carefully designed to play around nogoods and clauses. This is the first version of the solver and its first submission ever to a competition.

Although most of the XCSP3-core constraints are implemented, the current version of NACRE does not support all of them and cannot currently deal with optimization (COP). Therefore it will compete only into the CSP MiniTrack. A mini version, the one submitted to the competition, is available on a public deposit at https://github.com/gglorian/nacre_mini. You can always contact me for more information.

The submitted version uses all the following technologies:
- arc consistency [3]
- variable selection heuristic: dom/wdeg [1]
- value selection heuristic: minimum
- restart policy: \texttt{luby} \times 100 [2]
- a clause reasoning engine with lazy explanations, backjumping, clauses database reduction, etc.

Acknowledgements

This work has been supported by the project CPER Data from the region "Hauts-de-France". This solver is produced for my PhD thesis work on data structures designed for clause-based nogood recording. A big thank-you to Gilles Audemard, for his support and help.

References


This document shortly describes the solvers submitted for the Second XCSP3 competition [1] and how to launch them using the command line interface. This competition has for aim to assess the performances of solvers for CSP (Constraint Satisfaction Problem) and COP (Constrained Optimization Problem) instances in the XCSP3 format [2].

1 Solvers submitted

The solvers submitted have been developed by members of the beCool team [3] which is a research group of the ICTEAM institute at the Université catholique de Louvain in Belgium active in Constraint Programming and Local Search. The solvers use OscaR [4], a scala toolkit for Operations Research problems including a Constraint Programming engine.

The source code of the solvers is available at https://bitbucket.org/oscarlib/oscar/wiki/Home

1.1 Conflict ordering solver

The Conflict ordering solver performs a complete search with a conflict ordering heuristic [5]. An extension of this solver with a restart strategy was also submitted.

1.2 Hybrid solver

The Hybrid solver combines the approaches of the Conflict ordering with an Adaptive Large Neighbourhood Search approach [6]. It consists in performing a large neighbourhood search using different heuristics in order to adapt the solver during the search by using more often the efficient heuristics. The solver starts the search with a complete search using the conflict ordering heuristic in order to find and prove the optimum for small instances and to get a good starting solution for larger instances. Then it switches to an ALNS search to improve the current solution. Finally the solver restarts a complete search with the most successful heuristic in the ALNS in order to try to prove the optimality of the best solution found by the ALNS.

1.3 Parallel solver

The parallel solver uses an Embarrassingly Parallel Search approach with a conflict ordering heuristic.

2 Command line usage

All the solvers use the same command line interface. The solvers have been uploaded as executable jar files. To launch a solver with an instance file, use the following command:

```
java -XmxMEMLIMITm -jar SOLVER --randomseed RANDOMSEED --timelimit TIMELIMIT --memlimit MEMLIMIT --nbcore NBCORE (--tmpdir TMPDIR --dir DIR) BENCHNAME
```

where:

- SOLVER should be replaced by the path to the solver jar file.
• RANDOMSEED should be replaced by the random seed value.
• TIMELIMIT should be replaced by the allocated time in seconds.
• MEMLIMIT should be replaced by the allocated memory in MB.
• NBCORE should be replaced by the number of cores allocated.
• TMPDIR which is optional should be replaced by the temporary directory path.
• DIR which is optional should be replaced by the solver file directory.
• BENCHNAME which is mandatory and must always be the last argument should be replaced by the path to the instance file.

The order of the named arguments does not matter as long as their value directly follows the corresponding flag starting with a double dash and they are placed between the solver file path and the instance path. It is necessary to use the java Xmx command to set the maximal memory limit if the allocated memory is superior to the default value. Note that the values of TMPDIR and DIR will not be used but the arguments are still supported.

References

A Picat-based XCSP Solver – from Parsing, Modeling, to SAT Encoding

Neng-Fa Zhou and Håkan Kjellerstrand
1 CUNY Brooklyn College & Graduate Center
2 hakank.org

Abstract. This short paper gives an overview of a Picat-based XCSP3 solver, named PicatSAT, submitted to the 2018 XCSP competition. The solver demonstrates the strengths of Picat, a logic-based language, in parsing, modeling, and encoding constraints into SAT.

XCSP3

XCSP3 [1] is an XML-based domain specific language for describing constraint satisfaction and optimization problems (CSP). XCSP3 is positioned as an intermediate language for CSPs. It does not provide high-level constructs as seen in modeling languages. However, XCSP3 is significantly more complex than a canonical-form language, like FlatZinc. A constraint can sometimes be described in either the standard format or simplified format. The advanced format, which is used by group and matrix constraints, allows more compact description of constraints.

Picat

Picat [6] is a simple, and yet powerful, logic-based multi-paradigm programming language. Picat is a Prolog-like rule-based language, in which predicates, functions, and actors are defined with pattern-matching rules. Picat incorporates many declarative language features for better productivity of software development, including explicit non-determinism, explicit unification, functions, list comprehensions, constraints, and tabling. Picat also provides imperative language constructs, such as assignments and loops, for programming everyday things. Picat provides facilities for solving combinatorial search problems, including a common interface with CP, SAT, and MIP solvers, tabling for dynamic programming, and a module for planning. PicatSAT uses the SAT module, which generally performs better than the CP and MIP modules on competition benchmarks.

Parsing

The availability of different formats in XCSP3 makes it a challenge to parse the XCSP3 language. A parser implemented in C++ by the XCSP designers has
more than 10,000 lines of code. The entire Picat implementation of XCSP3 has about 2,000 lines of code, two-thirds of which is devoted to parsing and syntax-directed translation. As illustrated in the following example, Picat is well suited to parsing.

% E -> T E'
ex(Si,So) => term(Si,S1), ex_prime(S1,So).

% E' -> + T E' | - T E' | e
ex_prime(['+'|Si],So) =>
    term(Si,S1),
    ex_prime(S1,So).
ex_prime(['-'|Si],So) =>
    term(Si,S1),
    ex_prime(S1,So).
ex_prime(Si,So) => So = Si.

The parser follows the framework for translating context-free grammar into Prolog [3]: a non-terminal is encoded as a predicate that takes an input string (Si) and an output string (So), and when the predicate succeeds, the difference Si-So constitutes a string that matches the nonterminal. Unlike in Prolog, pattern-matching rules in Picat are fully indexed, which facilitates selecting right rules based on look-ahead tokens.

Modeling

It is well known that loops and list comprehensions are a necessity for modeling CSPs. The following Picat example illustrates the convenience of these language constructs for modeling.

post_constr(allDifferentMatrix(Matrix)) =>
    NRows = len(Matrix),
    NCols = len(Matrix[1]),
    foreach (I in 1..NRows)
        all_different(Matrix[I])
    end,
    foreach (J in 1..NCols)
        all_different([Matrix[I,J] : I in 1..NRows])
    end.

The allDifferentMatrix(Matrix) constraint takes a matrix that is represented as a two-dimensional array, and posts an all_different constraint for each row and each column of the matrix.

SAT Encoding

PicatSAT adopts the log encoding for domain variables. While log encoding had been perceived to be unsuited to arithmetic constraints due to its hindrance
to propagation [2], we have shown that log encoding can be made competitive with optimizations [4]. There are hundreds of optimizations implemented in Picasat, and they are described easily as pattern-matching rules in Picat. We have also shown that, with specialization, the binary adder encoding is not only compact, but also generally more efficient than BDD encodings for PB constraints [5]. Picasat adopts specialized decomposition algorithms for some of the global constraints. While competitive overall, Picasat is not competitive with state-of-the-art CP solvers on benchmarks that use path-finding constraints that require reachability checking during search. The future work is to design efficient encodings for these global constraints.

References

1 Solver Description

Sat4j-CSP is a constraint satisfaction and optimization library developed in Java. It is a mature project (Sat4j was born in 2004) included since June 2008 in the Eclipse IDE as part of its plugin dependency management engine. As indicated by its name, the original library design was built on top of a SAT solver. Taking advantage of this initial solver, several new kinds of solvers were added to the library (pseudo-Boolean solver, MAXSAT solver, ...) including a CSP solver.

Sat4J-CSP3, the version submitted to the 2018 XCSP3 Competition\(^1\), is far more than an improvement to Sat4j-CSP2\(^4\) giving it the ability to read XCSP3 input files. While Sat4j-CSP2 was intended to handle the most common constraints (extension, intension, allDifferent), the current version was built with the idea to handle the whole set of constraints proposed by the specifications of the XCSP3-core instance format\(^2\). Since, for most constraints, we had to start from scratch, we just translated most of the constraints into intension ones, taking advantage of the instance parser provided by Christophe Lecoutre\(^2\).

The intension constraints encoder developed for Sat4j-CSP2 was simply evaluating each constraint using javascript (Rhino library) and generated the whole set of nogoods for each of them. Although it was sufficient for instances with intension constraints considering few variables with small domains, the encoding of global constraints into intension constraints made the encoder absolutely inefficient. We thus developed a new intension constraint encoder, based on the principle of the Tseitin encoding:

1. from a constraint, we build a tree where the nodes are labeled with operators, and leaves with variables or constants;
2. each node is associated with a new integer variable giving its value (0 or 1 for nodes labeled with Boolean operators);
3. considering the nodes from the leaves to the root, the mapping between the values of its children and its own value is encoded using Boolean constraints;
4. the value of the root is enforced to a non-zero value. Concerning the optimization of objectives in functional forms, the same algorithm is used except for the last step: instead of enforcing the tree root value, the variable it is associated with is set as the objective function.

\(^1\) [http://xcsp.org/competition](http://xcsp.org/competition)
2 Future Work

Sat4j-CSP3 was developed with the aim to handle the whole set of constraints allowed in XCSF3-core. However, at this step, there is room for improvements. First, the encoding of integer variables uses one binary variable for each value in its range. For arrays of integer variables taking their values in large domains, this results in the declaration of a huge amount of Boolean variables, making the solver running out of memory. A slight adaptation of some encodings used for At-Most-1 constraints (like the binary encoding or the Two-Product encoding[3]) may help in terms of efficiency to keep the memory consumption under the system limits.

At this time, we did not develop any specific constraint encodings. We translated most of the global constraints into intension ones, so we do not take advantage of known encodings of these constraints. In addition, new integer variable encodings (as described in the beginning of this section) may bring some efficiency tracks to intension constraints encoding.

Finally, we plan to support several multicriteria optimization processes. This is possible because Sat4j has such capabilities for some families of Boolean functions.

3 Acknowledgments

This work has been supported by the project CPER DATA from the “Hauts-de-France” Region.

References

sCOP: SAT-based Constraint Programming System

XCSP3 Competition in 2018

Takehide Soh¹, Daniel Le Berre², Mutsunori Banbara¹, and Naoyuki Tamura¹

¹ Information Science and Technology Center, Kobe University, Japan
   {soh@lion.,banbara8,tamura@kobe-u.ac.jp}
² CRIL-CNRS, Université d’Artois, France
   leberre@cril.fr

1 Overview

sCOP is a SAT-based constraint programming system written in Scala. Like Sugar [3] and Diet-Sugar [2], sCOP encodes XCSP3 instances into conjunctive normal form (CNF) formulas using the order encoding [5, 4] and the log encoding for Pseudo-Boolean (PB) constraints [2]. Then, sCOP launches a SAT solver which will return a model if any. In last, a solution of the XCSP3 instance is decoded from the model computed.

This figure shows the framework of sCOP. In the following, we briefly explain each part of this framework.

2 Parsing and Normalizations

Parsing is done by using an official tool XCSP3-Java-Tools ³. Currently, sCOP accepts constraints in the XCSP3-core language⁴.

Normalizations in sCOP are almost same as ones in Sugar [3] and are follows:

Global Constraints. global constraints are translated into intensional constraints by a straightforward way but we use extra pigeon hole constraints for alldifferent constraints.

Extensional Constraints. extensional constraints are translated into intensional constraints by using a variant of multi-valued decision diagrams. This is a difference to ones in Sugar.

Intensional Constraints. using Tseitin transformation, intensional constraints are normalized to be in the form of CNF over linear comparisons $\sum a_i x_i \geq k$ where $a_i$’s are integer coefficients, $x_i$’s are integer variables and $k$ is an integer constant.

³ https://github.com/xcsp3team/XCSP3-Java-Tools
⁴ http://www.xcsp.org/specifications
3 Propagations and Encoding

Constraint propagations are executed to the normalized CSP (clausal CSP, i.e., in the form of CNF over linear comparisons $\sum_i a_i x_i \geq k$) to remove redundant values, variables, and linear comparisons. Currently, it is done by using an AC3 like algorithm.

Encoding methods used in sCOP are the followings:

Order Encoding [5, 4]. the order encoding uses propositional variables $p_{x \geq d}$’s meaning $x \geq d$ for each domain value $d$ of each integer variable $x$. To encode linear comparisons, Algorithm 1 of the literature [4] is used in sCOP.

Log Encoding. the log encoding uses a binary representation of integer variables. There are several ways to encode linear comparisons by using those propositional variables. In sCOP, we replace all integer variables with its binary representation—it gives us a set of PB constraints. We then encode PB constraints into CNF formulas by using the BDD encoding [1].

sCOP basically uses the order encoding but uses the log encoding in case that the huge number of clauses is expected to be encoded. For this expectation, the idea of domain product criteria [2] is used.

4 SAT Solvers

For sequential solving, sCOP uses a SAT solver MapleCOMSPS $^5$ which is a winning solver on the main track of the SAT competition 2016. It also shows a good performance for solving CSP instances encoded by sCOP. For parallel solving, sCOP uses a SAT solver glucose-syrup $^6$ which is a winning solver on the parallel track of the SAT competition 2017.

Acknowledgements. We would like to thank to the competition organizers for their huge efforts. This work was supported by JSPS KAKENHI Grant Numbers JP16K16036 and JP16H02803.

References


$^5$ https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/
$^6$ http://www.labri.fr/perso/lsimon/glucose/
Chapter 4

Results

In this chapter, rankings for the XCSP3 Competition 2018 are given. We also make a few general comments about the results. Importantly, remember that you can find all detailed results, including all traces of solvers at http://www.cril.fr/XCSP18/.

4.1 Rankings

Remember that the tracks of the competition are given by the following tables:

<table>
<thead>
<tr>
<th>Problem</th>
<th>Goal</th>
<th>Exploration</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSP</td>
<td>one solution</td>
<td>sequential</td>
<td>40 minutes</td>
</tr>
<tr>
<td>CSP</td>
<td>one solution</td>
<td>parallel</td>
<td>40 minutes</td>
</tr>
<tr>
<td>COP</td>
<td>best solution</td>
<td>sequential</td>
<td>4 minutes</td>
</tr>
<tr>
<td>COP</td>
<td>best solution</td>
<td>sequential</td>
<td>40 minutes</td>
</tr>
<tr>
<td>COP</td>
<td>best solution</td>
<td>parallel</td>
<td>40 minutes</td>
</tr>
</tbody>
</table>

Table 4.1: Standard Tracks.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Goal</th>
<th>Exploration</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSP</td>
<td>one solution</td>
<td>sequential</td>
<td>40 minutes</td>
</tr>
<tr>
<td>COP</td>
<td>best solution</td>
<td>sequential</td>
<td>40 minutes</td>
</tr>
</tbody>
</table>

Table 4.2: Mini-Solver Tracks.

Also, note that:

- The cluster was provided by CRIL and is composed of nodes with two quad-cores (Intel @ 2.67GHz with 32 GiB RAM).
- Hyperthreading was disabled.
- Sequential solvers were run on one processor (4 cores) and were allocated 15,500 MiB of memory.
• Parallel solvers were run on two processors (8 cores) and were allocated 31,000 MiB of memory.

• The selection of instances for the Standard tracks was composed of 236 CSP and 346 COP instances.

• The selection of instances for the Mini-solver tracks was composed of 176 CSP and 188 COP instances.

**About the Ranking.** It is based on the number of times a solver is able to prove a result (satisfiability for CSP, optimality for COP). Notice that for COP, another viewpoint is the number of times a solver can give the best known answer (optimality or best known bound); it is then given between parentheses in some tables.

Tables 4.3, 4.4 and 4.5 respectively give rankings for sequential (standard) solvers on CSP, COP and 'fast COP' instances. Table 4.7 gives the ranking for parallel solvers on CSP instances; note that there was not enough contestants for having a relevant ranking for parallel solvers on COP instances (but look at good results of Choco-solver 4.0.7b par on the website). Finally, Tables 4.7 and 4.8 respectively give rankings for mini-solvers on CSP and COP instances.

### 4.2 A few Comments

For simplicity, the solvers that encode constraints in SAT (PicatSAT, Sat4j and sCOP) will be called *SAT-based* solvers. The solvers that use more traditional CP techniques (Choco, Concrete, CoSoCo, Mistral, and OscaR) will be called *CP-based* solvers. Nacre is hybrid as it involves a strong clause reasoning engine. Finally, the solvers that exploit tree decomposition techniques (BTD, BTD12, and macht) will be called *decomposition-based* solvers.

Overall, one can observe that three types of solvers are rather complementary. It means that there are problems where SAT-based solvers, CP-based solvers and decomposition-based solvers are clearly superior. This is an interesting lesson from this competition.

#### 4.2.1 Standard Track – CSP – Sequential

**BIBD** Recall that two series 'sum' and 'sc' of 6 instances each have been selected. The series 'sc' is obtained by introducing auxiliary variables. Rather surprisingly, the solvers have very similar results on instances from the two series. Best solvers are PicatSAT, sCOP and Mistral.

**Car Sequencing** SAT-based solvers sCOP and PicatSAT are the best competitors, i.e., most robust, for this problem, succeeding in solving all (19) instances.

**ColouredQueens** All solvers have difficulty in scaling up. All competitors only solve 3 or 4 instances out of 12.

**Crosswords** These instances involve large table constraints. 8 instances out of 13 remain unsolved. Although the results obtained by the competitors are close, BTD appears to be the most efficient solver (decomposition seems to be effective).

**Dubois** Unsurprisingly, because of the SAT origin of the problem, SAT-based solvers PicatSAT and sCOP are very efficient. Interestingly, BTD is also able to solve all instances within a few milliseconds.
4.2. A FEW COMMENTS

<table>
<thead>
<tr>
<th>Solver</th>
<th>#solved</th>
<th>%inst.</th>
<th>%VBS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Best Solver (VBS)</td>
<td>164</td>
<td>104 SAT, 60 UNSAT</td>
<td>69%</td>
</tr>
<tr>
<td>scop order+MapleCOMSPS</td>
<td>146</td>
<td>92 SAT, 54 UNSAT</td>
<td>62%</td>
</tr>
<tr>
<td>scop both+MapleCOMSPS</td>
<td>140</td>
<td>87 SAT, 53 UNSAT</td>
<td>59%</td>
</tr>
<tr>
<td>PicatSAT 2018-08-14</td>
<td>138</td>
<td>85 SAT, 53 UNSAT</td>
<td>58%</td>
</tr>
<tr>
<td>Mistral-2.0</td>
<td>116</td>
<td>80 SAT, 36 UNSAT</td>
<td>49%</td>
</tr>
<tr>
<td>Choco-solver 4.0.7b seq</td>
<td>115</td>
<td>77 SAT, 38 UNSAT</td>
<td>49%</td>
</tr>
<tr>
<td>Concrete 3.9.2</td>
<td>92</td>
<td>64 SAT, 28 UNSAT</td>
<td>39%</td>
</tr>
<tr>
<td>OscaR-Conf. Ordering+restarts</td>
<td>90</td>
<td>62 SAT, 28 UNSAT</td>
<td>38%</td>
</tr>
<tr>
<td>Concrete 3.9.2-SuperNG</td>
<td>84</td>
<td>55 SAT, 29 UNSAT</td>
<td>36%</td>
</tr>
<tr>
<td>Sat4j-CSP</td>
<td>83</td>
<td>40 SAT, 43 UNSAT</td>
<td>35%</td>
</tr>
<tr>
<td>OscaR - Conflict Ordering</td>
<td>81</td>
<td>51 SAT, 30 UNSAT</td>
<td>34%</td>
</tr>
<tr>
<td>cosoco 1.12</td>
<td>79</td>
<td>53 SAT, 26 UNSAT</td>
<td>33%</td>
</tr>
<tr>
<td>BTD</td>
<td>76</td>
<td>32 SAT, 44 UNSAT</td>
<td>32%</td>
</tr>
<tr>
<td>macht</td>
<td>66</td>
<td>33 SAT, 33 UNSAT</td>
<td>28%</td>
</tr>
</tbody>
</table>

Table 4.3: Ranking for (Standard Track – CSP – sequential) over 236 instances.

<table>
<thead>
<tr>
<th>Solver</th>
<th>#best</th>
<th>%inst.</th>
<th>%VBS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Best Solver (VBS)</td>
<td>146</td>
<td>146 OPT</td>
<td>42%</td>
</tr>
<tr>
<td>PicatSAT 2018-08-14</td>
<td>132</td>
<td>132 OPT</td>
<td>38%</td>
</tr>
<tr>
<td>Concrete 3.9.2</td>
<td>105</td>
<td>105 OPT</td>
<td>30%</td>
</tr>
<tr>
<td>Choco-solver 4.0.7b seq</td>
<td>102</td>
<td>102 OPT</td>
<td>29%</td>
</tr>
<tr>
<td>OscaR-Conf. Ordering+restarts</td>
<td>99</td>
<td>99 OPT</td>
<td>29%</td>
</tr>
<tr>
<td>Concrete 3.9.2-SuperNG</td>
<td>99</td>
<td>99 OPT</td>
<td>29%</td>
</tr>
<tr>
<td>cosoco 1.12</td>
<td>64</td>
<td>64 OPT</td>
<td>18%</td>
</tr>
<tr>
<td>OscaR - Hybrid 2018-08-14</td>
<td>61</td>
<td>61 OPT</td>
<td>18%</td>
</tr>
<tr>
<td>Sat4j-CSP</td>
<td>54</td>
<td>54 OPT</td>
<td>16%</td>
</tr>
</tbody>
</table>

Table 4.4: Ranking for (Standard Track – COP – sequential) over 346 instances. Between parentheses, the number of times the solver can give the best known result.

<table>
<thead>
<tr>
<th>Solver</th>
<th>#best</th>
<th>%inst.</th>
<th>%VBS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Best Solver (VBS)</td>
<td>316</td>
<td>91%</td>
<td>100%</td>
</tr>
<tr>
<td>Concrete 3.9.2</td>
<td>151</td>
<td>44%</td>
<td>48%</td>
</tr>
<tr>
<td>Choco-solver 4.0.7b seq</td>
<td>146</td>
<td>42%</td>
<td>46%</td>
</tr>
<tr>
<td>OscaR - Hybrid</td>
<td>139</td>
<td>40%</td>
<td>44%</td>
</tr>
<tr>
<td>OscaR - Conflict Ordering with restarts</td>
<td>133</td>
<td>38%</td>
<td>42%</td>
</tr>
<tr>
<td>Concrete 3.9.2-SuperNG</td>
<td>129</td>
<td>37%</td>
<td>41%</td>
</tr>
<tr>
<td>Mistral-2.0</td>
<td>123</td>
<td>36%</td>
<td>39%</td>
</tr>
<tr>
<td>cosoco</td>
<td>107</td>
<td>31%</td>
<td>34%</td>
</tr>
<tr>
<td>Sat4j-CSP</td>
<td>78</td>
<td>23%</td>
<td>25%</td>
</tr>
</tbody>
</table>

Table 4.5: Ranking for (Standard Track – COP – sequential - 4’) over 346 instances. For this fast track, we consider the number of times the solver gives the best known result.
### CHAPTER 4. RESULTS

<table>
<thead>
<tr>
<th>#solved</th>
<th>%inst.</th>
<th>%VBS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Best Solver (VBS)</td>
<td>168</td>
<td>104 SAT, 64 UNSAT</td>
</tr>
<tr>
<td>1 scop order+glucose-syrup</td>
<td>151</td>
<td>95 SAT, 56 UNSAT</td>
</tr>
<tr>
<td>2 scop both+glucose-syrup</td>
<td>138</td>
<td>82 SAT, 56 UNSAT</td>
</tr>
<tr>
<td>3 Choco-solver 4.0.7b par</td>
<td>134</td>
<td>88 SAT, 46 UNSAT</td>
</tr>
<tr>
<td>4 OscaR - Parallel with EPS</td>
<td>89</td>
<td>56 SAT, 33 UNSAT</td>
</tr>
</tbody>
</table>

Table 4.6: Ranking for (Standard Track – CSP – parallel) over 236 instances.

<table>
<thead>
<tr>
<th>#solved</th>
<th>%inst.</th>
<th>%VBS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Best Solver (VBS)</td>
<td>113</td>
<td>53 SAT, 60 UNSAT</td>
</tr>
<tr>
<td>1 NACRE</td>
<td>86</td>
<td>43 SAT, 43 UNSAT</td>
</tr>
<tr>
<td>2 miniBTD_12</td>
<td>79</td>
<td>36 SAT, 43 UNSAT</td>
</tr>
<tr>
<td>3 miniBTD</td>
<td>75</td>
<td>32 SAT, 43 UNSAT</td>
</tr>
<tr>
<td>4 cosoco</td>
<td>72</td>
<td>42 SAT, 30 UNSAT</td>
</tr>
<tr>
<td>5 minimacht</td>
<td>69</td>
<td>37 SAT, 32 UNSAT</td>
</tr>
<tr>
<td>6 GG’s minicp</td>
<td>56</td>
<td>37 SAT, 19 UNSAT</td>
</tr>
<tr>
<td>7 Solver of Schul &amp; Smal</td>
<td>54</td>
<td>23 SAT, 31 UNSAT</td>
</tr>
<tr>
<td>8 MiniCPFever</td>
<td>54</td>
<td>34 SAT, 20 UNSAT</td>
</tr>
<tr>
<td>9 slowpoke</td>
<td>38</td>
<td>38 SAT</td>
</tr>
<tr>
<td>10 SuperSolver</td>
<td>31</td>
<td>31 SAT</td>
</tr>
<tr>
<td>11 The dodo solver</td>
<td>25</td>
<td>25 UNSAT</td>
</tr>
</tbody>
</table>

Table 4.7: Ranking for (Mini-Solver Track – CSP) over 176 instances.

<table>
<thead>
<tr>
<th>#solved</th>
<th>%inst.</th>
<th>%VBS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Best Solver (VBS)</td>
<td>48</td>
<td>48 OPT</td>
</tr>
<tr>
<td>1 cosoco</td>
<td>46 (122)</td>
<td>46 OPT</td>
</tr>
<tr>
<td>2 Solver of Schul &amp; Smal</td>
<td>35 (44)</td>
<td>35 OPT</td>
</tr>
<tr>
<td>3 GG’s minicp</td>
<td>3 (22)</td>
<td>3 OPT</td>
</tr>
<tr>
<td>4 MiniCPFever</td>
<td>0 (50)</td>
<td>0%</td>
</tr>
<tr>
<td>5 SuperSolver</td>
<td>0 (23)</td>
<td>0%</td>
</tr>
<tr>
<td>6 The dodo solver</td>
<td>0 (18)</td>
<td>0%</td>
</tr>
<tr>
<td>7 slowpoke</td>
<td>0 (12)</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 4.8: Ranking for (Mini-Solver Track – COP) over 188 instances. Between parentheses, the number of times the solver can give the best known result.
4.2. A FEW COMMENTS

**Eternity**  Mistral and Choco are the best competitors. 7 instances out of 15 remain unsolved.

**Frб.**  On these random instances from Model RB (and forced to be satisfiable), sCOP slightly outperforms Concrete, CoSoCo and OscaR. 9 instances out of 16 remain unsolved.

**Graceful Graph**  All solvers are close in term of performance, with Mistral succeeding in solving one more instance. 4 instances out of 11 remain unsolved.

**Haystack**  SAT-based solvers Picat-SAT and sCOP are the best competitors, solving all instances, followed by BTD. Learning and decomposition methods are quite operational for this problem.

**Langford**  OscaR is the best competitor, followed by Mistral, PicatSAT and sCOP. Note that CoSoCo can be very fast for solving some instances. Only one instance out of 11 remains unsolved.

**MagicHexagon**  Concrete, CoSoCo, and Mistral are the best competitors. Sat-Based solvers and BTD are not very effective for this problem. 5 instances out of 11 remain unsolved.

**MisteryShopper**  Mistral is the best competitor, followed by Choco and Concrete. All instances have been solved.

**PseudoBoolean**  sCOP is the best competitor, followed by Mistral. 5 instances out of 13 remain unsolved.

**QuasiGroups**  sCOP is the best competitor, followed by macht and PicatSAT. 7 instances out of 16 remain unsolved.

**RLFAP**  The most difficult instance from this classical (decision version of this) problem, is only solved by SAT-based solvers sCOP and PicatSAT as well as BTD. sCOP appears to be very robust and regular whereas BTD can be very efficient at solving some of these instances.

**SocialGolfers**  Compared to Choco, Concrete, Mistral and OscaR, SAT-based solvers PicatSAT and sCOP are able to solve 2 additional instances. Note how PicatSAT is very efficient for this problem. 4 instances out of 12 remain unsolved.

**SportsScheduling**  Oscar and Mistral obtain the best results. 5 instances out of 10 remain unsolved.

**StripPacking**  Sat-based solvers sCOP and PicatSAT are the best competitors.

**SubIsomorphism**  Choco, followed by BTD and Mistral. 3 instances out of 11 instances remain unsolved.

### 4.2.2 Standard Track – COP – Sequential

Note that Mistral is sometimes mentioned during analysis; although it has been disqualified for this track.
Auction  OscaR is clearly the best competitor for this problem, always giving the best bounds. Choco is the best second solver. There is no clear advantage of using the model variant 'cnt' (with atMost1) or the model variant 'sum'.

BACP  PicatSAT is the only solver able to prove optimality for 4 instances. On the other instances, solvers are rather close, almost always providing the same bounds. There is no clear advantage of using the model variant 'm1' or the model variant 'm2' that reformulates some constraints (to be compatible with the restrictions of the Mini Tracks).

CrosswordsDesign  This difficult optimization problem involves large short tables. For example, for \( n = 10 \), there are 20 tables containing 186,809 tuples and 20 other tables containing 143,417 tuples. CoSoCo is the best performer, succeeding in finding solutions for the most difficult instances. Concrete has also a good behavior on some instances of intermediary difficulty. Because of the huge size of some variable domains, Compact-Table [8] may be not the best table propagator.

FAPP  It is important to note that the model used for the competition exactly corresponds to the original and complete formulation of the problems used for the 2001 ROADEF challenge. Results obtained by teams are available at [http://www.roaef.org/challenge/2001/fr/resultats_phase1.php](http://www.roaef.org/challenge/2001/fr/resultats_phase1.php) On instance m2s-01-0200, Choco finds a solution with \( k = 4 \) (the main parameter related to some form of violation). On instance m2s-02-0250, Concrete finds a solution with \( k = 2 \). On instance m2s-03-0300, Choco and Concrete find a solution with \( k = 7 \). This corresponds to the best bounds found in 2001 by the best teams. Although it is rather unfair to compare results that are 17 years away, one can observe that a pure CP approach (benefiting from progress on table constraints) can be very effective (and note that no specific tuning is possible in the context of the XCSP competition). Note that optimality has been proved for 5 instances (and also for 2 easy ones); PicatSAT being the unique solver in proving optimality of instance test-01-0150.

GolombRuler  Concrete, CoSoCo, Mistral and OscaR are the best competitors. When decision variables are not provided (see instances with 'nodv' in their names), solvers are far less efficient in general.

GraphColoring  Concrete is the most robust solver for this problem, while CoSoCo is usually very fast.

Knapsack  Mistral and OscaR most often prove optimality. 4 instances out of 11 remains unsolved (to optimality).

LowAutocorrelation  PicatSAT is impressive on this problem (involving intensional and sum constraints): it proves optimality for 10 instances (out of 14). Mistral is also very efficient on this problem.

Mario  Surprisingly, four solvers (Choco, Concrete, Mistral and OScaR) have solved all 10 instances of the series.

NurseRostering  The results are interesting. While all solvers prove optimality on the two first instances, OscaR gives the best bound on the third instance, Sat4j gives the best bounds on the next 3 instances, Choco gives the best bounds on the next 6 instances, and finally CoSoCo gives the best bounds on the next 4 instances. This is rather curious.
4.2. A FEW COMMENTS

**PeacableArmies** On model variant m1, Oscar is the best competitor. On model variant m2, obtained bounds are usually worse; Sat4j and OscaR being dominating.

**PizzaVoucher** SAT-based solvers are very competitive for this problem. PicatSAT proves optimality on 7 instances (out of 10), and Sat4j provides the best bounds for the three most difficult instances.

**PseudoBoolean** Overall, solvers embedding SAT techniques (PicatSAT, and SAT4j) are good competitors for this problem, although there is no clear winner. CoSoCo and Choco also obtains good results.

**QuadraticAssignment** This is a classical problem. Clearly, OscaR is the best competitor, being ranked first 16 times (out of 19 instances). Note that SAT-based solvers have strong difficulties.

**RCPSP** This is another classical problem. OscaR is impressive here: proving rapidly optimality for 14 instances (out of 16 instances). Mistral, Choco and Concrete can also be very fast for many instances. Finally, PicatSAT is rather robust, succeeding in proving optimality for many instances.

**RLFAP** It is important to note that the model used for the competition exactly corresponds to the original and complete formulation of the problem (see [4]). CoSoCo is the most robust solver, being ranked first 13 times (out of 14+11 instances). Sat4j is also ranked first 7 times. Choco and Concrete also usually obtain interesting results.

**SteelMillSlab** Except for the very simple instances, no instances from models 'm1' and 'm2' have been solved at all. For model 'm2s', CoSoCo and Concrete obtain the best results.

**StillLife** PicatSAT is quite impressive here: proving optimality for all instances. All other solvers can do that for at most 3 instances.

**SumColoring** PicatSAT proves optimality for 5 instances. OscaR and Choco are rather robust.

**Tal** This is a problem of natural language processing. OscaR is the most robust solver. Choco also obtains good results.

**TemplateDesign** Instances of model variant 'm1s' are similar to those of model variant 'm1', except that symmetry breaking constraints are not present. Rather surprisingly, solver usually obtain better results without such constraints. The best competitor is Choco, followed by PicatSAT and Concrete.

**TravellingTournament** Choco obtains the best results, followed by OscaR and Concrete. Optimality has only been proved for two instances.

**TravellingSalesman** Optimality is proved for 3 instances by PicatSAT. CoSoCo and OscaR also obtain good results.
Bibliography


