XCSP3 Competition 2018
Proceedings

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

Preliminary Version – August 28, 2018

This document represent the proceedings of the XCSP3 Competition 2018. This is a preliminary version containing the description of problems (models) selected for the competition. Soon, it will also contain the description of the solvers and the results.

The website containing all **detailed results** is available at:

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

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.

Problem	Optimization	Constraints
Auction	max SUM	`count`, `sum`
BACP	min MAXIMUM	`intension`, `extension`, `count`, `sum`
BIBD		`sum`, `lexMatrix`
Car Sequencing		`extension`, `sum`, `cardinality`
Coloured Queens		`allDifferent`, `allDifferentMatrix`
Crosswords		`extension`
Crosswords Design	max SUM	`extension` (∗)
Dubois		`extension`
Eternity		`intension`, `extension`, `allDifferent`
FAPP	min SUM	`intension`, `extension`
FRB		`extension`
Golomb Ruler	min VAR	`intension`, `allDifferent`
Graceful Graph		`intension`, `allDifferent`
Graph Coloring	min MAXIMUM	`intension`
Haystacks		`extension`
Knapsack	max SUM	`sum`
Langford		`intension`, `element`
Low Autocorrelation	min SUM	`intension`, `sum`
Magic Hexagon		`intension`, `sum` and `allDifferent`
Magic Square		`allDifferent`, `sum`, `instantiation`
Mario	max SUM	`intension`, `extension`, `sum`, `circuit`
Mistery Shopper		`intension`, `extension`, `allDifferent`, `lexMatrix`, `channel`
Nurse Rostering	min SUM	`intension`, `extension`, `sum`, `count`, `regular`, `instantiation`, `slide`
Peacable Armies	max SUM	`intension`, `sum`, `count`
Pizza Voucher	min SUM	`intension`, `count`
Pseudo-Boolean	min SUM	`sum`
Quadratic Assignment	min SUM	`extension`, `allDifferent`
QuasiGroup		`intension`, `allDifferentMatrix`, `instantiation`, `element`
RCPSP	min VAR	`intension`, `cumulative`
RLFAP	min MAXIMUM / min NVALUES	`intension`, `instantiation`
Social Golfers		`intension`, `instantiation`, `cardinality`, `lexMatrix`
Sports Scheduling		`intension`, `extension`, `instantiation`, `allDifferent`, `count`, `cardinality`
Steel Mill Slab	min SUM	`intension`, `extension`, `ordered`, `sum`
Still Life	max VAR	`intension`, `extension`, `instantiation`, `sum`
Strip Packing		`intension`, `extension`, `noOverlap`
Subgraph Isomorphism		`extension`, `allDifferent`
Sum Coloring	min SUM	`intension`
TAL	min SUM	`intension`, `extension`, `count`
Template Design	min SUM	`intension`, `ordered`, `sum`
Traveling Tournament	min SUM	`intension`, `extension` (∗), `allDifferent`, `element`, `cardinality`, `regular`
Travelling Salesman	min SUM	`extension`, `allDifferent`

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, Peaceable 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, 17, 6, 16, 21] 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 [14], 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 subproblems 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 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 [13], which is the new version of MCSP3 to be officially released in October 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:

```
{
  "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:

```
{
  "nPeriods": 5,
  "minCredits": 6,
  "maxCredits": 15,
  "minCourses": 2,
  "maxCourses": 6,
  "credits": [2, 3, 2, 4, 1, 3, 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:

```
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");
    forall(range(nPrerequisites), i -> lessThan(s[prerequisites[i][0]],s[prerequisites[i][1]]))
      .note("handling prerequisites");

    minimize(MAXIMUM, cr)
      .note("minimizing the maximum number of credits in periods");

    decisionVariables(s);
  }
}
```

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 λ 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 λ between any pair of distinct rows. A BIBD is therefore specified by its parameters (v, b, r, k, λ)"

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:

```
class Bibd implements ProblemAPI {
  int v, b, r, k, lambda;

  public void model() {
    b = b != 0 ? b : (lambda * v * (v-1)) / (k * (k-1)); // when b is 0, we compute it
    r = r != 0 ? r : (lambda * (v-1)) / (k-1); // when r is 0, we compute it

    Var[][] x = array("x", size(v, b), dom(0, 1),
      "x[i][j] is the value at row i and column j of the matrix");

    forall(range(v), i -> sum(x[i], EQ, r))
      .note("constraints on rows");
    forall(range(b), j -> sum(columnOf(x, j), EQ, k))
      .note("constraints on columns");
    forall(range(v), i -> forall(range(i + 1, v), j -> sum(x[i], weightedBy(x[j]), EQ, lambda)))
      .note("scalar constraints with respect to lambda");
    lexMatrix(x, INCREASING).tag(SYMMETRY_BREAKING);
      .note("Increasingly ordering both rows and columns")
  }
}
```

This model involves 1 array of variables and 2 types of constraints: `sum` and `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.

2.4 Car Sequencing

This is Problem 001 on CSPLib.

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, sun-roof, 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."

Data

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

```
{
  "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, 1, 0, 0] },
    { "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 }
  ]
}
```

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);
  }
}
```

This model involves 2 arrays of variables and 3 types of constraints: `cardinality`, `extension` and `sum`. Note that instead of posting `extension` constraints, we could have used binary `intension` constraints with a predicate like $c_i = j \Rightarrow o_{i,k} = v$ where v is the value (0 or 1) of the kth option of the jth class. Also, note that we could have used a cache for the table built by `matchs()`. The last group of constraints corresponds to redundant constraints. A series of 17 instances has been selected for the competition.

2.5 Coloured Queens

Description

> The queens graph is a graph with n*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 [12].

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:

```
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:

```
{
  "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)");
    forall(range(nHoles), i -> extension(holes[i].scope(x), words.get(holes[i].size)))
      .note(”fill the grid with words”);
  }
}
```

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:

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

Model

The MCSP3 model used for the competition is:

```
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;

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

  @NotData
  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.add(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++) {
        int bp = p + words[i].length(); // position of the black point, just after the word
        if (bp <= n) {
          int[] tuple = new int[n + 4];
          tuple[0] = p;
          tuple[1] = i;
          tuple[2] = wordsPoints[i];
          if (lastWord && bp < n - 1)
            continue;
          tuple[3] = lastWord ? -1 : bp <= n - 2 ? bp + 1 : -1;
          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");
```

```
      maximize(SUM, vars(br, bc))
        .note("maximizing the summed benefit of words put in the grid");
  }
}
```

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:

```
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));
    forall(range(2), i -> extension(vars(x[n - 2 + i], x[3*n - 2], x[3 * n - 1]), table1));
    forall(range(n, 2*n - 2), i -> extension(vars(x[i], x[4*n - 3 - 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:

```
{
  "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:

```
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], tb = i < n - 1 ? top[i + 1][j] : bot[j];
      extension(vars(id[i][j], top[i][j], lr, tb, left[i][j]), piecesTable());
    }).note("pieces must be valid (i.e. correspond to those given initially, possibly after rotation)");
    block(() -> {
      forall(range(n), i -> equal(left[i][0], 0));
      forall(range(n), i -> equal(right[i], 0));
      forall(range(m), j -> equal(top[0][j], 0));
      forall(range(m), j -> equal(bot[j], 0));
    }).note("put special value 0 on borders");
  }
}
```

This model involves 5 arrays of variables and 3 types of constraints: `allDifferent`, `extension` and `intension` (primitive `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[1] that was part of the ROADEF'2001 challenge[2]. 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:
https://uma.ensta-paristech.fr/conf/roadef-2001-challenge/distrib/fapp_roadef01_rev2_tex.pdf

Data

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

```
{
  "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":
        [46,44,42,42,40,40,35,35,35,30,20],
      "neRelaxations": [39,37,35,35,33,33,28,28,28,23,13] },
    { "route1": 0, "route2": 8, "eqRelaxations":
        [30,29,28,27,26,25,24,23,22,21,20],
      "neRelaxations": [29,28,27,26,25,24,23,22,21,20,19] },
```

[1]This is an extended subject of the CALMA European project
[2]See `http://uma.ensta.fr/conf/roadef-2001-challenge/`

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

Model

The MCSP3 model used for the competition is:

```
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 after some reformulations) is compatible with the restrictions imposed for the mini-track.

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 [22], 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:

```
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");

    decisionVariables(x);
  }
}
```

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, [20].

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:

```
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:

```
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.

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:

```
{
  "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:

```
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 Langdford's number problem.

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 [9] for $k = 2$. The MCSP3 model used for the competition is:

```
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 kth 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 $E = \sum_{k=1}^{n-1} C_k^2$.

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:

```
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 sequence to be built.");
    Var[][] y = array("y", size(n - 1, n - 1), (k, i) -> dom(-1, 1).when(i < n - k - 1),
      "y[k][i] is the ith product value required to compute the kth autocorrelation");
    Var[] c = array("c", size(n - 1), k -> dom(range(-n + k + 1, n - k)),
      "c[k] is the value of the kth autocorrelation");
    Var[] s = array("s", size(n - 1), k -> dom(range(n - k).map(v -> v * v)),
      "s[k] is the square of the kth autocorrelation");

    forall(range(n - 1), k -> forall(range(n - k - 1), i -> equal(y[k][i], mul(x[i], x[i + k + 1]))))
      .note("computing product values");
    forall(range(n - 1), k -> sum(y[k], EQ, c[k]))
      .note("computing the values of the autocorrelations");
    forall(range(n - 1), k -> equal(s[k], mul(c[k], c[k])))
      .note("computing the squares of the autocorrelations");

    minimize(SUM, s)
      .note("minimizing the sum of the squares of the autocorrelation");
  }
}
```

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:

```
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), v2 = d / 2 - v1;
    Range r = range(d - Math.abs(d / 2 - i));
    return variablesFrom(r, j -> x[j + v1][i - Math.max(0, right ? v2 - j : j - v2)]);
  }

  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 - i)),
      "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");;
    forall(range(d), i -> sum(x[i], EQ, magic))
      .note("all rows sum to the magic value");
    forall(range(d), i -> sum(scopeForDiagonal(x, i, true), EQ, magic))
      .note("all right−sloping diagonals sum to the magic value");
    forall(range(d), i -> sum(scopeForDiagonal(x, i, false), EQ, magic))
      .note("all left−sloping diagonals sum to the magic value");

    block(() -> {
      lessThan(x[0][0], x[0][n - 1]);
      lessThan(x[0][0], x[n - 1][d - 1]);
      lessThan(x[0][0], x[d - 1][n - 1]);
      lessThan(x[0][0], x[d - 1][0]);
      lessThan(x[0][0], x[n - 1][0]);
      lessThan(x[0][n - 1], x[n - 1][0]);
    }).tag(SYMMETRY_BREAKING);
  }
}
```

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:

```
{
  "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:

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

or, equivalently:

```
{
  "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:

```
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 companys 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:

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

Model

The MCSP3 model used for the competition is:

```
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 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, [15]).

Description

The description is rather complex. Hence, we refer the reader to:
http://www.schedulingbenchmarks.org/instances1_24.html.

Data

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

```
{
  "nDays": 14,
  "shifts": [ { "id": "D", "length": 480, "forbiddenFollowingShifts": "null" } ],
  "staffs": [
    { "id": "A",
      "maxShifts": [14],
      "minTotalMinutes": 3360,"maxTotalMinutes": 4320,
      "minConsecutiveShifts": 2,"maxConsecutiveShifts": 5,
      "minConsecutiveDaysOff": 2,
      "maxWeekends": 1, "daysOff": [0],
      "onRequests": [
                { "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 nWeeks = nDays / 7, 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[][] ds = array("ds", size(nDays, nShifts), dom(range(nStaffs + 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]))
```

```
        .note("computing number of days");
      forall(range(nDays).range(nShifts), (d, s) -> exactly(x[d], takingValue(s), ds[d][s]))
        .note("computing number of persons");

      forall(range(nStaffs).range(nWeeks), (p, w) -> {
        implication(ne(x[w * 7 + 5][p], off), eq(wk[p][w], 1));
        implication(ne(x[w * 7 + 6][p], off), eq(wk[p][w], 1));
      }).note("computing worked week-ends");

      if (rotationTable().size() > 0)
        forall(range(nStaffs), p -> slide(columnOf(x, p), range(nDays - 1), i ->
          extension(vars(x[i][p], x[i + 1][p]), rotationTable())))
        .note("rotation shifts");

      forall(range(nStaffs), p -> sum(wk[p], LE, staffs[p].maxWeekends))
        .note("maximum number of worked week-ends");

      int[] lengths = valuesFrom(shifts, shift -> shift.length);
      forall(range(nStaffs), p ->
        sum(ps[p], weightedBy(lengths), IN, range(staffs[p].minTotalMinutes,staffs[p].maxTotalMinutes + 1)))
      .note("minimum and maximum number of total worked minutes");

      forall(range(nStaffs), p -> {
        int k = staffs[p].maxConsecutiveShifts;
        forall(range(nDays - k), i ->
          atLeast1(select(columnOf(x, p), range(i, i + k + 1)), takingValue(off)));
      }).note("maximum consecutive worked shifts");

      forall(range(nStaffs), p -> {
        int k = staffs[p].minConsecutiveShifts;
        forall(range(nDays - k), i ->
          regular(select(columnOf(x, p), range(i, i + k + 1)), automatonMinConsec(nShifts, k, true)));
      }).note("minimum consecutive worked shifts");

      forall(range(nStaffs), p -> {
        int k = staffs[p].minConsecutiveDaysOff;
        forall(range(nDays - k), i ->
          regular(select(columnOf(x, p), range(i, i + k + 1)), automatonMinConsec(nShifts, k, false)));
      }).note("minimum consecutive days off");

     forall(range(nStaffs), p -> {
        int k = staffs[p].minConsecutiveShifts;
        if (k > 1) {
          forall(range(1, k), i -> implication(ne(x[0][p], off), ne(x[i][p], off)));
          forall(range(1, k), i -> implication(ne(x[nDays - 1][p], off), ne(x[nDays - 1 - i][p], off)));
        }
      }).note("managing off days on schedule ends");

     forall(range(nStaffs).range(nDays), (p, d) -> {
        if (onRequest(p, d) != null)
          equivalence(eq(x[d][p], shiftPos(onRequest(p, d).shift)), eq(cn[p][d], 0));
        if (offRequest(p, d) != null)
          equivalence(eq(x[d][p], shiftPos(offRequest(p, d).shift)), ne(cf[p][d], 0));
      }).note("cost of not satisfying on and off requests");

      forall(range(nDays).range(nShifts), (d,s) -> extension(vars(ds[d][s],cc[d][s]), indexing(costs(d,s))))
        .note("cost of under/over covering");

      minimize(SUM, vars(cn, cf, cc));
    }
}
```

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 [19]."

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:

```
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 [19], 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 2×7 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:

```
{
  "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:

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

  class Voucher {
    int payPart;
    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), i -> implication(le(v[i], 0), ne(pp[i], 0)))
      .note("a pizza must be paid iff a free voucher part is not used to have it free");

    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:

```
{
  "n": 144,
  "e": 704,
  "ctrs": [
    { "coeffs": [1,1,1,1,1,1], "nums": [1,17,33,49,65,81], "op": "=", "limit": 1
        },
    { "coeffs": [1,1,1,1,1,1], "nums": [2,18,34,50,66,82], "op": "=", "limit": 1
        },
    ...
  ],
  "obj": {
    "coeffs": [1,2,1,1,1,2,2,1,1,2,1,1],
    "nums": [96,97,101,108,109,111,116,124,131,133,140,143]
  }
}
```

Model

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[i] 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.

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],
    [90, 0, 0, 0, 0, 88, 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:

```
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))
```

```
      .note("minimizing summed up distances multiplied by flows");
  }
}
```

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)));

      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)));
      forall(range(n).range(n), (i, j) -> {
        if (i != j) {
          element(columnOf(x, j), at(x[j][i]), takingValue(y[i][j]));
          element(x[i], at(x[j][i]), takingValue(y[i][j]));
        }
      });
    }
  }
}
```

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 2×8 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:

```
{
  "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:

```
{
  "domains": {
    "1": [16, 30, 44, 58, 72, 86, 100, 114, 128, 142, 156, 254, 268, ...],
    "2":[30, 58, 86, 114, 142, 268, 296, 324, 352, 380, 414, 442, 470, ...],
    ...
  },
  "vars": [
    { "number": 13, "domain": 1, "value": "null", "mobility": "null" },
    { "number": 14, "domain": 1, "value": "null", "mobility": "null" },
    ...
  ],
  "ctrs":[
    { "x": 13, "y": 14, "equality": true, "limit": 238, "weight": 0 },
    { "x": 13, "y": 16, "equality": false, "limit": 186, "weight": 0 },
    ...
  ],
  "interferenceCosts": [0, 1000, 100, 10, 1],
  "mobilityCosts": [0, 0, 0, 0, 0]
}
```

Model

The MCSP3 model used for the competition is:

```
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));
    int[] fixedVals = valuesFrom(indexes, index -> vars[index].value);
    instantiation(fixedVars, fixedVals).note("managing pre-assigned frequencies");
    forall(range(e), i -> {
      Var x = f[index(ctrs[i].x)], y = f[index(ctrs[i].y)];
      if (ctrs[i].equality)
        equal(dist(x, y), ctrs[i].limit);
      else
        greaterThan(dist(x, y), ctrs[i].limit);
    }).note("hard constraints on radio-links");

    if (modelVariant("span"))
      minimize(MAXIMUM, f)
        .note("minimizing the largest frequency");
    else if (modelVariant("card"))
      minimize(NVALUES, f)
        .note("minimizing the number of used 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 ($\texttt{nGroups} = 8, \texttt{groupSize} = 4, \texttt{nWeeks} = 6$):

Model

The MCSP3 model used for the competition is:

```
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");
    forall(range(nWeeks), w -> cardinality(x[w], allGroups, occursEachExactly(groupSize)));
      .note("respecting the size of the groups");

    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 $n1$ 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 matchs() {
    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[][] h = array("h", size(nPeriods, nWeeks), dom(allTeams),
      "h[p][w] is the number of the home opponent");
    Var[][] a = array("a", size(nPeriods, nWeeks), dom(allTeams),
      "a[p][w] is the number of the away opponent");
    Var[][] 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[] 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[] 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:

```
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");
      forall(range(nSlabs), s -> extension(vars(ld[s], ls[s]), indexing(possibleLosses)))
        .note("computing the loss of each slab");
      forall(range(nSlabs), s -> sum(colRep(allColors).map(g -> or(treesFrom(g, o -> eq(sb[o],s)))),LE,2))
        .note("no more than two colors for each slab");
    }
    if (modelVariant("m2")) {
      Var[][] y = array("y", size(nSlabs, nOrders), dom(0, 1),
        "y[s][o] is 1 iff the slab s is used to produce the order o");
      Var[][] z = array("z", size(nSlabs, nColors), dom(0, 1),
        "z[s][c] is 1 iff the slab s is used to produce an order of color c");

      forall(range(nSlabs).range(nOrders), (s, o) -> equivalence(eq(sb[o], s), eq(y[s][o], 1)))
        .note("linking variables sb and y");
      forall(range(nSlabs).range(nOrders), (s, o) -> {
        int c = Utilities.indexOf(orders[o].color, allColors);
        implication(eq(sb[o], s), eq(z[s][c], 1));
      }).note("linking variables sb and z");
      forall(range(nSlabs), s -> sum(y[s], weightedBy(sizes), EQ, ld[s]))
        .note("computing (and checking) the load of each slab");
      forall(range(nSlabs), s -> extension(vars(ld[s], ls[s]), indexing(possibleLosses)))
        .note("computing the loss of each slab");
      forall(range(nSlabs), s -> sum(z[s], LE, 2))
        .note("no more than two colors for each slab");
    }

    sum(ld, EQ, totalSize)
      .tag(REDUNDANT_CONSTRAINTS);

    block(() -> {
      decreasing(load);
      forall(range(nOrders).range(nOrders), (i, j) -> {
        if (i < j && orders[i].size == orders[j].size && orders[i].color == orders[j].color)
          lessEqual(sb[i], sb[j]);
      });
    }).tag(SYMMETRY_BREAKING);

    minimize(SUM, ls)
      .note("minimizing summed up loss");
  }
}
```

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`). Noe 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:

```
class StillLife implements ProblemAPI {
  int n;

  @NotData
  private Predicate<int[]> p = x -> {
    int s1 = x[0] + x[1] + x[2] + x[3] + x[5] + x[6] + x[7] + x[8];
    int s2 = x[0] * x[2] + x[2] * x[8] + x[8] * x[6] + x[6] * x[0] + x[1] + x[3] + x[5] + x[7];
    int s3 = x[1] + x[3] + x[5] + x[7];
    return (x[4] != 1 || s1 >= 2) && (x[4] != 1 || s1 <= 3) && (x[4] != 0 || s1 != 3)
      && (x[4] != 1 || s2 > 1 || x[9] >= 1) && (x[4] != 1 || s2 > 0 || x[9] >= 2)
      && (x[4] != 0 || s3 < 4 || x[9] >= 2) && (x[4] != 0 || s3 > 1 || x[9] >= 1)
      && (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);
      slide(x[1], range(n), j -> extension(vars(x[1][j], x[1][j + 1], x[1][j + 2]), conflicts));
      slide(x[n], range(n), j -> extension(vars(x[n][j], x[n][j + 1], x[n][j + 2]), conflicts));
      slide(columnOf(x, 1), range(n), i -> extension(vars(x[i][1], x[i + 1][1], x[i + 2][1]), conflicts));
      slide(columnOf(x, n), range(n), i -> extension(vars(x[i][n], x[i + 1][n], x[i + 2][n]), conflicts));
    }).note("ensuring that cells at the border remain dead");

    int[][] tuples = allCartesian(vals(2, 2, 2, 2, 2, 2, 2, 2, 2, 3), p);
    forall(range(1, n + 1).range(1, n + 1), (i, j) -> {
      Var[] neighbors = select(x, range(i - 1, i + 2).range(j - 1, j + 2));
      extension(vars(neighbors, w[i][j]), tuples);
    }).note("still life + wastage constraints");

    block(() -> {
      forall(range(1, n + 1), j -> equal(add(w[0][j], x[1][j]), 1));
      forall(range(1, n + 1), j -> equal(add(w[n + 1][j], x[n][j]), 1));
      forall(range(1, n + 1), i -> equal(add(w[i][0], x[i][1]), 1));
      forall(range(1, n + 1), i -> equal(add(w[i][n + 1], x[i][n]), 1));
    }).note("managing wastage on the border");

    forall(range(n + 2), i -> sum(i == 0 ? w[0] : vars(ws[i - 1], w[i]), EQ, ws[i]))
      .note("summing wastage");
    sum(vars(z, ws[n + 1]), vals(4, 1), EQ, 2 * n * n + 4 * n)
      .note("setting the value of the objective");
    forall(range(n + 1), i -> greaterEqual(sub(ws[n + 1], ws[i]), 2 * ((n - i) / 3) + n / 3))
      .tag(REDUNDANT_CONSTRAINTS);

    maximize(z)
      .note("maximizing the number of alive cells");
  }
}
```

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 [10]. 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:

```
{
  "container": { "width": 20, "height": 20 },
  "rectangles":[
    { "width": 2, "height": 12 },
    { "width": 7, "height": 12 },
        ...
      ]
    }
```

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:

```
{
  "nPatternNodes": 180,
  "nTargetNodes": 200,
  "patternEdges": [ [0,1], [0,3], [0,17], ... ],
  "targetEdges": [ [0,34], [0,65], [0,129], ...]
}
```

Model

The MCSP3 model used for the competition is:

```
class Subisomorphism implements ProblemAPI {
  int nPatternNodes, nTargetNodes;
  int[][] patternEdges, targetEdges;

  private int[] selfLoops(int[][] edges) {
    return Stream.of(edges).filter(t -> t[0] == t[1]).mapToInt(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(nPatternNodes).map(i -> degree(patternEdges, i));
    int[] tDegrees = range(nTargetNodes).map(i -> degree(targetEdges, i));
    int l = pLoops.length, e = patternEdges.length;

    Var[] x = array("x", size(nPatternNodes), dom(range(nTargetNodes)),
      "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(nPatternNodes), i -> {
      int[] conflicts = range(nTargetNodes).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:

```
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 [11].

Data

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

```
{
  "maxArity": 7,
  "maxHeight": -1,
  "sentence": [15, 11, 13, 9, 1, 11, 7, 4],
  "grammar": [
    null,
    [ [0,0,2147483646,0], [1,2,13,0], [1,2,26,16], ... ],
    [ [0,0,2147483646,2147483646,0], [1,2,13,2147483646,0], ... ],
    [ [0,0,2147483646,2147483646,2147483646,2147483646,0], ... ],
  ],
  "tokens": ["AP", "COORD", "NP", "PP", "SENT", ..., "VPP", "VPR", "VS"],
  "costs": [0, 1, 2, 4, 8, 16]
}
```

```
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) : dom(0), "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)) : dom(0), "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)) : dom(0), "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])) : dom(0), "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<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[0][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");
    forall(range(1, nLevels, 2), p -> greaterThan(a[p][0], 0));
    forall(range(1, nLevels, 2).range(1, nWords), (i, j) -> {
      if (j < lengths[i] && j + maxArity > lengths[i - 1])
        lessEqual(a[i][j], lengths[i - 1] - j);
    });

    forall(range(2, nLevels, 2), i -> {
      equal(x[i][0], 0);
      equal(l[i][0], l[i - 1][0]);
    });

    forall(range(2, nLevels, 2), i -> forall(range(1, lengths[i]), j -> {
      greaterEqual(x[i][j], j);
      implication(eq(l[i][j], 0), eq(x[i][j], lengths[i] - 1));
      implication(gt(l[i][j], 0), gt(x[i][j], x[i][j - 1]));
      implication(eq(l[i][j - 1], 0), eq(l[i][j], 0));
      Var[] vect = select(l[i - 1], k -> k < lengths[i - 1]);
      extension(vars(x[i][j], vect, l[i][j]), tableFor(vect.length));
    }));

    forall(range(1, nLevels, 2), i -> forall(range(lengths[i]), j -> {
      equivalence(eq(l[i][j], 0), eq(a[i][j], 0));
      int nPossibleSons = Math.min(lengths[i - 1] - j, maxArity);
      Var[] scp = vars(a[i][j], l[i][j], select(l[i - 1], range(j, j + nPossibleSons)), c[i][j]);
      extension(scp, grammar[nPossibleSons]);
    }));

    forall(range(0, nLevels, 2), i -> forall(range(lengths[i]), j -> predicate(l, a, i, j, lengths)));

    if (0 < maxHeight && 2 * maxHeight < l.length)
      equal(l[2 * maxHeight][1], 0);

    minimize(SUM, vars(c))
      .note("minimizing summed up cost");
  }
}
```

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 [18].

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

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:

```
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");
    block(() -> {
      decreasing(p);
      forall(range(nTemplates), t -> equivalence(eq(u[t], 0), eq(d[t][0], nSlots)));
    }).tag(SYMMETRY_BREAKING);

    minimize(SUM, u)
      .note("minimizing the number of used templates");
  }
}
```

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 [8].

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:

```
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");
    forall(range(nTeams), i -> {
      extension(vars(h[i][0], o[i][0], t[i][0]), tableEnd(i)));
      extension(vars(h[i][nRounds-1], o[i][nRounds-1], t[i][nRounds]), tableEnd(i)));
    }.note("handling travelling for the first and last games");

    forall(range(nTeams).range(nRounds - 1), (i, k) ->
      extension(vars(h[i][k], h[i][k + 1], o[i][k], o[i][k + 1], t[i][k + 1]), tableOther(i)))
    .note("handling travelling for two successive games");

    forall(range(nRounds), k -> allDifferent(columnOf(o, k))).tag(REDUNDANT_CONSTRAINTS)
      .note("at each round, opponents are all different");
    lessThan(o[0][0], o[0][nRounds - 1]).tag(SYMMETRY_BREAKING);

    minimize(SUM, t)
      .note("minimizing summed up travelled distance");
  }
}
```

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:

```
{
  "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:

```
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, the description of all solvers having participated to the XCSP3 Competition 2018 are given.

This chapter will be completed soon

Chapter 4

Results

In this chapter, the description of the results for the XCSP3 Competition 2018 are given.

This chapter will be completed soon. Meanwhile, see all detailed results on http://www.cril.fr/XCSP18/

Bibliography

[1] O. Akgun, I. Gent, C. Jefferson, I. Miguel, P. Nightingale, and A. Salamon. Automatic discovery and exploitation of promising subproblems for tabulation. In *Proceedings of CP'18*, 2018.

[2] E. Bourreau and T. Benoist. Fast global filtering for eternity ii. *Constraint Programming Letters*, 3:36–49, 2008.

[3] F. Boussemart, C. Lecoutre, G. Audemard, and C. Piette. XCSP3: An integrated format for benchmarking combinatorial constrained problems. Technical Report arXiv:1611.03398, Specifications 3.0.5, CoRR, 2016-2017. Available from http://www.xcsp.org/format3.pdf.

[4] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio Link Frequency Assignment. *Constraints*, 4(1):79–89, 1999.

[5] Kenil C. K. Cheng and Roland H. C. Yap. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2):265–304, 2010.

[6] A. Cire D. Bergman and, W. van Hoeve, and J. Hooker. *Decision diagrams for Optimization*. Springer, 2016.

[7] J. Dekker, G. Bjordal, M. Carlsson, P. Flener, and J.-N. Monette. Auto-tabling for subproblem presolving in minizinc. *Constraints*, 22(4):512–529, 2017.

[8] K. Easton, G. Nemhauser, and M. Trick. Solving the travelling tournament problem: A combined integer programming and constraint programming approach. In *Proceedings of PATAT'02*, pages 100–112, 2002.

[9] I.P. Gent, C. Jefferson, and I. Miguel. Watched literals for constraint propagation in minion. In *Proceedings of CP'06*, pages 182–197, 2006.

[10] E. Hopper and B. Turton. An empirical investigation of meta-heuristic and heuristic algorithms for a 2d packing problem. *European Journal of Operational Researc*, 128(1):34–57, 2001.

[11] R. Coletta J. Prost and C. Lecoutre. Compilation de grammaire de propriétés pour l'analyse syntaxique par optimisation de contraintes. In *Actes de* **TALN'16**, pages 396–402, Paris, France, 2016.

[12] T. Kelsey, S. Linton, and C.M. Roney-Dougal. New developments in symmetry breaking in search using computational group theory. In *Proceedings of AISC'04*, pages 199–210, 2004.

[13] C. Lecoutre. MCSP3: Easy modeling for everybody (version 1.1). Technical report, To appear in October, 2018.

[14] Jean-Baptiste Mairy, Yves Deville, and Christophe Lecoutre. The smart table constraint. In *Proceedings of CPAIOR'15*, pages 271–287, 2015.

[15] J.-P. Mtivier, P. Boizumault, and S. Loudni. Solving nurse rostering problems using soft global constraints. In *Proceedings of CP'09*, pages 73–87, 2009.

[16] G. Perez. *Decision diagrams : constraints and algorithms*. PhD thesis, University of Cte d'Azur, 2017.

[17] Guillaume Perez and Jean-Charles Régin. Improving GAC-4 for Table and MDD constraints. In *Proceedings of CP'14*, pages 606–621, 2014.

[18] L. Proll and B. Smith. Integer linear programming and constraint programming approaches to a template design problem. *INFORMS Journal on Computing*, 10(3):265–275, 1998.

[19] B. Smith, K. Petrie, and I. Gent. Models and symmetry breaking for 'peaceable armies of queens'. In *Proceedings of CPAIOR'04*, pages 271–286, 2004.

[20] B. Smith and J.-F. Puget. Constraint models for graceful graphs. *Constraints*, 15(1):64–92, 2010.

[21] H. Verhaeghe, C. Lecoutre, and P. Schaus. Compact-MDD: efficiently filtering (s)MDD constraints with reversible sparse bit-sets. In *Proceedings of IJCAI'18*, pages 1383–1389, 2018.

[22] K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. Random constraint satisfaction: easy generation of hard (satisfiable) instances. *Artificial Intelligence*, 171(8-9):514–534, 2007.