XCSP3 Competition 2018
Proceedings

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

December 3, 2018

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

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

Contents

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, 18, 6, 17, 22] for that constraint, but note that `regular` is quite related to `mdd`. The constraint `allEqual` is basically an ease of modeling, because it trivially corresponds to a set of binary equality constraints. A related form of the constraint `nValues` is indirectly represented in some RLFAP instances where the type of the objective is NVALUES. Similarly, a related form of `maximum` is indirectly represented in 3 problems where the type of the objective is MAXIMUM. On the other hand, in many problems, one can observe the presence of `intension` and `extension`. The frequent occurrence of `intension` is quite natural since in many problems a few primitives (e.g., like $x < y$) are required. The frequent occurrence of `extension` can be explained by the usefulness of that constraint. Sometimes, ordinary and short tables simply happen to be simple and natural choices for dealing with tricky situations. This is the case when no known (global) constraint exists or when converting a logical combination of (small) constraints into a table is needed for filtering efficiency reasons. Basically, table constraints offer the user a direct way to handle disjunction (a choice between tuples), and this is clearly emphasized with smart tables [15], which could be introduced in next editions (possibly, in a special track). Another argument showing the importance of universal structures like tables, and also MDDs (Multi-valued Decision Diagrams), is the rising of tabulation techniques, i.e., the the process of converting sub-problems into tables (or MDDs), by hand, by means of heuristics [1] or by annotations [7].

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

Selection. This year, the selection of problems and instances has been performed by Christophe Lecoutre. As a consequence, the solver AbsCon didn't enter the competition.

Chapter 2

Problems and Models

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

2.1 Auction

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

Description (from Patrick Prosser on CSPLib)

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

Data

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

```
{
  "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 [13].

Data

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

Model

The MCSP3 model used for the competition is:

```
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 [23], while guaranteeing satisfiability.

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

2.12 Golomb Ruler

This is Problem 006 on CSPLib, called Golomb Ruler.

Description (from Peter van Beek on CSPLib)

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

Data

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

Model

The MCSP3 model used for the competition is:

```
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, [21].

Description (from Karen Petrie on CSPLib)

"A labelling f of the nodes of a graph with q edges is graceful if f assigns each node a unique label from $\{0, 1, \dots q\}$ 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 $1, 2, \dots, q$.)"

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 [10] 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, [16]).

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 [20]."

Data

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

Model

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

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

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

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

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

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

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

      forall(range(n).range(n).range(n).range(n), (i1, j1, i2, j2) -> {
        if (i1 < i2 || (i1 == i2 && j1 < j2))
          if (i1 == i2 || j1 == j2 || Math.abs(i1 - i2) == Math.abs(j1 - j2))
            different(add(x[i1][j1], x[i2][j2]), 3);
      }).note("No two opponent queens can attack each other");

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

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

Following [20], two model variants, called 'm1' and 'm2' have been written. The first variant model involves 2 arrays of variables and 2 types of constraints: `sum` and `intension` (`lessEqual`). The second variant model involves 1 array of variables, 2 stand-alone variables and 2 types of constraints: `count` and `intension` (`equal` and `different`). A series of 2×7 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 [11]. See also the OR-library.

Description

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

Data

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

```
{
  "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 [12].

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 [19].

Description (from Barbara Smith on CSPLib)

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

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 [9].

Description

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

Data

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

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

Model

The MCSP3 model used for the competition is:

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

- *BTD, miniBTD* (Philippe Jegou, Helene Kanso and Cyril Terrioux)
- *BTD_12, miniBTD_12* (Philippe Jegou, Djamal Habet, Helene Kanso and Cyril Terrioux)
- *Choco-solver* (Charles Prud'homme and Jean-Guillaume Fages)
- *Concrete* (Julien Vion)
- *CoSoCo* (Gilles Audemard)
- GG's minicp (Arnaud Gellens and Simon Gustin)
- *macht, minimacht* (Djamal Habet and Cyril Terrioux)
- MiniCPFever (Victor Joos and Antoine Vanderschueren)
- *Mistral-2.0* (Emmanuel Hebrard and Mohamed Siala)
- *NACRE* (Gal Glorian)
- *OscaR* (OscaR Team)
- *PicatSAT* (Neng-Fa Zhou and Hakan Kjellerstrand)
- *Sat4j-CSP* (Daniel Le Berre and Emmanuel Lonca)
- *scop* (Takehide Soh, Daniel Le Berre, Mutsunori Banbara, Naoyuki Tamura)
- slowpoke (Alexandre Gerlache and vincent vandervilt)
- Solver of Xavier Schul and Yvhan Smal (Xavier Schul and Yvhan Smal)
- SuperSolver (Florian Stevenart Meeus and Jean-Baptiste Macq)
- The dodo solver (Alexandre Dubray)

BTD and miniBTD

Philippe Jégou[1], Hélène Kanso[1,2], and Cyril Terrioux[1]

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

1 Solver description

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

For the competition, we have made the following choices:

- the tree-decompositions are computed thanks to the heuristic H_5-TD-WT [2]
- *dom/wdeg* [1] is exploited for ordering the variables inside a cluster,
- *lexico* is used as value ordering heuristic,
- generalized arc-consistency is enforced by a propagation-based system exploiting events,
- restarts are performed according to a geometric restart policy based on the number of backtrack with an initial cutoff set to 100 and an increasing factor set to 1.1,
- the first root cluster is the cluster having the maximum ratio number of constraints to its size minus one and then, at each restart, the selected root cluster is one which maximizes the sum of the weights of the constraints whose scope intersects the cluster.

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

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

2 Command line

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

```
SOLVER 3 TIMELIMIT BENCHNAME
```

where:

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

Acknowledgments

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

References

1. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: ECAI. pp. 146–150 (2004)
2. Jégou, P., Kanso, H., Terrioux, C.: Towards a Dynamic Decomposition of CSPs with Separators of Bounded Size. In: CP. pp. 298–315 (2016)
3. Robertson, N., Seymour, P.: Graph minors II: Algorithmic aspects of treewidth. Algorithms 7, 309–322 (1986)

BTD_12 and miniBTD_12

Djamal Habet[1], Philippe Jégou[1], Hélène Kanso[1,2], and Cyril Terrioux[1]

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

1 Solver description

BTD_12 and miniBTD_12 are written in C++ and both implement the algorithm BTD-MAC+RST+Merge [1]. This algorithm exploits the structure of CSP instances thanks to the notion of tree-decomposition [3].

For the competition, we have made the following choices:

- the tree-decompositions are computed thanks to the heuristic H_5-TD-WT [1]
- an heuristic based on the exponential recency weighted average [2,4] is exploited for ordering the variables inside a cluster,
- *lexico* is used as value ordering heuristic,
- generalized arc-consistency is enforced by a propagation-based system exploiting events,
- restarts are performed according to a geometric restart policy based on the number of backtrack with an initial cutoff set to 100 and an increasing factor set to 1.1,
- the first root cluster is the cluster having the maximum ratio number of constraints to its size minus one and then, at each restart, the selected root cluster is one which maximizes the sum of the weights of the constraints whose scope intersects the cluster.

Note that, at now, BTD_12 only takes into account the following constraints:

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

2 Command line

BTD_12 and miniBTD_12 can be launched thanks to the following command line:

```
SOLVER 12 TIMELIMIT BENCHNAME
```

where:

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

Acknowledgments

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

References

1. Jégou, P., Kanso, H., Terrioux, C.: Towards a Dynamic Decomposition of CSPs with Separators of Bounded Size. In: CP. pp. 298–315 (2016)
2. Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Exponential recency weighted average branching heuristic for sat solvers. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence. pp. 3434–3440. AAAI Press (2016)
3. Robertson, N., Seymour, P.: Graph minors II: Algorithmic aspects of treewidth. Algorithms 7, 309–322 (1986)
4. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA, USA, 1st edn. (1998)

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

Charles Prud'homme[1] and Jean-Guillaume Fages[2]

[1] IMT-Atlantique, France,
Charles.Prudhomme@imt-atlantique.fr
[2] COSLING S.A.S., France,
jg.fages@cosling.com

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

Keywords: constraint programming, solver

1 Introduction

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

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

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

2 A Modeling API

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

Up to 100 constraints –and more than 150 propagators– are provided : from classic ones, such as arithmetical constraints, to must-have global constraints, such as *allDifferent* or *cumulative*, and include less common even though useful ones, such as *tree*. One can pick some existing propagators to compose a new constraint or create its own one in a straightforward way by implementing a filtering algorithm and a satisfaction checker.

The library supports natively real variables and constraints also, and relies on Ibex [3] to solve the continuous part of the problem [4]. Graph variables and constraints on them can be declared by adding a dependency to `choco-graph` [6].

3 Resolution Toolbox

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

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

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

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

4 The code and the dev team

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

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

References

1. David Allouche, Simon De Givry, Georgios KATSIRELOS, Thomas Schiex, and Matthias Zytnicki. Anytime hybrid best-first search with tree decomposition for weighted CSP. In *CP 2015 - 21st International Conference on Principles and Practice of Constraint Programming*, page 17 p., Cork, Ireland, August 2015.
2. Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 146–150, 2004.
3. Gilles Chabert. Ibex 2.2.0, June 2017.

4. Jean-Guillaume Fages, Gilles Chabert, and Charles Prud'Homme. Combining finite and continuous solvers Towards a simpler solver maintenance. In *The 19th International Conference on Principles and Practice of Constraint Programming*, page TRICS'13 Workshop: Techniques foR Implementing Constraint programming Systems, Uppsala, Sweden, September 2013.
5. Jean-Guillaume Fages and Charles Prud'Homme. Making the first solution good! In *ICTAI 2017 29th IEEE International Conference on Tools with Artificial Intelligence*, Boston, MA, United States, November 2017.
6. Jean-Guillaume Fages, Charles Prud'homme, and Xavier Lorca. *Choco-Graph : a module for graph variables in Choco Solver (version 4.0.0)*. COSLING S.A.S. and TASC, LS2N CNRS UMR 6241, 2017.
7. Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'79, pages 356–364, San Francisco, CA, USA, 1979. Morgan Kaufmann Publishers Inc.
8. William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'95, pages 607–613, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.
9. Narendra Jussien and Olivier Lhomme. Unifying search algorithms for csp. Research report RR0203, EMN, 2002.
10. Laurent Michel and Pascal Van Hentenryck. Activity-based search for black-box constraint programming solvers. In Nicolas Beldiceanu, Narendra Jussien, and Éric Pinson, editors, *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems*, pages 228–243, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
11. Franǫis Laburthe. Choco: implementing a cp kernel. In *Proceedings of Techniques foR Implementing Constraint programming Systems (TRICS'00)*, pages 118–133, 2000.
12. Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC, LS2N CNRS UMR 6241, COSLING S.A.S., 2017.
13. Philippe Refalo. Impact-based search strategies for constraint programming. In Mark Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, pages 557–571, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
14. Christian Schulte and Guido Tack. *Views and Iterators for Generic Constraint Implementations*, pages 118–132. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
15. Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. pages 417–431. Springer-Verlag, 1998.
16. Toby Walsh. Depth-bounded discrepancy search. In *In Proceedings of IJCAI-97*, pages 1388–1393, 1997.

Concrete 3.9.2: A CSP solving software & API

Julien Vion
`http://github.com/concrete-cp/concrete`

1 Features

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

- the use of persistent data structures [16] for managing domain states and some constraint states. We use bit vectors copied on-the fly, hash tries, trees with a high branching factor, and red-black trees. For the state of many constraints, semi-persistent data structures (mainly sparse sets [4]) or backtrack-stable data (watched literals [6] or residues [9]) are preferred.
- the use of the companion project CSPOM 2.25 [19], a solver-independent modeling assistant able to perform automatic reformulation such as constraint aggregation. CSPOM is able to parse problems written in FlatZinc [13], XCSP3 [1], the legacy XCSP2 format or its own Java and Scala DSL (yet to be documented).

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

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

Concrete natively supports the following constraints:

- Extension (list of allowed or forbidden tuples). An optimized algorithm should be automatically selected for binary constraints (AC3-bit+rm) [11] or MDD [20].
- Linear ($a \cdot x + b \cdot y + \ldots \ \{= / < / \leq / \neq\} \ k$). Bound consistency (except for $\neq$) [7] or full domain consistency for ternary constraints (using residues).

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

All FlatZinc constraints are supported, and other documented MiniZinc constraints are supported *via* provided decomposition or reformulation. All XCSP3 constraints selected for the 2018 competition are supported *via* (trivial) decomposition or reformulation. Some XCSP3 constraints may not be supported.

2 Search strategies

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

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

3 Present and near-future of Concrete

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

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

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

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

References

[1] G. Audemard, F. Boussemart, C. Lecoutre, C. Piette, et al. *XCSP3*. `http://www.xcsp.org`. 2016.

[2] F. Boussemart, F. Hemery, C. Lecoutre, and L. Saïs. “Boosting Systematic Search by Weighting Constraints”. In: *Proc. 16th ECAI.* 2004, pp. 146–150.

[3] F. Boussemart, F. Hemery, and C. Lecoutre. “Revision Ordering Heuristics for the CSP”. In: *Proc. CPAI’04 workshop held with CP’04.* Toronto, Canada, 2004, pp. 29–43.

[4] P. Briggs and L. Torczon. “An Efficient Representation for Sparse Sets”. In: *ACM Letters on Programming Languages and Systems* 2.1–4 (1993), pp. 59–69.

[5] J.-G. Fages and C. Prud’Homme. “Making the first solution good!” In: *Proc. 29th ICTAI*. Boston, MA, United States, Nov. 2017.

[6] I. P. Gent, C. Jefferson, and I. Miguel. “Watched literals for constraint propagation in Minion”. In: *Proc. 12th CP*. 2006, pp. 182–197.

[7] W. Harvey and J. Schimpf. “Bounds Consistency Techniques for Long Linear Constraints”. In: *In Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems*. 2002, pp. 39–46.

[8] J. Hwang and D.G. Mitchell. “2-way vs d-way branching for CSP”. In: *Proc. 11th CP*. 2005, pp. 343–357.

[9] C. Lecoutre and F. Hemery. “A Study of Residual Supports in Arc Consistency”. In: *Proc. 20th IJCAI*. 2007, pp. 125–130.

[10] C. Lecoutre, L. Saïs, S. Tabary, and V. Vidal. “Nogood Recording from Restarts”. In: *Proc. 20th IJCAI*. 2007.

[11] C. Lecoutre and J. Vion. “Enforcing AC using Bitwise Operations”. In: *Constraint Programming Letters* 2 (2008), pp. 21–35.

[12] A. López-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek. “A fast and simple algorithm for bounds consistency of the alldifferent constraint”. In: *Proc. 18th IJCAI*. 2003, pp. 245–250.

[13] N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack. “Minizinc: Towards a standard CP modelling language”. In: *Proc. 13th CP*. Ed. by C. Bessière. 2007, pp. 529–543.

[14] P. Nightingale, Ö. Akgün, I. P. Gent, C. Jefferson, I. Miguel, and P. Spracklen. “Automatically Improving Constraint Models in Savile Row”. In: *Artificial Intelligence* 251.Supplement C (2017), pp. 35–61.

[15] M. Odersky et al. *The Scala Programming Language*. `http://www.scala-lang.org/`. 2001.

[16] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[17] P. Shaw. “A Constraint for Bin Packing”. In: *Proc. 10th CP*. Ed. by M. Wallace. 2004, pp. 648–662.

[18] R.M. Stallman. *GNU Lesser General Public License*. GNU Project–Free Software Foundation, http://gnu.org/licenses. 1999.

[19] J. Vion. *CSP Object Model*. `http://github.com/concrete-cp/cspom`. 2008–2016.

[20] J. Vion and S. Piechowiak. “From MDD to BDD and Arc consistency”. In: *Constraints* (July 2018). DOI: `10.1007/s10601-018-9286-5`.

[21] J. Vion and S. Piechowiak. “Handling Heterogeneous Constraints in Revision Ordering Heuristics”. In: *Proc. of the TRICS’2010 workshop held in conjunction with CP’2010*. 2010, pp. 62–82.

[22] J. Vion and S. Piechowiak. “Une simple heuristique pour rapprocher DFS et LNS pour les COP”. In: *Actes des 13^e JFPC*. 2017, pp. 39–44.

CoSoCo 1.12

XCSP3 Competition 2018

Gilles Audemard

CRIL-CNRS, UMR 8188
Université d'Artois, F-62307 Lens France
`audemard@cril.fr`

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

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

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

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

Acknowledgements

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

References

1. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.
2. F. Boussemart, C. Lecoutre, and C. Piette. XCSP3: an integrated format for benchmarking combinatorial constrained problems. *CoRR*, abs/1611.03398, 2016.

3. C. Lecoutre. *Constraint networks: techniques and algorithms*. ISTE/Wiley, 2009.
4. C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Reasonning from last conflict(s) in constraint programming. *Artificial Intelligence*, 173(18):1592–1614, 2009.
5. J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19:229–250, 1979.

macht and minimacht

Djamal Habet and Cyril Terrioux

Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France
{djamal.habet, cyril.terrioux}@lis-lab.fr

1 Solver description

macht and minimacht are written in C++ and both implement the algorithm MAC+RST+NG [1].
For the competition, we have made the following choices:

- an heuristic based on the exponential recency weighted average [2,3] is exploited for ordering the variables,
- *lexico* is used as value ordering heuristic,
- generalized arc-consistency is enforced by a propagation-based system exploiting events,
- restarts are performed according to a geometric restart policy based on the number of backtrack with an initial cutoff set to 100 and an increasing factor set to 1.1,

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

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

2 Command line

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

```
SOLVER TIMELIMIT BENCHNAME
```

where:

- `SOLVER` is the path to the executable macht or minimacht,
- `TIMELIMIT` is the number of seconds allowed for solving the instance,
- `BENCHNAME` is the name of the XML file representing the instance we want to solve.

Acknowledgments

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

References

1. Lecoutre, C., Saïs, L., Tabary, S., Vidal, V.: Recording and Minimizing Nogoods from Restarts. JSAT 1(3-4), 147–167 (2007)
2. Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Exponential recency weighted average branching heuristic for sat solvers. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence. pp. 3434–3440. AAAI Press (2016)
3. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA, USA, 1st edn. (1998)

Mistral 2.0

XCSP3 Competition 2017

Emmanuel Hebrard[1] and Mohamed Siala[2]

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

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

Solver Engine

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

Propagators

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

Search Strategy

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

Applications of Mistral

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

References

1. Christian Artigues, Emmanuel Hebrard, Valentin Mayer-Eichberger, Mohamed Siala, and Toby Walsh. SAT and hybrid models of the car sequencing problem. In *Proceedings of the 11th International Conference on Integration of AI and OR Techniques in Constraint Programming – CPAIOR*, pages 268–283, 2014.
2. Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence – ECAI*, pages 146–150, 2004.
3. Clément Carbonnel and Emmanuel Hebrard. Propagation via kernelization: The vertex cover constraint. In *Proceedings of the 22nd International Conference on Principles and Practice of Constraint Programming – CP*, pages 147–156, 2016.
4. Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, and Toby Walsh. Propagation algorithms for lexicographic ordering constraints. *Artif. Intell.*, 170(10):803–834, 2006.
5. Diarmuid Grimes and Emmanuel Hebrard. Solving variants of the job shop scheduling problem through conflict-directed search. *INFORMS Journal on Computing*, 27(2):268–284, 2015.
6. Emmanuel Hebrard, Eoin O'Mahony, and Barry O'Sullivan. Constraint programming and combinatorial optimisation in numberjack. In *Proceedings of the 7th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems – CPAIOR*, pages 181–185, 2010.
7. Emmanuel Hebrard and Mohamed Siala. Explanation-based weighted degree. In *Proceedings of the 14th International Conference on Integration of AI and OR Techniques in Constraint Programming – CPAIOR*, pages 167–175, 2017.
8. Christophe Lecoutre, Lakhdar Sas, Sbastien Tabary, and Vincent Vidal. Reasoning from last conflict(s) in constraint programming. *Artificial Intelligence*, 173(18):1592 – 1614, 2009.
9. Alejandro López-Ortiz, Claude-Guy Quimper, John Tromp, and Peter van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence – IJCAI*, pages 245–250, 2003.

10. Claude-Guy Quimper, Peter van Beek, Alejandro López-Ortiz, Alexander Golynski, and Sayyed Bashir Sadjad. An efficient bounds consistency algorithm for the global cardinality constraint. In *Proceedings of the 9th International Conference Principles and Practice of Constraint Programming – CP*, pages 600–614, 2003.
11. Mohamed Siala, Christian Artigues, and Emmanuel Hebrard. Two clause learning approaches for disjunctive scheduling. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming – CP*, pages 393–402, 2015.
12. Mohamed Siala, Emmanuel Hebrard, and Marie-José Huguet. An optimal arc consistency algorithm for a particular case of sequence constraint. *Constraints*, 19(1):30–56, 2014.

NACRE

XCSP3 2018 Competition

Gaël Glorian

CRIL-CNRS, UMR 8188
Université d'Artois, F-62307 Lens France
`glorian@cril.fr`
`http://www.cril.univ-artois.fr/~glorian/`

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

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

The submitted version uses all the following technologies :

- arc consistency [3]
- variable selection heuristic: dom/wdeg [1]
- value selection heuristic: minimum
- restart policy: `luby` $\times$ 100 [2]
- a clause reasoning engine with lazy explanations, backjumping, clauses database reduction, etc.

Acknowledgements

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

References

[1] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence*, 2004.

[2] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 1993.

[3] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 1977.

Second XCSP3 Competition - Solvers description

Charles Thomas (charles.thomas@uclouvain.be)

August 29, 2018

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

1 Solvers submitted

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

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

1.1 Conflict ordering solver

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

1.2 Hybrid solver

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

1.3 Parallel solver

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

2 Command line usage

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

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

where:

- SOLVER should be replaced by the path to the solver jar file.

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

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

References

[1] Second international XCSP3 competition, `http://www.xcsp.org/competition`

[2] F. Boussemart, C. Lecoutre, and C. Piette, XCSP3: An integrated format for benchmarking combinatorial constrained problems, Tech. Report arXiv:1611.03398, CoRR, 2016, Available from `https://arxiv.org/abs/1611.03398` and `http://www.xcsp.org/format3.pdf`.

[3] beCool: Belgium Constraints Group, `http://becool.info.ucl.ac.be/`

[4] OscaR: Scala in OR, `https://bitbucket.org/oscarlib/oscar/wiki/Home`

[5] Steven Gay, Renaud Hartert, Christophe Lecoutre, Pierre Schaus: Conflict Ordering Search for Scheduling Problems, CP 2015.

[6] Charles Thomas, Pierre Schaus: Revisiting the Self-adaptive Large Neighborhood Search, CPAIOR 2018.

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

Neng-Fa Zhou[1] and Håkan Kjellerstrand[2]

[1] CUNY Brooklyn College & Graduate Center
[2] hakank.org

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

XCSP3

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

Picat

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

Parsing

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

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

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

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

Modeling

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

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

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

SAT Encoding

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

References

1. Frederic Boussemart, Christophe Lecoutre, and Gilles Audemard. XCSP3 - an integrated format for benchmarking combinatorial constrained problems. Technical report, xcsp.org.
2. Donald Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability.* Addison-Wesley, 2015.
3. Fernando C. N. Pereira and David H. D. Warren. Definite clause grammars for language analysis - A survey of the formalism and a comparison with augmented transition networks. *Artif. Intell.*, 13(3):231–278, 1980.
4. Neng-Fa Zhou and Håkan Kjellerstrand. Optimizing SAT encodings for arithmetic constraints. In *CP*, pages 671–686, 2017.
5. Neng-Fa Zhou and Håkan Kjellerstrand. Encoding pb constraints into sat via binary adders and bdds – revisited. In *Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion RCRA*, 2018.
6. Neng-Fa Zhou, Håkan Kjellerstrand, and Jonathan Fruhman. *Constraint Solving and Planning with Picat.* Springer, 2015.

Sat4j-CSP

XCSP3 Competition – 2018

Daniel Le Berre and Emmanuel Lonca

CRIL – Univ. Artois and CNRS
lastname@cril.fr

1 Solver Description

Sat4j[1] is a constraint satisfaction and optimization library developed in Java. It is a mature project (Sat4j was born in 2004) included since June 2008 in the Eclipse IDE as part of its plugin dependency management engine. As indicated by its name, the original library design was built on top of a SAT solver. Taking advantage of the this initial solver, several new kinds of solvers were added to the library (pseudo-Boolean solver, MAXSAT solver, ...) including a CSP solver.

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

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

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

[1] http://xcsp.org/competition

2 Future Work

Sat4j-CSP3 was developed with the aim to handle the whole set of constraints allowed in XCSP3-core. However, at this step, there is room for improvements. First, the encoding of integer variables uses one binary variable for each value in its range. For arrays of integer variables taking their values in large domains, this results in the declaration of a huge amount of Boolean variables, making the solver running out of memory. A slight adaptation of some encodings used for At-Most-1 constraints (like the binary encoding or the Two-Product encoding[3]) may help in terms of efficiency to keep the memory consumption under the system limits.

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

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

3 Acknowledgments

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

References

1. Berre, D.L., Parrain, A.: The sat4j library, release 2.2. JSAT **7**(2-3), 59–6 (2010), https://satassociation.org/jsat/index.php/jsat/article/view/82
2. Boussemart, F., Lecoutre, C., Piette, C.: XCSP3: an integrated format for benchmarking combinatorial constrained problems. CoRR **abs/1611.03398** (2016), http://arxiv.org/abs/1611.03398
3. Chen, J.: A new sat encoding of the at-most-one constraint. Proc. Constraint Modelling and Reformulation (2010)
4. Le Berre, D., Lynce, I.: Csp2sat4j: a simple csp to sat translator. Proceedings of the 2nd International CSP Solver Competition pp. 43–54 (2008)

sCOP: SAT-based Constraint Programming System

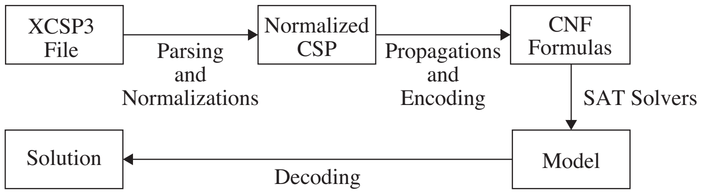
XCSP3 Competition in 2018

Takehide Soh[1], Daniel Le Berre[2], Mutsunori Banbara[1], and Naoyuki Tamura[1]

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

1 Overview

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



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

2 Parsing and Normalizations

Parsing is done by using an official tool XCSP3-Java-Tools [3]. Currently, sCOP accepts constraints in the XCSP3-core language[4].

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

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

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

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

[3] https://github.com/xcsp3team/XCSP3-Java-Tools
[4] http://www.xcsp.org/specifications

3 Propagations and Encoding

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

Encoding methods used in sCOP are the followings:

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

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

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

4 SAT Solvers

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

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

References

1. Eén, N., Sörensson, N.: Translating pseudo-Boolean constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation 2(1-4), 1–26 (2006)
2. Soh, T., Banbara, M., Tamura, N.: Proposal and evaluation of hybrid encoding of CSP to SAT integrating order and log encodings. International Journal on Artificial Intelligence Tools 26(1), 1–29 (2017)
3. Tamura, N., Banbara, M.: Sugar: a CSP to SAT translator based on order encoding. In: Proceedings of the 2nd International CSP Solver Competition. pp. 65–69 (2008)
4. Tamura, N., Banbara, M., Soh, T.: PBSugar: Compiling pseudo-boolean constraints to SAT with order encoding. In: Proceedings of the 25th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2013), IEEE. pp. 1020–1027 (Nov 2013)
5. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. Constraints 14(2), 254–272 (2009)

[5] https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/
[6] http://www.labri.fr/perso/lsimon/glucose/

Chapter 4

Results

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

4.1 Rankings

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

Problem	Goal	Exploration	Timeout
CSP	one solution	sequential	40 minutes
CSP	one solution	parallel	40 minutes
COP	best solution	sequential	4 minutes
COP	best solution	sequential	40 minutes
COP	best solution	parallel	40 minutes

Table 4.1: Standard Tracks.

Problem	Goal	Exploration	Timeout
CSP	one solution	sequential	40 minutes
COP	best solution	sequential	40 minutes

Table 4.2: Mini-Solver Tracks.

Also, note that:

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

- Parallel solvers were run on two processors (8 cores) and were allocated $31,000$ MiB of memory.
- The selection of instances for the Standard tracks was composed of 236 CSP and 346 COP instances.
- The selection of instances for the Mini-solver tracks was composed of 176 CSP and 188 COP instances.

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

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

4.2 A few Comments

For simplicity, the solvers that encode constraints in SAT (PicatSAT, Sat4j and sCOP) will be called *SAT-based* solvers. The solvers that use more traditional CP techniques (Choco, Concrete, CoSoCo, Mistral, and OscaR) will be called *CP-based* solvers. Nacre is hybrid as it involves a strong clause reasoning engine. Finally, the solvers that exploit tree decomposition techniques (BTD, BTD_12, and macht) will be called *decomposition-based* solvers.

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

4.2.1 Standard Track – CSP – Sequential

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

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

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

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

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

		#solved		%inst.	%VBS
	Virtual Best Solver (VBS)	164	104 SAT, 60 UNSAT	69%	100%
1	scop *order+MapleCOMSPS*	146	92 SAT, 54 UNSAT	62%	89%
2	scop *both+MapleCOMSPS*	140	87 SAT, 53 UNSAT	59%	85%
3	PicatSAT *2018-08-14*	138	85 SAT, 53 UNSAT	58%	84%
4	Mistral-2.0	116	80 SAT, 36 UNSAT	49%	71%
5	Choco-solver *4.0.7b seq*	115	77 SAT, 38 UNSAT	49%	70%
6	Concrete *3.9.2*	92	64 SAT, 28 UNSAT	39%	56%
7	OscaR-Conf. Ordering+restarts	90	62 SAT, 28 UNSAT	38%	55%
8	Concrete *3.9.2-SuperNG*	84	55 SAT, 29 UNSAT	36%	51%
9	Sat4j-CSP	83	40 SAT, 43 UNSAT	35%	51%
10	OscaR - Conflict Ordering	81	51 SAT, 30 UNSAT	34%	49%
11	cosoco *1.12*	79	53 SAT, 26 UNSAT	33%	48%
12	BTD_12	76	32 SAT, 44 UNSAT	32%	46%
13	BTD	76	31 SAT, 45 UNSAT	32%	46%
14	macht	66	33 SAT, 33 UNSAT	28%	40%

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

		#solved		%inst.	%VBS
	Virtual Best Solver (VBS)	146	146 OPT	42%	100%
1	PicatSAT *2018-08-14*	132 (132)	132 OPT	38%	90%
2	Concrete *3.9.2*	105 (148)	105 OPT	30%	72%
3	Choco-solver *4.0.7b seq*	102 (154)	102 OPT	29%	70%
4	OscaR-Conf. Ordering+restarts	99 (132)	99 OPT	29%	68%
5	Concrete *3.9.2-SuperNG*	99 (139)	99 OPT	29%	68%
6	cosoco *1.12*	64 (112)	64 OPT	18%	44%
7	OscaR - Hybrid *2018-08-14*	61 (132)	61 OPT	18%	42%
8	Sat4j-CSP	54 (86)	54 OPT	16%	37%

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

		#best	%inst.	%VBS
	Virtual Best Solver (VBS)	316	91%	100%
1	Concrete *3.9.2*	151	44%	48%
2	Choco-solver *4.0.7b seq*	146	42%	46%
3	OscaR - Hybrid	139	40%	44%
4	OscaR - Conflict Ordering with restarts	133	38%	42%
5	Concrete *3.9.2-SuperNG*	129	37%	41%
6	Mistral-2.0	123	36%	39%
7	cosoco	107	31%	34%
8	Sat4j-CSP	78	23%	25%

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

		#solved		%inst.	%VBS
Virtual Best Solver (VBS)		168	104 SAT, 64 UNSAT	71%	100%
1	scop *order+glucose-syrup*	151	95 SAT, 56 UNSAT	64%	90%
2	scop *both+glucose-syrup*	138	82 SAT, 56 UNSAT	58%	82%
3	Choco-solver *4.0.7b par*	134	88 SAT, 46 UNSAT	57%	80%
4	OscaR - Parallel with EPS	89	56 SAT, 33 UNSAT	38%	53%

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

		#solved		%inst.	%VBS
Virtual Best Solver (VBS)		113	53 SAT, 60 UNSAT	64%	100%
1	NACRE	86	43 SAT, 43 UNSAT	49%	76%
2	miniBTD_12	79	36 SAT, 43 UNSAT	45%	70%
3	miniBTD	75	32 SAT, 43 UNSAT	43%	66%
4	cosoco	72	42 SAT, 30 UNSAT	41%	64%
5	minimacht	69	37 SAT, 32 UNSAT	39%	61%
6	GG's minicp	56	37 SAT, 19 UNSAT	32%	50%
7	Solver of Schul & Smal	54	23 SAT, 31 UNSAT	31%	48%
8	MiniCPFever	54	34 SAT, 20 UNSAT	31%	48%
9	slowpoke	38	38 SAT	22%	34%
10	SuperSolver	31	31 SAT	18%	27%
11	The dodo solver	25	25 UNSAT	14%	22%

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

		#solved		%inst.	%VBS
Virtual Best Solver (VBS)		48	48 OPT	26%	100%
1	cosoco	46 (122)	46 OPT	24%	96%
2	Solver of Schul & Smal	35 (44)	35 OPT	19%	73%
3	GG's minicp	3 (22)	3 OPT	2%	6%
4	MiniCPFever	0 (50)		0%	0%
5	SuperSolver	0 (23)		0%	0%
6	The dodo solver	0 (18)		0%	0%
7	slowpoke	0 (12)		0%	0%

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.2.2 Standard Track – COP – Sequential

Note that Mistral is sometimes mentioned during analysis; although it has been disqualified for this track.

Auction OscaR is clearly the best competitor for this problem, always giving the best bounds. Choco is the best second solver. There is no clear advantage of using the model variant 'cnt' (with `atMost1`) or the model variant 'sum'.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Bibliography

[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] K. Cheng and R. 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] J. Demeulenaere, R. Hartert, C. Lecoutre, G. Perez, L. Perron, J.-C. Régin, and P. Schaus. Compact-table: efficiently filtering table constraints with reversible sparse bit-sets. In *Proceedings of CP'16*, pages 207–223, 2016.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] C. Lecoutre. MCSP3: Easy modeling for everybody (version 1.1). Technical report, December, 2018. Available from https://github.com/xcsp3team/XCSP3-Java-Tools/blob/master/doc/modeler1-1.pdf.

[15] J.-B. Mairy, Y. Deville, and C. Lecoutre. The smart table constraint. In *Proceedings of CPAIOR'15*, pages 271–287, 2015.

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

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

[18] G. Perez and J.-C. Régin. Improving GAC-4 for Table and MDD constraints. In *Proceedings of CP'14*, pages 606–621, 2014.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.