Teaching Constraints through Logic Puzzles

Péter Szeredi[1,2]

[1] Dept. of Computer Science and Information Theory,
Budapest University of Technology and Economics
H-1117 Budapest, Magyar tudósok körútja 2
szeredi@cs.bme.hu
[2] IQSYS
Information Systems Ltd.
H-1135 Budapest, Csata u. 8

Abstract. The paper describes the experiences of teaching a constraint logic programming course at the Budapest University of Technology and Economics. We describe the structure of the course, the material covered, some examples, assignments, and examination tasks. Throughout the paper we show how logic puzzles can be used to illustrate constraint programming techniques.

1 Introduction

Logic puzzles are getting more and more popular. For example, in the last few years, four monthly periodicals, publishing logic puzzles only, have been established in Hungary. There are numerous web-sites offering this type of amusement. Best puzzle solvers gathered in October 2003 in the Netherlands for the 12th time to compete in the World Puzzle Championship. Puzzles from earlier Championships were published in e.g. [1, 2].

Personally, I found logic puzzles quite intriguing, and started to use them in my Prolog lectures at the Budapest University of Technology and Economics (BUTE), and later also in the lectures on constraint logic programming (CLP). This is not new, as puzzles have been used for demonstrating the capabilities of CLP from the very establishment of this discipline. For example, Pascal van Hentenryck devotes a complete section to solving puzzles in his seminal book [3].

This paper gives an account of the constraint course at BUTE, with a brief summary of the preceding Prolog course. While describing various aspects of the course, I highlight those parts where logic puzzles are used.

The paper is structured as follows. First, in Sect. 2, the prerequisite Prolog course is briefly introduced. The next two sections describe the structure of the constraint course. Section 5 presents the details of one of the case studies, in which two solvers for the Domino puzzle are developed. Section 6 describes the major assignments of the past four years, while Sect. 7 presents some examination problems. In the last two sections we discuss the lessons learned and conclude the paper. Finally, Appendix A contains the complete code of a solver for "knights, knaves, and normals" type puzzles, while Appendix B gives a detailed performance evaluation of the Domino puzzle solvers.

K.R. Apt et al. (Eds.): CSCLP 2003, LNAI 3010, pp. 196–222, 2004.
© Springer-Verlag Berlin Heidelberg 2004

2 Background

The paper presents an elective constraint logic programming course for undergraduate students of informatics at BUTE[1]. It presupposes the knowledge of logic programming, which is taught in a compulsory "Declarative Programming" (DP) course in the second year. The DP course also covers functional programming, which is represented by Moscow SML, and is taught by Péter Hanák. The logic programming part, presented by Péter Szeredi, uses SICStus Prolog.

The DP course, in part due to its late introduction into the curriculum, is a lectures only course, with no laboratory exercises. It is very difficult to teach programming languages without the students doing some programming exercises. In an attempt to solve this problem, we have developed a computer tool, called ETS (Electronic Teaching aSsistant, or Elektronikus TanárSegéd in Hungarian) [4], which supports Web-based student exercising, assignment submission/evaluation, marking, etc.

During the DP course we issue 4–6 so called minor programming assignments and a single major one. The major assignment normally involves writing a solver for a logic puzzle, a task very much suited for constraint programming. Examples of such puzzles are given in Sects. 5, 6, and 7. The marks for the major assignment are based on the number of test cases solved by the student's program, as well as on the quality of the documentation. There is a so called "ladder competition" for students who have solved all the normal test cases. In this competition additional points are awarded to students who can solve the largest puzzles.

This creates some interest in constraint programming, especially when the students are told that their puzzle-solving programs can be made much faster, often by two orders of magnitude, by using CLP. Because of this, a relatively large number of students enrol in the elective constraints course immediately, in the semester following their DP course. For example, 55 students registered for the constraints course in the autumn term of 2002, of which 38 took DP in the previous semester[2].

3 The Structure of the Constraint Course

The constraint course has been presented since 1997, eight times up till 2003 (in the spring term till 2000, and in the autumn term since 2000).

There is a 90 minute lecture each week during the term, which usually lasts 14 weeks. There are no laboratory exercises for this course. To ensure that students do some actual programming, there is (and has always been) a compulsory major assignment, which can be submitted up till the end of the exam session following the term. To motivate and help the students in doing programming exercises during the term, minor assignments were introduced in the autumn term of

[1] Occasionally the course is taken by students of electrical engineering, and by postgraduates.

[2] Altogether 324 students completed the DP course in the spring 2002 semester, of which 62 got the highest grade (5), of which 27 enrolled in the constraints course.

Table 1. The layout of the course

	Topic	Time	Slides
1.	Prolog extensions relevant for CLP *Minor Assignment 1:* CLP(MiniB)	3 units	15 slides
2.	The CLP($\mathcal{X}$) scheme and CLP(R/Q)	3 units	19 slides
3.	CLP(B)	2 units	8 slides
4.	CLP(FD) *Minor Assignments 2-4, Major Assignment*	14 units	109 slides
5.	Constraint Handling Rules	2 units	11 slides
6.	Mercury	4 units	23 slides
	$\sum$	28 units	185 slides

2002. There were four of these, with a due date of approximately two weeks after issue.

The layout of the course is shown in Table 1, with approximate time schedule, and the number of (fairly dense) handout slides for each topic. The *unit* of the time schedule is 45 minutes, i.e. there are 2 units per lecture.

In addition to its main topic of constraints, the course also contains a brief overview of the Mercury language[3]. In this paper we limit our attention to the constraint part of the course, which uses SICStus Prolog [5], as the underlying implementation. We will now discuss in turn the topics 1.-5. listed in Table 1,

The first topic deals with Prolog extensions relevant for constraint programming. These include the portray hook, term- and goal expansion hooks, and, most importantly, the coroutining facilities. These features are introduced by gradually developing a simple "constraint solver" on the domain of natural numbers. This system, called CLP(MiniNat), is based on the Peano arithmetic, and allows function symbols `+`, `-`, and `*`, as well as the usual six relational symbols. The user interface is similar to CLP(R):

```
| ?- {X*X+Y*Y=25, X > Y}.
X = 4, Y = 3 ? ;
X = 5, Y = 0 ? ; no
```

Using goal expansion, such constraints are transformed into a sequence of calls to predicates `plus/3` and `times/3`. Both predicates block until at least one of the arguments is ground, and then execute, possibly creating choice-points.

[3] The actual title of the course is "High Efficiency Logic Programming". The title reflects the original idea, which was to split the course into two, roughly equal parts: one about efficient compilation of logic programs, exemplified by the Mercury system, and the other about applying constraint techniques for solving search problems efficiently, using constraint techniques. However, from the very start, the focus shifted to the constraint topic.

Assignment 1 is issued at the end of the first part. Here the students are asked to write a coroutining-based simple quasi-CLP system on Booleans, called CLP(MiniB).

The second part starts with a brief overview of the CLP($\mathcal{X}$) scheme, introducing the notions of constraint store, primitive and non-primitive constraints. These are then exemplified by discussing the CLP(Q/R) extension, culminating in Colmerauer's Perfect Rectangle Problem [6]: tiling a rectangle with different squares. Next, a summary of the declarative and procedural semantics of the CLP scheme is given. This setup, where the theoretical issues are discussed after a concrete example of the CLP($\mathcal{X}$) scheme is shown, seems to be more easily digested by the students.

The next part presents the second instantiation of the CLP scheme: CLP(B). This is illustrated by simple examples of circuit verification, and a program playing (the user part of) the well known minesweeper game.

The fourth part, dealing with CLP(FD), is naturally the largest and is discussed in the next section. There are three minor assignments related to this part, and the major assignment is also issued here.

The constraint part of the course is concluded with a brief overview of CHR (Constraint Handling Rules). Here the biggest example is a CHR program for solving the Areas puzzle (see Fig. 1).

The Areas puzzle: A rectangular board is given with some squares specified as positive integers. Fill in all squares of the board with positive integers so that any maximal contiguous set of squares containing the same integer has the area equal to this integer (two squares are contiguous if they share a side). An example puzzle and its unique solution (with thick lines marking the area boundaries) is given below:

2	1		4	3
			2	
	5		3	
3				2

2	4	4	3	3
2	1	4	4	3
3	5	5	2	2
3	5	3	3	3
3	5	5	2	2

Fig. 1. The Areas puzzle, a CHR example

4 The CLP(FD) Part of the Course

The CLP(FD) topic takes up roughly half of the semester. Its subdivision is shown in Table 2. We start with a brief overview of the CSP approach. Next, the basics of the CLP(FD) library are introduced: the arithmetic and membership constraints, their execution process, the notion of interval and domain

Table 2. Subdivision of the CLP(FD) part of the course

	CLP(FD) subtopic	Time	Slides
4.1	CSP overview, CLP(FD) basics	3 units	23 slides
4.2	Reification, propositional constraints, labeling *Assignment 2:* Cross sums puzzle	3 units	25 slides
4.3	Combinatorial constraints *Major Assignment:* Magic Spiral puzzle	1 unit	12 slides
4.4	User defined constraints *Assignment 3:* Write a specific indexical *Assignment 4:* Write a specific global constraint	2 units	20 slides
4.5	The FDBG debugging tool	1 unit	10 slides
4.6	Case studies	4 units	19 slides

consistency, the importance of redundant constraints. This is exemplified by the classical CSP/CLP(FD) problems: map colouring, n-queens, the zebra puzzle, and the magic series.

The last example (magic series) leads to the topic of reified and propositional constraints, interval and domain entailment. An elegant illustration of the propositional constraints is a program for solving "knights, knaves and normals" puzzles of Smullyan [7], shown in Appendix A.

Although simple uses of the labeling predicates were already introduced, the full discussion of labeling comes at this point. This includes user-definable variable selection and value enumeration functions, as well as a comparison of the performance of various labeling schemes on the n-queens problem.

Having covered all the stages of constraint solving, the next minor assignment is issued. The students are required to write a program for solving a numeric crossword puzzle. Figure 2 describes the assignment and gives a sample run.

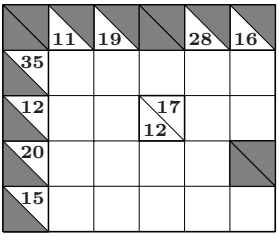
The next subtopic is an overview of combinatorial constraints provided by SICStus Prolog. This is rather dry, not much more than the manual pages, lightened up by a few small additional examples.

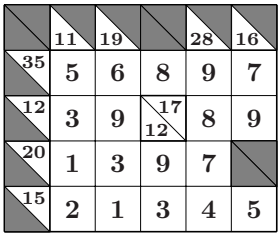
At this point the major assignment is issued. In the last three years, this was exactly the same assignment as the one issued for the Declarative Programming course in the preceding semester (see Sect. 6 for examples).

The fourth subtopic of the CLP(FD) part is about user-definable constraints. Both global constraints and FD-predicates (indexicals) are covered, including the reifiable indexicals. The last two minor assignments are about writing such constraints, as shown in Fig. 3.

When students submit an assignment, the ETS teaching support tool runs it on several predefined test-cases, and sends the results back to the student. I had to repeatedly extend the set of test cases for the FD-predicate assignment, which seems to be the most difficult one, as it was not exhaustive, and thus

The Cross Sums puzzle: Fill in a "numeric" crossword puzzle, with integers from the $[1, Max]$ interval. All integers within a "word" have to be different, and the sum of these is given as the "definition" of each word. An example and its solution ($Max = 9$):





Assignment 2: Write a CLP(FD) program for solving the Cross Sums puzzle. Sample run:

```
| ?- T = [x\x, 11\x,19\x, x\x,28\x,16\x],
         [x\35,   _,   _,   _,   _,   _],
         [x\12,   _,   _,12\17,  _,   _],
         [x\20,   _,   _,   _,   _, x\x],
         [x\15,   _,   _,   _,   _,   _]],
     cross_sums(T, 9).

     T = [x\x, 11\x,19\x, x\x,28\x,16\x],
         [x\35,   5,   6,   8,   9,   7],
         [x\12,   3,   9,12\17,  8,   9],
         [x\20,   1,   3,   9,   7, x\x],
         [x\15,   2,   1,   3,   4,   5]] ? ;
no
| ?-
```

Fig. 2. The Cross Sums puzzle

Assignment 3: Write an FD-predicate `'z>max(x,y)'(X, Y, Z)` which implements a domain-consistent constraint, with the meaning equivalent to `Z #> max(X,Y)`. Write all four clauses of the FD-predicate.

Assignment 4: Write a global constraint `max_lt(L, Z)`, where `L` is list of FD variables, and `Z` is an FD variable. The meaning of the constraint: the maximum of `L` is less than `Z`, Make the global constraint efficient, avoid re-scanning irrelevant variables, by using a state. For example, the following goal should run in linear time with respect to `N`:

```
| ?- N = 500, length(L, N), domain(L, -5, 0), X in 0..N,
     max_lt([X|L], Z), X#>0, X#>1, ..., X#>N-1.
```

Fig. 3. Minor assignments for writing user-defined constraints

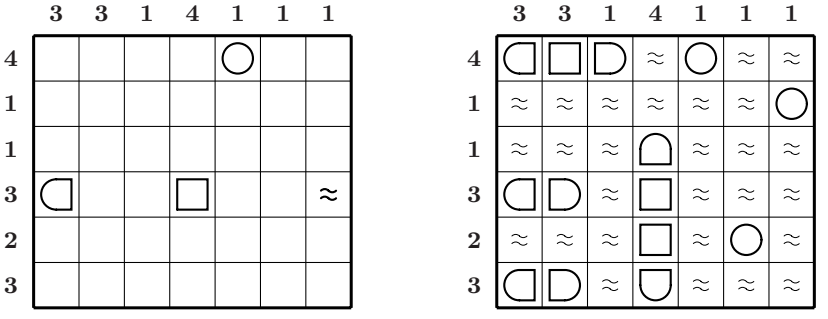


Fig. 4. A Battleship puzzle with its solution

incorrect student solutions got accepted. Finally, I developed, and handed out to students, a simple tool for exhaustively checking the semantics of an FD-predicate against a specification in Prolog. The specification is a Prolog goal for *testing* the constraint with ground arguments. The semantic check can be run automatically, if some directives are inserted in front of the FD-predicate. In the case of Assignment 3 (Fig. 3) these may look like this:

```
:- fd_pred_semantics('z>max(x,y)'(X,Y,Z), Z>max(X,Y)).
:- fd_test_range(1, 2).      % Try all combinations in the range 1..2
```

The next subtopic within CLP(FD) is debugging, as the students are expected now to start developing their major assignments. A brief overview of the SICStus FDBG finite domain debugging library [8] is presented here. An interesting point is that this library has been developed by two former students of this course.

The CLP(FD) part is concluded by three case studies. First the Perfect Square Problem from [9] is presented. This gives an opportunity to discuss disjunctive constraints and special labeling techniques. Performance results are presented for several solution variants, including those using the SICStus Prolog library predicates `cumulative` and `disjoint2`.

The next two case studies are major assignments from earlier years: the Battleship and the Domino puzzles. The latter is discussed in the next section.

The Battleship puzzle involves placing ships of different lengths on a rectangular board (all ships are one square wide). The ships can be placed horizontally or vertically, and they can be of different colours. For each length and colour we are given the number of battleships to be placed. We are also given, for each colour, the number of battleship pieces in each row and column. The battleships cannot touch each other, not even diagonally. Furthermore, for certain squares of the board it is specified that they contain a certain kind of battleship piece, or sea. Ships of length one have a circle shape, while longer ships have rounded off bow and stern pieces, with square mid-pieces in between.

Figure 4 shows an example of a Battleship puzzle, with a single colour, together with its solution. The numbers shown above the columns and in front of

the rows are the counts of battleship pieces. On the board, three ship pieces are given, and one field is marked as sea. In addition to the board on the left hand side, the puzzle solver is given the ship counts: length 4: 1 ship, length 3: 1 ship, length 2: 2 ships, length 1: 3 ships.

The Battleships case study (program, test-data, results) can be downloaded from [10].

5 A Case Study: The Domino Puzzle

The Domino puzzle was issued as the major assignment in spring 2000. Since 2001, it has been used as a case study, showing the steps and techniques needed to solve a relatively complex problem using CLP.

5.1 The Puzzle

A rectangular board is tiled with the full set of dominoes with up to d dots on each half domino. The set of all such dominoes can be described as:

$$D_d = \{\langle i, j\rangle | 0 \leq i \leq j \leq d\}$$

For each square of the board we are told the number of dots on the half domino there, but we do not know the domino boundaries. The task is to find out the tiling, i.e. to reconstruct the domino boundaries.

Figure 5 shows a sample board, its solution, and the Prolog format of these.

The problem:

1	3	0	1	2
3	2	0	1	3
3	3	0	0	1
2	2	1	2	0

The (unique) solution:

1	3	0	1	2
3	2	0	1	3
3	3	0	0	1
2	2	1	2	0

The Prolog description:

```
[[1, 3, 0, 1, 2],
 [3, 2, 0, 1, 3],
 [3, 3, 0, 0, 1],
 [2, 2, 1, 2, 0]]
```

The solution in Prolog:

```
[[n, w, e, n, n],
 [s, w, e, s, s],
 [w, e, w, e, n],
 [w, e, w, e, s]]
```

Fig. 5. A sample Domino problem for $d = 3$

The task is to write a `domino/2` predicate, which takes, as its first argument, a board represented by a matrix of values from $[0, d]$, and returns, in the second

argument, a matrix of the same size. The elements of the output matrix are compass point abbreviations `n`, `w`, `s`, or `e`, indicating that the square in question is respectively a northern, western, southern, or eastern half of a domino.

In the case study two alternative solutions are presented: one based on the *compass* model, and another one using the *border* model. An independent Eclipse solution to this puzzle, found in [11], uses the latter model.

5.2 The Compass Model

This model follows naturally from the expected output format: each square of the board is assigned a *compass* variable, which specifies the compass point of the half domino it is covered with. The compass variables are named $CV_{yx}, 1 \leq y \leq max_y, 1 \leq x \leq max_x$ (where max_y and max_x are the number of rows and columns of the board), their domain is an arbitrary numeric encoding, say n, w, s, e, of the four compass points.

With this representation, it is easy to ensure that the tiling is consistent: if a square is "northern", its neighbour to the south has to be "southern", etc. However it is difficult to guarantee that each domino of the set is used only once.

Therefore we introduce another, redundant set of variables. For each domino $\langle i, j \rangle \in D_d$, we set up a *domino* variable DV_{ij} specifying the position of this domino on the board. This automatically ensures that each domino is used exactly once.

The domino positions can be described e.g. by suitably encoding the triple $\langle row, column, dir \rangle$, where the coordinates are those of the northern/western half of the domino, and *dir* specifies its direction (vertical or horizontal). If a domino $\langle i, j \rangle$ can be placed on k positions, then the domain of D_{ij} will be $[1, k]$. The mapping from this domain to the triples is needed only while posting the constraints, and so it can be kept in a separate Prolog data structure.

For example, the $\langle 0, 2 \rangle$ domino of Fig. 5 can be placed on the following board positions: $\langle 2, 2, horizontal \rangle$, $\langle 3, 4, vertical \rangle$ and $\langle 4, 4, horizontal \rangle$. Therefore the domain of D_{02} will be $\{1, 2, 3\}$, describing these three placings.

The constraints of the compass model are the following:

Neighbourship constraints: For each pair of neighbouring squares on the board we state that their compass variables have to be consistent, e.g. $CV_{14} = n \Leftrightarrow CV_{24} = s$, $CV_{14} = w \Leftrightarrow CV_{15} = e$, etc.

Placement constraints: For each domino variable, and for each of its possible values, we state that the given horizontal (vertical) placement holds iff the compass variable of the square specified by this placement is a western (northern) half of a domino. For example, the $\langle 0, 2 \rangle$ domino of Fig. 5 gives rise to the following constraints: $DV_{02} = 1 \Leftrightarrow CV_{22} = w$, $DV_{02} = 2 \Leftrightarrow CV_{34} = n$, $DV_{02} = 3 \Leftrightarrow CV_{44} = w$.

Note that both constraint types are of form $X = c \Leftrightarrow Y = d$, where c and d are constants. Three implementations for this constraint are presented. The first variant is the trivial formulation using reification and propositional constraints.

The second uses an FD-predicate implementing the constraint $X = c \Rightarrow Y = d$, and calls it twice, to ensure equivalence. Third, the whole equivalence is coded as a single FD-predicate. The two FD-predicates are shown in Fig. 6. All solutions have the same pruning behaviour. Our measurements (see Table 4 in Appendix B) show that the second is the fastest variant. This involves four indexicals, as opposed to the the third, single FD-predicate implementation, which has only two. However, in the latter case the indexicals are more complex, and wake up unnecessarily often, hence the inferior performance of this solution.

```
'x=c=>y=d'(X, C, Y, D) +:
        X in (dom(Y) /\ {D}) ? (inf..sup) \/ \({C}),
        Y in ({X} /\  \({C})) ? (inf..sup) \/ {D}.

'x=c<=>y=d'(X, C, Y, D) +:
        X in ((dom(Y) /\ {D}) ? (inf..sup) \/ \({C})) /\
             ((dom(Y) /\ \({D})) ? (inf..sup) \/ {C}),
        Y in ((dom(X) /\ {C}) ? (inf..sup) \/ \({D})) /\
             ((dom(X) /\ \({C})) ? (inf..sup) \/ {D}).
```

Fig. 6. FD-predicates for implementing equivalence constraints

5.3 The Border Model

In this model we assign a variable to borders (line segments) between the squares of the board. Such a *border* variable is 1, if the given segment is a centerline of a domino, otherwise it is 0. Let us denote by E_{yx} (S_{yx}) the variables corresponding to the eastern (southern) borders of the square (y, x) on the board ($1 \leq y \leq max_y, 1 \leq x \leq max_x$) [4].

Analogously to the compass model, we have two types of constraints:

Neighbourship constraints: For each square on the board we state that *exactly one* of the four line segments bordering it will be a domino centerline, i.e. the sum of the corresponding variables is 1. For example, for the square $(2, 4)$ the constraint is: $S_{14} + E_{23} + S_{24} + E_{24} = 1$.

Placement constraints: For each domino, consider all its possible placements. We state that exactly one of these placements has to be selected, i.e. from amongst the line segments in the centre of these placements, exactly one will be a domino centerline. For example, the $\langle 0, 2 \rangle$ domino of Fig. 5 gives rise to the following constraint: $E_{22} + S_{34} + E_{44} = 1$.

Note that again both constraint types are of the same form: the sum of some 0-1 variables is 1. Two implementations are evaluated for the $\sum X_i = 1$ constraint: one using the `sum/3` global constraint of SICStus Prolog, the other using

[4] Note that in this set of variables there are some, which correspond to line segments on the outer border of the board (e.g. $S_{max_y x}$), these are assigned a constant 0 value. To cover the northern and the western border of the board, we use similarly constant 0 valued variables S_{0x} and E_{y0}.

indexicals. For the latter we make use of the fact that the SICStus `clpfd` library is capable of compiling linear arithmetic constraints to indexicals. For example, an FD-predicate implementing a three-way sum constraint can be specified as `sum3(A,B,C) +: A+B+C #= 1.` Measurements show that the FD-predicate is faster up to 5 summands, and slower above that.

5.4 Search

In the compass model we have a choice of which variable-set to label: the compass variables, or the domino variables. Furthermore, in both models, we can explore different variable selection schemes. However, for large test cases to be solved in acceptable time, we have to apply an additional technique, *shaving*.

Shaving [12] is a straightforward, indirect proof scheme, which seems to work well for logic puzzles. It involves tentatively posting a constraint: if this leads to failure, then the negation of the constraint is known to hold. Otherwise the constraint is withdrawn, using the "negation by failure" scheme. The following piece of Prolog code tries out if setting `X` to `Value` leads to failure, and excludes the given value from the domain of `X` if this happens.

```
shave(X, Value) :-
        (   \+ X #= Value -> X #\= Value
        ;   true
        ).
```

Such shaving operations are normally performed during labeling. For example, in the Battleship puzzle case study, shaving was used to exclude a specific value (the one corresponding to "sea") from the domain of the FD variables describing the squares of the board. This process was performed relatively rarely, before labeling the placement of ships of each colour. Experiments showed that it is worthwhile to do shaving, but it does not pay off to do repetitive shaving (i.e. repeat the shaving, until no more domains are pruned).

A slightly more general shaving scheme is presented in the Domino case study. We still try to prove inconsistency by setting an FD variable to a concrete value. However, here one can specify multiple values to be tried in shaving, one after the other. The frequency of doing (non-repetitive) shaving can also be prescribed: a shaving scan can be requested before labeling every kth variable.

In the compass model, a shaving step involves scanning all the compass variables. Out of the two opposite compass values (e.g. northern and southern) it is enough to try at most one, because the neighbourship constraints will force the shaving of the other value. Also, shaving all compass variables with two non-opposite values will result in all domino variables being substituted with all their possible values, because of the placement constraints. Consequently, there is no need to shave the domino variables. In the compass model we thus shave the compass variables only, trying out the $[n, w]$ and $[n]$ value sets.

In the border model we explore shaving variants using value sets [0], [1], and $[0, 1]$.

The case study is concluded with a relatively detailed performance evaluation, the essence of which is shown in Appendix B.

6 Major Assignments

In 1997 and 1998 the major assignments were the same: writing a Nonogram solver[5]. The major assignments of the 1999 and 2000 spring semesters, the Battleship and Domino puzzles, later became part of the course as case studies. These were discussed in the previous sections. In this section we describe the problems set as major assignments in (the autumn semesters of) years 2000–2003. As opposed to the case studies, here we do not publish any details of the solution, so that the problems can be re-used in other constraint courses.

6.1 The Countries Puzzle

The Countries puzzle was set as the major assignment of the constraint course in the autumn semester of 2000.

The Puzzle. A rectangular board of size $n * m$ is given. Certain fields in the board contain numbers, these represent the capitals of different countries. Each country can only occupy contiguous fields in the row and/or in the column of the capital. This means that each country consists of the capital, plus $I_n \geq 0$ adjacent fields northwards, $I_w \geq 0$ adjacent fields westwards, $I_s \geq 0$ adjacent fields southwards, and $I_e \geq 0$ adjacent fields eastwards. The number $I \geq 0$ given on the board is the number of fields occupied by the given country, in addition to the capital, i.e. $I = I_n + I_w + I_s + I_e$.

The task is to determine, how much each country extends in each of the four main compass directions[6].

Figure 7 shows an instance of the puzzle for $n = 4, m = 5$, together with its (single) solution, showing the country boundaries.

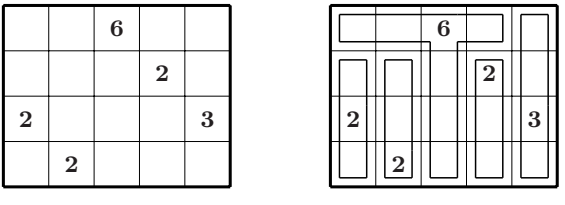


Fig. 7. A Countries puzzle and its solution

[5] Nonongrams are known by many other names, see e.g. [13]. A similar student project, for developing a nonogram solver using constraint techniques, is described on page [14].

[6] In Hungarian, this puzzle is called the compass point *(szélrózsa)* puzzle.

The Prolog Representation. A Countries puzzle is described by a Prolog structure of form `c(`*n*`,`*m*`,`*Capitals*`)`, where *Capitals* is a list of capitals, each represented by a structure `c(`*Row*`,`*Column*`,`*CountrySize*`)`. A solution is a list which, for each capital, as listed in *Capitals*, gives the extent of its country in the four compass directions. Each element of the solution list is a structure of form `e(`*North*`,`*West*`,`*South*`,`*East*`)`.

Below is a sample run of the Countries program, for the test case of Fig. 7.

```
| ?- countries(c(4, 5,
                 [c(1,3,6),c(2,4,2),c(3,1,2),c(3,5,3),c(4,2,2)]), S).
S = [e(0,2,3,1),e(0,0,2,0),e(1,0,1,0),e(2,0,1,0),e(2,0,0,0)] ? ;
no
```

6.2 The Tents Puzzle

The Tents puzzle was the major assignment for the declarative programming course in spring 2001, and for the constraint course in autumn 2001.

The Puzzle. A rectangular board of size $n * m$ is given. Certain fields in the board contain *trees*. The task is to tie a single *tent* to each tree. The tent should be placed on a field next to its tree, either to the south, west, north, or east, but not diagonally. No tents touch each other, not even diagonally. As an additional constraint, the number of tents is specified for some rows and columns.

Figure 8 shows an instance of the puzzle for $n = m = 5$, together with its single solution, where the trees are denoted by the Υ symbol, and the tents by Δ. The numbers in front of the board and above it are the tent counts for the rows and the columns.

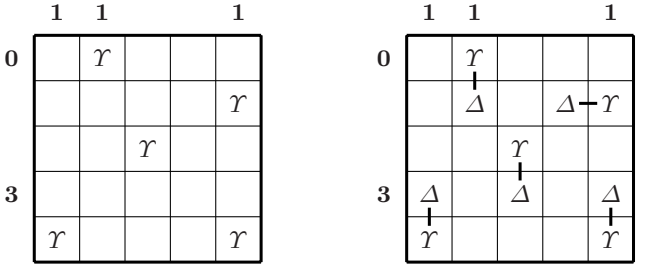


Fig. 8. A Tents puzzle and its solution

The Prolog Representation. The Prolog structure `t(`*RCs*`,`*CCs*`,`*Ts*`)` is used to describe a Tents puzzle, where *RCs* is a list of length n, containing the tent-counts of the rows. Similarly, the list *CCs* of length m gives the tent-counts of the columns. Finally, *Ts* is a list specifying the coordinates of the trees in the form of *Row*`-`*Column* pairs. In the first two lists, `-1` stands for an unspecified

count. A solution is a list which, for each tree as listed in *Ts*, gives the direction in which its tent should be placed. The latter is specified as one of the letters `n`, `w`, `s`, or `e`. Below is a sample run of the tents program:

```
| ?- tents(t([0,-1,-1,3,-1],[1,1,-1,-1,1],[1-2,2-5,3-3,5-1,5-5]), Dirs).
Dirs = [s,w,s,n,n] ? ;
no
```

6.3 The Magic Spiral Puzzle

The Magic Spiral puzzle was set as the major assignment for the constraint course in the 2002 autumn term, having served the same role in the spring term for the DP course.

The Puzzle. In this puzzle a square board of $n * n$ fields is given. The task is to place integer numbers, chosen from the range $[1..m]$, $m \leq n$, on certain fields of the board, so that the following conditions hold:

1. in each row and each column all integers in $[1, m]$ occur exactly once, and there are $n - m$ empty fields;
2. along the spiral starting from the top left corner, the integers follow the pattern $1, 2, \ldots m, 1, 2, \ldots, m, \ldots$ (number m is called the period of the spiral).

Initially, some numbers are already placed on the board.

Figure 9 shows an instance of the puzzle for $n = 7$ and $m = 4$, as well as its (single) solution.

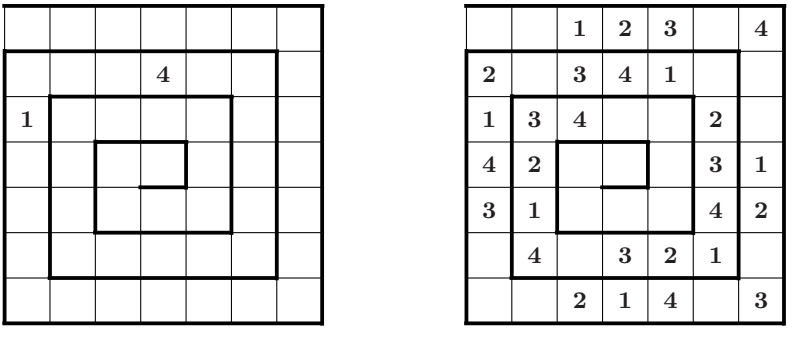


Fig. 9. A Magic Spiral puzzle and its solution

The Prolog Representation. A Magic Spiral puzzle is described by a structure of form `spiral(`n`,`m`,`*Ps*`)`, where *Ps* is a list of `i(`*Row*`,`*Column*`,`*Value*`)` triplets specifying the known elements on the board. A solution is simply a matrix represented as a list of lists of integers, where 0 means an empty field, while other integers correspond to the values on the board. Here is a sample run of the spiral program:

```
| ?- magic_spiral(spiral(7,4,[i(2,4,4),i(3,1,1)]), SpTable).
SpTable = [[0,0,1,2,3,0,4],
           [2,0,3,4,1,0,0],
           [1,3,4,0,0,2,0],
           [4,2,0,0,0,3,1],
           [3,1,0,0,0,4,2],
           [0,4,0,3,2,1,0],
           [0,0,2,1,4,0,3]] ? ;
no
```

6.4 The Snake Puzzle

The Snake puzzle was used in the spring and autumn terms of 2003 for the DP and the constraints course, respectively.

The Puzzle. A rectangular board of size $n * m$ is given. The task is to mark certain fields on the board as belonging to the *snake*, observing the following conditions:

1. The snake consists of a sequence of neighbouring fields (i.e. fields sharing a side).
2. The position of the snake head (the first field) and of the tail (last field) is given.
3. The snake can touch itself only diagonally, i.e. no snake field can have more than two snake field neighbours, but it can share vertices with other snake fields.
4. Certain fields of the table contain a number.
   (a) The snake cannot pass through fields containing a number.
   (b) If a field contains number c, then among the eight side and diagonal neighbours of this field there are exactly c which contain a snake piece. (This condition is similar to the one used in the popular minesweeper game.)

The left side of Fig. 10 shows an instance of the Snake puzzle for $n = m = 5$, with the snake head/tail marked as **H** and **T**. In the right hand side of the figure the unique solution is given, marking the fields of the snake with a filled circle.

The Prolog Representation. A Snake puzzle is represented by a structure of form `s(Size,Head,Tail,RefPoints)`. The first three arguments are all of form `Row-Column` and denote the board size, the position of the snake head and that of the snake tail. The `RefPoints` argument contains the list of reference points, describing the board positions containing a number. A reference point is represented by the structure `p(R,C,N)`, indicating that number `N` is placed in row `R`, column `C` of the board.

The solution of the puzzle, i.e. the snake, is represented by a list of neighbouring snake positions, starting with the head. Again, each position is given in the `Row-Column` format.

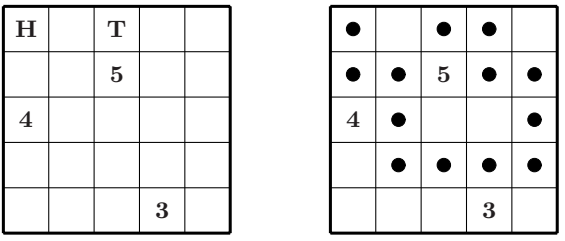


Fig. 10. A Snake puzzle and its solution

Below is a sample run of the a program solving the Snake puzzle, for the example of Fig. 10.

```
| ?- snake(s(5-5,1-1,1-3,[p(2,3,5),p(3,1,4),p(5,4,3)]), Snake).
Snake = [1-1,2-1,2-2,3-2,4-2,4-3,4-4,4-5,3-5,2-5,2-4,1-4,1-3] ? ;
no
```

6.5 Generating Puzzles

When making assignments, it is not enough to design a suitable project, one also has to provide appropriate test data, on which the student programs can be tested. In our case this means finding puzzles of the given type. One has to use a sequence of test puzzles of increasing difficulty, so that students can compete how far they can get in solving these.

The notion of difficulty cannot be precisely defined, but certain factors influencing the difficulty can easily be identified. For example, the puzzle normally becomes more difficult if its size increases, or if the number of the clues decreases.

In earlier years we used puzzles published in periodicals, with some variations. This was the case, e.g. for the Battleship assignment, where we started from a published puzzle and made it more difficult by taking away clues. In the last few years we used constraint techniques for generating puzzles, for both the declarative programming and the constraints course.

Generally, puzzles are built "backwards", in the following phases:

1. Build a solved puzzle.
2. Construct a puzzle from the solution,
3. Tune the puzzle difficulty.

Let us illustrate this on the generator program for the Magic Spiral puzzle. The generator is supplied the following parameters: the puzzle size (n), the period (m) and a difficulty grade, to be explained below. The generator is built around a solver for the puzzle, which incorporates a randomising labeling scheme.

First, the solver is invoked with the n, m parameters, and no integers placed on the board. By using randomised labeling, we arrive at a random magic spiral. For larger puzzles, it became important to limit the time spent in this process. In case of time-out, we restart the generation a few times, before giving up.

In the case of the Magic Spiral puzzle, the solution itself can be viewed as a (trivial) puzzle. We then build a proper puzzle, by taking away clues, i.e. numbers from the board. We have chosen to keep the "unique solution" property of the puzzle as long as possible.

The transformation of the puzzle is thus the following: we omit a randomly selected number from the spiral, and check, again using the solver, that the resulting problem has a single solution[7]. If the deletion leads to multiple solutions, we try other deletions. If none of the deletions is suitable, we arrive at a puzzle with a (locally) minimal set of clues, which still has the "unique solution" property.

The obtained minimal unique puzzle is treated as having a fixed difficulty grade, say grade 0. If this grade was requested, then the minimal unique puzzle is returned. If the difficulty grade is negative, we put back some of the clues, again randomly selected, onto the board. The lower the difficulty grade, the more numbers are put back.

On the other hand, if the difficulty grade is positive, then this is taken as a request to generate a non-unique puzzle, by taking away further clues from the board. Here we have adopted the principle that higher difficulty grades correspond to more solutions: for the difficulty grade g we keep taking away clues until a puzzle with at least g solutions is found. However, we exclude puzzles with too many, say more than 20, solutions.

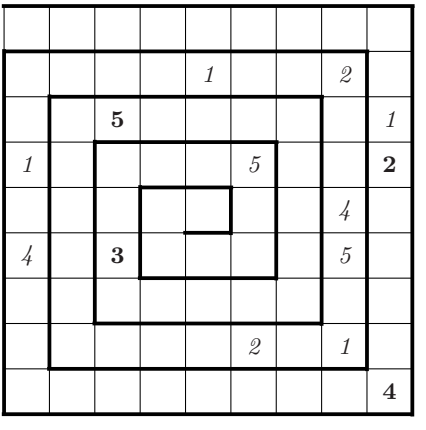


Fig. 11. A generated Magic Spiral puzzle, period 5, difficulty -5

Figure 11 shows a Magic Spiral puzzle, of difficulty grade -5, generated by the procedure described above. The numbers shown in boldface constitute the

[7] Alternatively, we may insist that the resulting problem is solvable without labeling. This latter scheme, which results in easier puzzles, was used in generating the puzzle of Fig. 11.

minimal unique puzzle, while the ones in italic are redundant clues, added to make the puzzle easier.

The puzzle in Fig. 9 has also been generated by the above procedure, using difficulty grade 0.

A similar puzzle generator was created for the Snake puzzle. Here again, we first generate a random snake. Next, we produce a trivial puzzle by inserting numbers in all non-snake fields of the board, and keep on taking away these clues until a "minimal unique" puzzle is found, etc. However, for large boards (say, above 50x50), the random snake generation using the solver proved to be infeasible. Instead, we resorted to using a simple limited backtracking search. Another speed improvement was achieved by first taking away clues in larger chunks, and halving the chunk size, when a non-unique puzzle instance was hit.

The outlined puzzle generation scheme seems to be applicable for most puzzles with a variable number of clues. However, it requires a fairly good solver to be built first. There may also be a concern that this approach results in puzzles which are biased towards the specific solver used.

7 Examination Problems

The exams are in writing, with some verbal interaction. The students are asked to solve two simpler problems selected from the following four topics: CLP(Q), CLP(B), CHR, Mercury. Next, they are asked to write an indexical and a global constraint, similar to ones given as minor assignments shown in Fig. 3. Finally, they have to solve a simple CLP(FD) problem, usually involving reification.

In the past years there were some puzzle related problems issued in this last category.

The first such task, used in 1999, is related to the Nonogram puzzle. The students were asked to write a predicate constraining a list of 0-1 variables to contain a specified sequence of blocks of 1's. This is the base constraint to be posted for each row and column of a nonogram puzzle.

The task was thus to write the following predicate:
`blocks(`*`Bits, N, Lengths`*`)`: *N* is a given integer, *Lengths* is a given list of integers, and *Bits* is a list of *N* 0..1 valued constraint variables. The list *Lengths* specifies the lengths of the blocks (maximal contiguous sequences of 1's) occurring in *Bits*. For example, the list `[0,1,1,0,1,1,1]` contains two blocks and their lengths are `[2,3]`. Examples:

```
| ?- blocks(Bits, 7, [2,3]).
       Bits = [_A,1,_B,_C,1,1,_D],
       _A in 0..1, _B in 0..1, _C in 0..1, _D in 0..1 ? ; no

| ?- blocks(Bits, 7, [2,3]), Bits=[0|_].
       Bits = [0,1,1,0,1,1,1] ? ; no
```

This task proved to be a bit too difficult, but with some help most of students could solve it. The help involved proposing some auxiliary variables: for each

element of the *Length* list introduce a variable S_i, which is the starting index of block i. Subsequently for each block i and each bit-vector position j define a Boolean variable B_{ij} which states whether block i covers position j [8].

The second exam task, from spring 2000, is linked to the Skyscrapers puzzle.

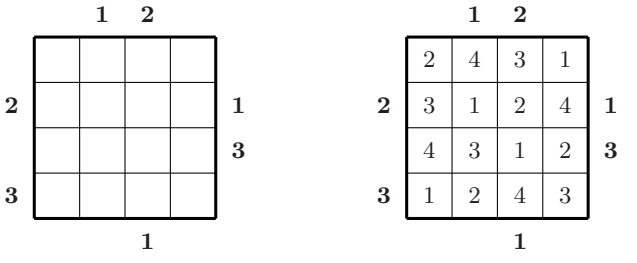


Fig. 12. A Skyscrapers puzzle and its solution

The Skyscrapers Puzzle. A permutation of integers from the `[1,n]` interval is to placed in each row and column of a square board, where n is the size of the board. Each field of the board represents a skyscraper, the number in the field is the height of the building. There are some numbers shown by the side of the square, they indicate how many skyscrapers are *visible* from these directions. A skyscraper is visible, if all buildings between it and the viewer are of smaller height. In Fig. 12 a sample puzzle is shown with its solution.

The students were given the above puzzle description and were then asked to write a CLP(FD) predicate which implements the following constraint, without using speculative disjunction (i.e. without creating choice-points).

`visible(`*List*`,` *Length*`,` *NVisible*`)`: *List* is a permutation of integers between 1 and *Length*, in which the number of elements visible from the front is the given integer *NVisible*. An element of the list is visible from the front, if it is greater than all elements preceding it. Examples:

```
| ?- visible(L,3,3).
                L = [1,2,3] ? ;  no
| ?- visible(L,3,2), labeling([], L).
                L = [1,3,2] ? ;
                L = [2,1,3] ? ;
                L = [2,3,1] ? ;
                no
```

The third exam task to be presented is related to the End View puzzle.

[8] The constraint linking the S_i and B_{ij} variables can be stated as:

S_i `in` $(j - l_i + 1)$ `..` j `#<=>` B_{ij}

where l_i is the i^{th} element of the *Length* lengths list. The j^{th} element of the *Bits* bit-vector, B_j, can then be defined through the constraint $B_j = \sum_i B_{ij}$.

The End View Puzzle. A square board of size $n * n$ is given. Numbers from the range $[1, m]$, $m \leq n$ are to be placed on the board, so that all integers in $[1, m]$ occur exactly once in each row and each column (and there are $n - m$ empty fields in each of these). As clues, numbers are placed outside the board, prescribing the first number seen from the given direction[9].

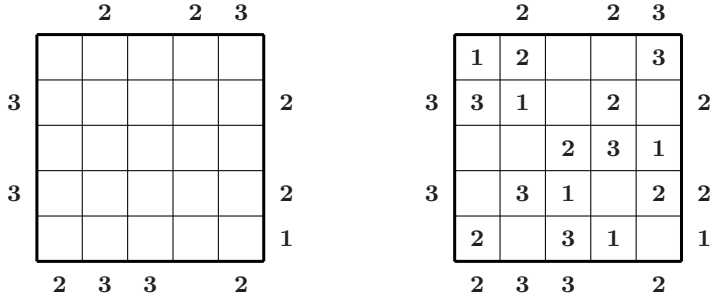


Fig. 13. An End View puzzle and its solution

Figure 13 shows a sample puzzle of this kind, for $n = 5, m = 3$, together with its unique solution. Here, for example, the number 2 placed below the fifth column prescribes that the last number placed in that column is 2, while the number 3 placed in front of row 4 requires that this number is the first in that row.

The following exam task, first set in 2001, formulates the "end view constraint" of this puzzle, assuming that empty spaces are represented by 0 values.

Write a `clpfd` constraint `firstpos(`*`List, Number`*`)`, where *`List`* contains non-negative integers, and its first positive element is equal to *`Number`*. *`Number`* is a given integer, while *`List`* is a list of constraint variables or integers. Avoid speculative disjunctions. Examples:

```
| ?- firstpos([X], 9).
                X = [9] ? ; no
| ?- firstpos([0,X,2], 1).
                X = 1 ? ; no
| ?- firstpos([A,X,2,4,3], 2), A #=< 0.
                A = 0, X in  {0}\/{2} ? ; no
```

The difficulty of this task seems to lie in the fact, that except for the first positive element, the rest of the list is not constrained. At the same time the position of the first positive element is not known. Some students needed a hint: introduce a Boolean variable for each list position, reflecting the fact that all elements before this position are zero.

[9] This puzzle is usually presented with letters A, B, C, D, etc., instead of numbers 1, 2, 3, 4, ...

8 Discussion

This section presents a brief discussion of some general issues related to the topic of this paper.

The Course. The constraint course seems to be fairly successful. In spite of the competition with less-demanding elective courses, a large number of students enrol in the course. There is a relatively large drop-out rate: in 2002 only 29 students completed the course, out of the 55 who had enrolled in it. The biggest obstacle seems to be the compulsory major assignment, which requires a considerable amount of work to complete. On the other hand, those who do submit the assignment, get very good grades: 24 students got the highest grade, 5, and only five got the next highest, 4.

The course focuses on the practical programming side of constraints, the theoretical background of CLP is not treated in detail. However, the basic notions, such as the primitive and non-primitive constraints, the constraint store, and its operations, are reiterated for each specific constraint library. This strengthens the students' understanding of the common basis of the constraint packages covered.

The discussion of the finite domain library is the most elaborate, covering a whole spectrum, from simple arithmetic constraints to user defined ones. The weakest part is on combinatorial constraints, perhaps the discussion of these could be limited to the ones actually used in the case studies. The students report that the topic of FD predicates poses the biggest challenge, but the exams prove that practically all students become capable of working with indexicals.

The application areas of constraint programming are just briefly mentioned at the beginning of the course. As one of the reviewers pointed out, a possible improvement would be to use one of the case study slots for discussing a prototype of a real-life constraint application.

The Assignments. The recently introduced minor assignments were quite popular, they gave the opportunity to practice some of the techniques described in the lectures. Students were given extra points for correct assignments, half the points were awarded even to students with late submissions.

The major assignment, which involves student competition, seems to give good motivation. Students compete not only against each other, but they would also like to beat the teacher's solution, which does happen in some cases. Usually, there is quite a bit of traffic on the course mailing list as the submission deadline approaches. The progress of some students can be seen from the questions they pose: first they write just the necessary constraints, then introduce some redundant ones, and finally try various labeling schemes and shaving.

Automated testing of assignments was found very useful. Students can submit their solutions any number of times, only the last submission is marked. The testing tool provides almost immediate feedback, at least for the minor assignments. Regarding the major assignment, where testing takes longer, we got some suggestions to improve the testing interface. For example, students asked for an option with which they can specify which test cases to run.

Logic Puzzles. Tasks involving puzzles seem to be very good means for teaching constraints. Simpler puzzles are attractive because of the very concise, logical solutions, see e.g. the knights and knaves program in Appendix A. More complex puzzles give a good opportunity to present the different phases of constraint solving: modelling, establishing constraints, enumeration of solutions. Within each phase a lot of relevant issues can be discussed, as shown in the Domino case study (Sect. 5). Students then have to go through this process themselves, when solving the major assignment.

With logic puzzles, one can try solving them by hand. Since humans rarely do backtracking search, the human process of solution often gives clues how to deduce more information, avoiding unnecessary search.

It can be argued that our course setup, with a lot of puzzles, may give the students a false impression, namely that constraint programming is only good for solving non-practical problems. Although this danger is there, I do not believe it to be a serious problem. This is demonstrated by the fact that students do recognise relevant search problems in their subsequent research and thesis work, and use CLP techniques for solving these.

Future Work. During the past decade we have accumulated quite a few constraint problems, which were formulated during the declarative programming and constraints courses. We also have numerous student solutions for these problems, in plain Prolog, Prolog with CLP extensions, as well as in SML. It would be very interesting to process the student solutions and try to find out which techniques are most often used, which are avoided, etc., giving feedback on how to improve the courses.

It would be nice to extend the local competition in solving these problems to a wider circle, both in time and in space. The former would mean that students can submit solutions to problems set in earlier terms, while the latter would enlarge the audience to any interested party outside our university/course. Obviously, the automatic assignment testing facilities of the web-based ETS system will have to be extended to support this.

9 Conclusions

Various puzzle competitions and periodicals publishing logic puzzles seem to offer an inexhaustible supply of interesting and scalable constraint problems. We have successfully used these problems in attracting and directing students' attention first towards declarative programming, and subsequently towards CLP.

In this paper we have presented an overview of the constraint course at Budapest University of Technology and Economics. We argued that logic puzzles of various kind and difficulty can be successfully used in various roles during the course: programming examples, assignments, case studies, exam tasks, etc.

We hope that the material presented here can be re-used in other constraint courses.

Acknowledgements

I am grateful to Tamás Benkő, for the numerous fruitful discussions on how to teach constraints and for developing the material for the Mercury part. Mats Carlsson was always at hand when I had problems with understanding CLP. He also kindly let me use some material from his CLP course at Uppsala University. Most major assignments presented here were originally developed for the DP course, together with Péter Hanák, Tamás Benkő, Dávid Hanák, and the student demonstrators. Dávid Hanák and Tamás Szeredi, the developers of the FDBG debugging library, produced the first version of the material describing this tool. The comments of the anonymous referees have helped a lot in improving the paper.

Thanks are also due to all the students of the course, who patiently let me learn the secrets of constraint programming while I have been pretending to teach them.

References

1. Shortz, W., ed.: Brain Twisters from the First World Puzzle Championships. Times Books (1993)
2. Shortz, W., Baxter, N., eds.: World-Class Puzzles from the World Puzzle Championships. Times Books (2001)
3. Van Hentenryck, P.: Constraint Satisfation in Logic programming. The MIT Press (1989)
4. Hanák, D., Benkő, T., Hanák, P., Szeredi, P.: Computer aided exercising in Prolog and SML. In: Proceedings of the Workshop on Functional and Declarative Programming in Education, PLI 2002, Pittsburgh PA, USA. (2002)
5. SICS, Swedish Institute of Computer Science: SICStus Prolog Manual, 3.10. (2003)
6. Colmerauer, A.: An introduction to Prolog III. Communications of the ACM **33** (1990) 69–90
7. Smullyan, R.M.: What is the Name of This Book? Prentice Hall, Englewood Cliffs, New Jersey (1978)
8. Hanák, D., Szeredi, T.: Finite domain constraint debugger. In: SICStus Manual ([5], Chapter 36). (2003)
9. Van Hentenryck, P., Saraswat, V., Deville, Y.: Design, implementation, and evaluation of the constraint language cc(FD). In Podelski, A., ed.: Constraint Programming: Basics and Trends. Springer, Berlin, Heidelberg (1995) 293–316
10. Szeredi, P.: CLP resources (2003) `http://www.cs.bme.hu/~szeredi/clp.html`.
11. Harvey, W., Schimpf, J.: Eclipse sample code: Domino (2000) `http://www-icparc.doc.ic.ac.uk/eclipse/examples/domino.ecl.txt`.
12. Martin, P., Shmoys, D.B.: A new approach to computing optimal schedules for the job-shop scheduling problem. In: Proceedings of the 5th International IPCO Conference. (1996) 389–403
13. Simpson, S.: Nonogram solver (2003) `http://www.comp.lancs.ac.uk/computing/users/ss/nonogram/`.
14. Walsh, T.: Teaching by Toby Walsh: Nonogram solver (2000) `http://www-users.cs.york.ac.uk/~tw/Teaching/nonogram.html`.

A The Knights, Knaves and Normals Puzzle

This appendix describes a CLP(FD) program for solving a generic puzzle type, made popular by Raymond Smullyan, e.g. in [7].

The Puzzle. There is an island, on which there are three kinds of natives: *knights* always tell the truth, *knaves* always lie, and *normals* sometimes lie and sometimes tell the truth. Some local people make statements about themselves and we have to find out what kind of natives they are.

For example, in puzzle 39 from [7] three natives make the following statements:

A says: I am normal.
B says: That is true.
C says: I am not normal.

We know that A, B, and C are of different kinds. Determine who they are.

The Solution. In Fig. 14 we present a CLP(FD) program for solving puzzles of the above kind. Knights tell true statements, i.e. ones with the truth value 1, therefore knights are represented by the integer value 1. For analogous reasons, knaves are encoded as 0. Finally, a normal person is represented by the value 2. Statements are Prolog structures with the following syntax:

```
St ---> Ntv = Ntv | Ntv says St | St and St | St or St | not St
```

Here `St` denotes a statement, while `Ntv` is an integer or a constraint variable representing a native, i.e. one in the `0..2` range. The atoms "`says`", "`and`", "`or`", and "`not`" are declared operators. As syntactic sweeteners, we allow statements of form "`Ntv is Kind`", where `Kind` is one of the atoms `knight`, `knave`, or `normal`. This gets transformed to "`Ntv = Code`", where `Code` is the numeric encoding of `Kind`. To make the output more readable, we define an appropriate `portray` predicate.

The entry point of the program is the `says/2` predicate. The Prolog goal "`X says St`" expresses the constraint that native `X` says the statement `St`. With this interface the above concrete puzzle instance can be transformed into an "almost natural language" Prolog goal, as shown in Fig. 15.

B Performance Evaluation of the Domino Puzzle Solvers

In this Appendix we discuss the performance of various solutions of the domino puzzle, described in Sect. 5.

Test Cases. Table 3 shows the test sets used in the evaluation of the two domino solver variants. Altogether we have 63 test cases, grouped into four sets, according to their difficulty level. The difficulty seems to be most correlated to the number of solutions. The test cases can be downloaded from [10].

```
:- op(700, fy, not).              :- op(800, yfx, and).
:- op(900, yfx, or).              :- op(950, xfy, says).


% Native A can say sentence Sent.
A says Sent :- truth(A says Sent, 1).

% native(X): X is a native:
native(X) :- X in 0..2.

% truth(S, Value): The truth value of sentence S is Value.
truth(X = Y, V) :-
        native(X), native(Y), V #<=> (X #= Y).
truth(X says S, V) :-
        native(X), truth(S, V0),
        X #= 2 #\/              % Either X is normal or
        (V #<=> V0 #= X).       % the truth of what he says (V0) should be
                                % equal to him being a knight (X)
truth(S1 and S2, V) :-
        truth(S1, V1), truth(S2, V2), V #<=> V1 #/\ V2.
truth(S1 or S2, V) :-
        truth(S1, V1), truth(S2, V2), V #<=> V1 #\/ V2.
truth(not S, V) :-
        truth(S, V0), V #<=> #\ V0.
truth(X is Kind, V) :-
        code(Kind, Code), truth(X = Code, V).

% code(Kind, Code): Atom Kind is encoded by integer Code.
code(knave, 0).   code(knight, 1).   code(normal, 2).

portray(Code) :- code(Kind, Code), write(Kind).
```

Fig. 14. The CLP(FD) program for the knights, knaves, and normals puzzle

```
| ?- A says A is normal,
     B says A is normal,
     C says not C is normal,
     all_different([A,B,C]),
     labeling([], [A,B,C]).
A = knave, B = normal, C = knight ? ; no
```

Fig. 15. A run of the knights, knaves, and normals puzzle

Tuning Parameters. We evaluate the effect of various parameters on the performance of the presented solutions. The following is a list of tunable parameters and their possible settings. We will later refer to these using the phrases typeset in italics.

- labeling:
  - which kind of variables to label (only in the compass model): domino variables (DV) or compass variables (CV),
  - variable selection options: *leftmost*, *ff*, *ffc*

Table 3. Test sets for the Domino problem

Test set	Number of Tests	Best average time (sec)	Number of solutions (average)	Description
`base`	16	0.08	19	very basic tests for $d = 1..25$
`easy`	24	0.38	13	easy tests mostly of size $d = 15..25$
`diff`	22	41	110	difficult tests of size $d = 28, 30$
`hard`	1	332	1536	a very hard test of size $d = 28$

- shaving frequency (*freq. =*) *1, 2, 3, ..., once* (only before labeling), *none* (no shaving at all)
- values shaved (*shave =*)
  - (compass model): *[n,w], [n]*
  - (border model): *[0], [1], [0,1]*
- implementation of the base constraint:
  - (compass model): implementing $X = c \Leftrightarrow Y = d$ through reification (*reif*), or by using an indexical for implication (*impl*), called twice, or by using an indexical for equivalence (*equiv*)
  - (border model): implementing the $\sum_n X_i = 1$ constraint using the library predicate `sum/3` (*libsum*), or using an FD-predicate for $n \leq 5$ and `sum/3` otherwise (*fdsum*)[10].

Performance. Table 4 presents performance results for some combinations of the above parameters, run with SICStus Prolog version 3.10.1 on a VIA C3 866 MHz processor (roughly equivalent to a 600 MHz Pentium III). For each test case all solutions were collected, with a CPU time limit of 1800 seconds (except for the hard test case, where the limit was 7200 seconds).

The table has three parts, presenting the results for the border model, for the compass model with DV labeling, and for the compass model with CV labeling. For each part, a full header line shows those parameter-settings which produced the best results. In subsequent lines the first column shows a (possibly empty) variation in the settings, while the subsequent columns show the performance results. Here two figures are given: the first is the total run-time in seconds, while the second one is the total number of backtracks, both for all test cases in the test-set. The $>$ symbol in front of the numbers indicates that there was a time-out while running at least one test case within the test-set. Note that backtracks during shaving are not counted.

Evaluation. Overall, the simpler border model seems to be faster than the compass model, except for the `hard` test case.

The border model involves Boolean variables only, so obviously the `ff` variable selection gives exactly the same search space as the `leftmost` one (as indicated by the same backtrack count). Intuitively, the `leftmost` strategy should

[10] The threshold of 5 seems to be optimal for SICStus Prolog. Additional measurements (not shown in the Table) give poorer results for thresholds of 4 and 6.

Table 4. Performance of the Domino solutions

Variation	`base`		`easy`		`diff`		`hard`	
border model, `leftmost`, freq. = 2, shave = [1], fdsum								
	1.56	1	9.26	8	910	1399	1373	2254
`ff`	1.53	1	9.18	8	910	1399	1351	2254
`ffc`	1.56	1	9.85	8	1792	2838	2181	3732
freq. = 1	1.63	1	9.59	3	1100	787	1642	1277
freq. = 3	1.53	1	9.28	20	931	2436	1370	3851
shave = [0]	1.29	2	11.04	103	1532	10719	2306	17300
shave = [0,1]	1.36	1	9.45	7	904	1324	1370	2150
libsum	2.75	1	13.85	8	1193	1399	1782	2254
freq. = once	1.41	1	10.93	1663				
freq. = none	2.68	818	49.73	21181				
compass model, DV labeling, `ff`, freq. = 3, shave = $[n, w]$, impl								
	3.17	1	18.57	19	2536	3597	332	477
`leftmost`	3.16	1	18.95	38	3389	8782	932	2547
`ffc`	3.19	1	17.34	17	>3790	>5374	2722	4095
freq. = 2	3.28	1	18.56	13	2516	2074	343	288
freq. = 4	3.15	1	19.06	41	2543	5720	353	727
shave = $[n]$	2.92	13	23.84	75	3971	11012	737	1820
reif	3.94	1	22.10	19	2670	3597	349	477
equiv	2.99	1	18.83	19	2691	3597	354	477
compass model, CV labeling, `ff`, freq. = 3, shave = $[n, w]$, impl								
	3.18	1	16.61	21	1684	2398	2570	3907
`leftmost`	3.18	1	16.59	21	1684	2398	2571	3907
`ffc`	3.18	1	19.28	25	>6606	>9840	>7200	>9343
shave = $[n]$	2.83	5	20.84	73	2600	6889	4609	12697

be faster, as its variable selection algorithm is simpler. In practice, both `ff` and `leftmost` settings give the same results (with a 2% variation, probably due to measurement errors).

For both models the best shaving frequency seems to be around 2-3. With a single shaving or no shaving at all, the performance is very poor.

In the border model, shaving the *[1]* value is bound to cause two constraints to fully instantiate all their variables, hence its performance is better than that of shaving *[0]*. On the other, hand shaving with both *[0,1]* gives a negligible reduction in the number of backtracks, and practically the same time as the variant shaving only the value *[1]*. In the compass model, shaving a single compass value (*[n]*), produces results inferior to those achieved when two non-opposite values (*[n,w]*) are shaved.

There are some open issues. For example, it would be interesting to find out those characteristics of the `hard` test case, which make the compass model more suitable for it. It also needs to be investigated why do we get very poor performance with the `ffc` variable selection strategy.