Teaching Constraints through Logic Puzzles

Conference Paper *in* Lecture Notes in Computer Science · June 2003

DOI: 10.1007/978-3-540-24662-6_11 · Source: DBLP

Péter Szeredi
Budapest University of Technology and Economics

Teaching Constraints through Logic Puzzles

Péter Szeredi
szeredi@cs.bme.hu

Dept. of Computer Science and Information Theory,
Budapest University of Technology and Economics
H-1117 Budapest, Magyar tudósok körútja 2.

IQSYS
Information Systems Ltd.
H-1135 Budapest, Csata u. 8.

Abstract. The paper describes the experiences of seven years 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. In the last few years, here in Hungary, four monthly periodicals appeared which publish solely logic puzzles. There are numerous web-sites offering this type of amusement. Best puzzle solvers gather this autumn in the Netherlands for the 12th time to compete in the World Puzzle Championship. Puzzles from earlier Championships were published in e.g. [6, 7].

Personally, I found logic puzzles quite intriguing, and started to use them in my lectures on (constraint) logic programming at the Budapest University of Technology and Economics (BUTE). This paper gives an account of the constraint course, with a brief summary of the preceding Prolog course. While describing various aspects of the course, I highlight those parts where logic puzzles can be used.

The paper is structured as follows. First, in Section 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 and evaluated. Section 6 describes two puzzles which were issued as major assignments in the past two years, while Section 7 gives some examination tasks related to puzzles. In the last two sections we discuss some lessons learned and conclude the paper.

2 Background

The paper presents an elective constraint logic programming course for (undergraduate) students of informatics at BUTE[1]. It presupposes the knowledge of

[1] Occasionally the course is taken by students of electrical engineering, and by post-graduates

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) [2], 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 Sections 5 and 6. 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 enroll in the electable constraints course immediately, in the semester following their DP course. For example, 55 students registered for the constraints course in fall 2002, of which 38 took DP in the previous semester[2].

3 The structure of the course

The actual title of the constraint course discussed in this paper is “High Efficiency Logic Programming”. This course has been presented since 1997, seven times up to now (in the spring term till 2000, and in the fall term since 2000). 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, in the context of the constraint libraries of SICStus Prolog [8]. However, from the very start, the focus shifted to the second topic. In fact, the order of the two parts is reversed, so the Mercury language is presented at the end of the semester.

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

[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.

the term. To motivate and help the students in doing programming exercises during the term, minor assignments were introduced in the last, 2002 fall term. There were four of these, with a due date of approximately two weeks after issue.

The layout of the 2002 fall semester is shown in Table 1, with approximate time schedule, and the number of (A4 size) handout slides for each topic. The *unit* of the time schedule is 45 minutes, i.e. there are 2 units per lecture.

We will now focus on the constraint part of the course, discussing in turn the topics 1.-5. listed in the table.

	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

Table 1. The layout of the course

The first topic deals with SICStus 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” demo 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 similar is 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.

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).

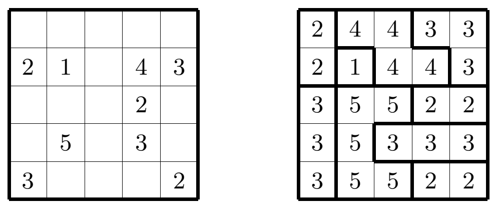
The second part starts with a brief overview of the CLP($\mathcal{X}$) scheme, introducing the notions of constraint store, primitive and non-primitive constraints. These are then exemplified by discussing the CLP(Q/R) extension, culminating in Colmerauer’s Perfect Rectangle Problem [1]: tiling a rectangle 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 the Constraint Handling Rules library of SICStus Prolog. 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 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 [9], shown in the Appendix.

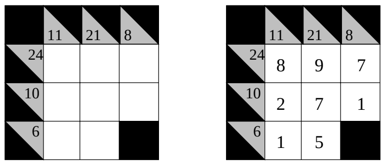
	CLP(FD) subtopic	Time	Slides
4.1.	CSP overview, CLP(FD) basics	3 units	23 slides
4.2.	Reification, propositional constraints, labeling *Assignment 2:* Cross sums puzzle	3 units	25 slides
4.3.	Combinatorial constraints *Major Assignment:* Magic Spiral puzzle	1 unit	12 slides
4.4.	User defined constraints *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

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

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 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:

```
| ?- Table = [[x\x, 11\x,21\x, 8\x],
              [x\24,   _,   _,   _],
              [x\10,   _,   _,   _],
              [x\6,    _,   _, x\x]], cross_sums(Table, 9).
Table = [[x\x, 11\x,21\x,8\x],
         [x\24,8,   9,   7  ],
         [x\10,2,   7,   1  ],
         [x\6, 1,   5,   x\x]] ? ;
no
```

Fig. 2. The Cross Sums puzzle

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 fall 2002, this was the Magic Spiral puzzle (see Sect. 6.2), exactly the same assignment that had been issued for the Declarative Programming course in the preceding semester.

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

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 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 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:

```
:- use_module(fdcheck).      % This is the file containing the checker
:- fd_pred_semantics('z>max(x,y)'(X,Y,Z), Z>max(X,Y)).
:- fd_test_range(1, 2).      % Try all combinations in this range

'z>max(x,y)'(X,Y,Z) +: ...   % Here comes the FD-predicate
```

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 [3] 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 [11] is presented. This gives an opportunity to discuss the issue of disjunctive constraints and to introduce the technique of dual labeling. 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 former involves placing rectangles of size $1 \times n$, called battleships, on a rectangular board (sea). Battleships can be placed horizontally or vertically, and they can be of different colours. For each length (n) 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 can not 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 (i.e. no battleship piece there). The Battleships case study (program, test-data, results) can be downloaded from [10].

The third case study is discussed in detail in the next section.

5 A case study: the Domino puzzle

The Domino puzzle was issued as the major assignment in spring 2000.

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 domino boundaries. The task is to find out the tiling, i.e. to reconstruct the domino boundaries.

Figure 4 shows a sample board, its solution, and the Prolog format of these.

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, filled in with compass point abbreviations: one of `n`, `w`, `s`, or `e`. These return values specify, for each square, whether it is a northern, western, southern, or eastern half of a domino.

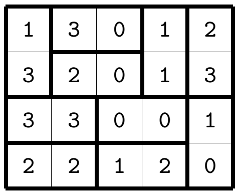
In the case study two alternative solutions are presented: one based on the *compass* model, and another one using the *border* model. An independent Eclipse solution to this puzzle, found in [4], uses the latter model.

The compass model. This model follows naturally from the expected output format: each square of the board is assigned a *compass* variable specifying the compass point of the half domino it is covered with. The compass variables are

The problem:

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

The (only) 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. 4. A sample Domino problem for $d = 3$

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/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. 4 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. 4 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. 5. All solutions have the same pruning behaviour. Our measurements (see later, in Table 4) show, that the second is the fastest variant. This involves 4 indexicals, as opposed to the the third, single FD-predicate implementation which has only 2. 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. 5. FD-predicates for implementing equivalence constraints

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 ($1 \leq y \leq max_y, 1 \leq x \leq max_x$). 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 western border of the board, we use similarly constant 0 valued variables S_{0x} and E_{y0}.

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 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. 4 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 $\sum_n X_i = 1$ constraint: one using the `sum/3` global constraint of SICStus Prolog, the other using 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.

Shaving. The shaving technique [5] was introduced to students in the Battleship puzzle case study. There it 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. The process of shaving 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 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, 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. In the compass model we thus try shaving with $[n, w]$ and $[n]$.

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

Performance evaluation. Table 3 shows the test sets used in the evaluation of the two domino solver variants. Altogether we have 63 test cases, which were grouped into four sets, based on the performance of the solvers. The difficulty seems to be most correlated to the number of solutions. The test cases can be downloaded from [10].

We evaluate the effect of various parameters on the performance of the presented solutions. The following is a list of tunable parameters and their 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*

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 $d = 28$

Table 3. Test sets for the Domino problem

- shaving frequency (*freq.* =) *1*, *2*, *3*, ..., *once* (only before labeling), *none* (no shaving)
- values shaved (*shave* =)
  - (compass model): *[n,w]*, *[n]*
  - (border model): *[0]*, *[1]*, *[0,1]*
- base constraint implementation:
  - (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*)[3].

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).

Table 4 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.

The case study is now concluded with a brief evaluation of the results. 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

[3] 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.

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

Table 4. Performance of the Domino solutions

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 [1] variant.

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.

6 Major assignments

In 1997 and 1998 the major assignments were the same: writing a Nonogram solver [12]. 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 section. In this section the major assignments of the last two years are briefly described.

6.1 The Tents 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 6 shows an instance of the puzzle for $n = m = 5$, together with its solution, where the trees are denoted by Υ and the tents by the Δ symbol. The numbers in front of the board and above it are the tent counts for the rows and the columns.

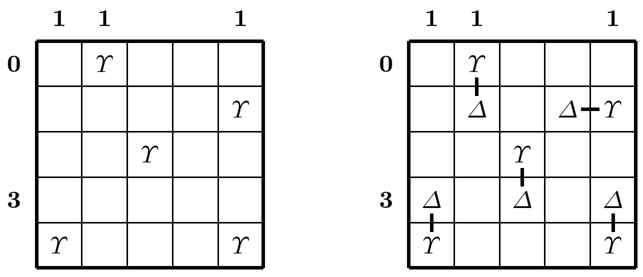


Fig. 6. A Tents Puzzle and its solution

The puzzle is described by a Prolog structure `t(`*RCs*`,`*CCs*`,`*Ts*`)`, 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, while *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 `s`, `w`, `n`, 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
```

6.2 The Magic Spiral 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 such 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 7 shows an instance of the puzzle for $n = 7$ and $m = 4$, as well as its (single) solution.

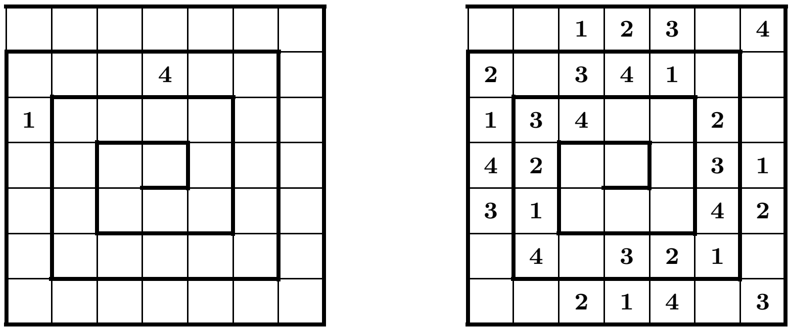


Fig. 7. A Magic Spiral Puzzle and its solution

The puzzle is specified as a structure `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]|...] ? ;
no
```

7 Examinations

The exams are in writing, with some verbal interaction. The students are asked to solve two simpler problems selected, for the given exam, 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 continuous sequences of 1's) in *Bits*. For example `[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 ?

| ?- blocks(Bits, 7, [2,3]), Bits=[0|_].
        Bits = [0,1,1,0,1,1,1] ?
```

This task proved to be a bit too difficult, but when an auxiliary predicate was proposed, most students could solve it.

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

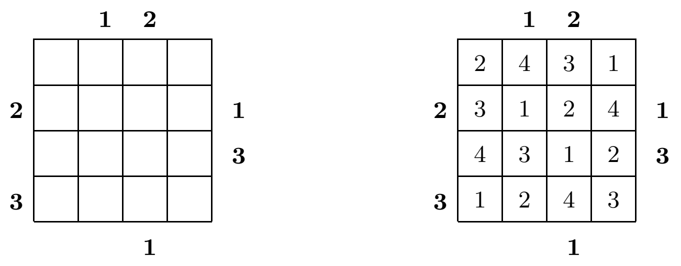


Fig. 8. A Skyscrapers puzzle and its solution

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. 8 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:

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

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 enroll in the course. There is a relatively large drop-out rate: last year 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: last year 24 students got the highest grade, 5, and only five got the next highest, 4.

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. There was quite a bit of traffic on the course mailing list as the submission deadline approached. The progress of some students can be seen from the questions they pose: first they write just the necessary constraints, then they introduce some redundant ones, and finally they try various labeling schemes and shaving. There were a lot of questions regarding shaving, this seems to be one of the most difficult topics of the course.

Automated testing of assignments was found very useful. Students can submit their solutions any number of times, only the last submission is marked. Students like the testing tool as they get almost immediate feedback, at least for the minor assignments. Regarding the major assignment, we got some suggestions to improve the testing interface, so that students 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 the Appendix. More complex puzzles give a good opportunity to present the phases of solving constraint problems: 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.

Generating puzzles. Logic puzzles made for humans are not big enough for computer programs to compete. On the other hand, a good puzzle solver can often be used for generating puzzles. This approach was actually used in the last few years for generating test cases for both the DP and the constraints course.

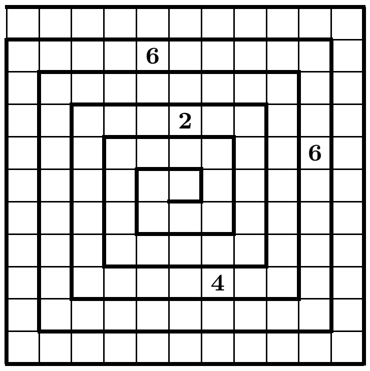


Fig. 9. A larger Magic Spiral puzzle ($n = 11, m = 6$, difficulty=10)

For example, a uniquely solvable Magic Spiral puzzle is generated from the following parameters: the size (n), the period (m) and a difficulty grade on the scale 0..10. The solver is invoked with the n, m parameters, and no integers placed on the board. By using a randomizing labeling scheme we arrive at a random magic spiral. In the next step we omit a randomly selected number from the spiral, and check, again using the solver, that the resulting problem has a single solution[4]. If the deletion leads to multiple solutions, we try other deletions. By repeating this process we arrive at a (locally) minimal set of numbers which

[4] Alternatively, we may insist that the resulting problem is solvable without any labeling (but with a single shaving pass). This latter scheme was used in generating the puzzle of Fig. 9.

still result in a unique solution. If difficulty 10 was requested, we return this as the puzzle. Otherwise we put back some numbers, again randomly selected (for difficulty 0 we put back half the numbers taken away, for intermediate difficulties, proportionally less). Figure 9 shows a puzzle, difficulty grade 10, generated by the above method.

9 Conclusions

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

We believe that the use of puzzles in all these roles contributed to the success of the course.

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 material from his CLP course at Uppsala University. The major assignments presented here were originally developed for the DP course, together with Péter Hanák, Tamás Benkő, and Dávid Hanák. Dávid Hanák and Tamás Szeredi, the developers of the FDBG debugging tool, produced the first version of the material describing this tool.

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

References

[1] Alain Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, July 1990.

[2] Dávid Hanák, Tamás Benkő, Péter Hanák, and Péter Szeredi. Computer aided exercising in Prolog and SML. In *Proceedings of the Workshop on Functional and Declarative Programming in Education, PLI 2002, Pittsburgh PA, USA*, October 2002.

[3] Dávid Hanák and Tamás Szeredi. Finite domain constraint debugger. In *SICStus Manual (Chapter 36)* [8].

[4] Warwick Harvey and Joachim Schimpf. Eclipse sample code: Domino, 2000. `http://www-icparc.doc.ic.ac.uk/eclipse/examples/domino.ecl.txt`.

[5] Paul Martin and David B. Shmoys. A new approach to computing optimal schedules for the job-shop scheduling problem. In *Proceedings of the 5th International IPCO Conference*, pages 389–403, 1996.

[6] Will Shortz, editor. *Brain Twisters from the First World Puzzle Championships*. Times Books, February 1993.

[7] Will Shortz and Nick Baxter, editors. *World-Class Puzzles from the World Puzzle Championships.* Times Books, May 2001.
[8] SICS, Swedish Institute of Computer Science. *SICStus Prolog Manual, 3.10*, January 2003.
[9] Raymond M. Smullyan. *What is the Name of This Book?* Prentice Hall, Englewood Cliffs, New Jersey, 1978.
[10] Péter Szeredi. CLP resources, 2003. `http://www.cs.bme.hu/~szeredi/clp.html`.
[11] P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(FD). In A. Podelski, editor, *Constraint Programming: Basics and Trends*, pages 293–316. Springer, Berlin, Heidelberg, 1995.
[12] Toby Walsh. Teaching by Toby Walsh: Nonogram solver . `http://www-users.cs.york.ac.uk/~tw/Teaching/nonogram.html`.

Appendix: 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 [9].

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 [9] 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. 10 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
```

where `St` denotes a statement, and `Ntv` an integer or a constraint variable representing a native, i.e. one in the `0..2` range (`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 "sweeten" the output, we define an appropriate `portray` predicate.

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

| ?- 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. 10. The CLP(FD) program for the knights, knaves, and normals puzzle