Constraint Programming: Assignment no. 9

Marco Kuhlmann, M.Sc., Dipl.-Inform. Guido Tack

http://www.ps.uni-sb.de/courses/cp-ss07/

This is the fourth and final graded assignment. The submission deadline is Wednesday, June 27th, 23:59 CEST. Please submit solutions for the individual exercises as source code, packed into a `tar.gz` or `zip` archive, to `tack@ps.uni-sb.de`.

This week's assignment is about modeling using finite set constraints.

Exercise 9.1 (The social golfers problem)

The social golfers problem is to schedule $g \cdot s$ golfers into g groups of s players over w weeks such that no golfer plays in the same group with any other golfer more than once.

Your task in this exercise is to implement the constraint model for the social golfers problem that was given in the lecture. (Recall that this model uses a list of set variables S_j, one for each of the $w \cdot g$ groups.) The instances that you should evaluate your implementation on are those that have $1 \leq w \leq 10$, $g = 8$, $s = 4$.

a) Implement the constraint model using Gecode/J. For the branching strategy, use `SETBVAR_NONE` and `SETBVAL_MIN`. What is the largest value of w for which you still find a solution in reasonable time?

b) Implement a different branching strategy as follows: Choose the first player i and the first set variable S_j for which it is not yet decided whether i plays in group S_j or not. Then, assign i to S_j in the left branch, and exclude i from S_j in the right branch. What is the largest value of w for which you still find a solution now? (There is a branching strategy in Gecode/J that implements this semantics.)

Exercise 9.2 (Binary relations)

In the lecture, you have seen how to implement binary relation variables on top of finite set variables. In this exercise you are going to build an actual implementation using Gecode/J.

We will need the following constraints on binary relations:

a) reflexivity ($\forall x.Rxx$)

b) transitivity ($\forall x, y, z.Rxy \wedge Ryz \Rightarrow Rxz$)

c) anti-symmetry ($\forall x, y.Rxy \wedge Ryx \Rightarrow y = x$)

d) converse ($\forall x, y.Rxy \Rightarrow R'yx$)

You find a definition of transitivity on the slides for Lecture 9. It uses selection constraints, which are available as `selectUnion` and `selectInterIn` in Gecode/J.

Anti-symmetry and the converse relation can be defined in a similar way. Here, we show you the bigger part of how to derive an implementation in terms of selection constraints from the definition:

$$\begin{aligned} R \text{ anti-symmetric} &\Leftrightarrow & \forall x.\forall y.Rxy \wedge Ryx &\Rightarrow x = y \\ &\Leftrightarrow & \forall x.\forall y \in Rx.x \neq y &\Rightarrow \neg Ryx \\ &\Leftrightarrow & \forall x.\forall y \in Rx \setminus \{x\}. &\neg Ryx \\ &\Leftrightarrow & \forall x.\forall y \in Rx \setminus \{x\}. &x \notin Ry \end{aligned}$$

$$\begin{aligned} R \text{ converse of } R' &\Leftrightarrow & \forall x.\forall y.Rxy &\leftrightarrow R'yx \\ &\Rightarrow & \forall x.\forall y \in Rx. &R'yx \\ &\Leftrightarrow & \forall x.\forall y \in Rx. &x \in R'y \end{aligned}$$

Note that for the converse, we only show one direction here. The other direction is completely symmetric. For both examples, the last step is missing: transforming part of the formula into selection constraints.

Attention: The indices used by the selection constraints start at 0!

a) Complete the definitions above using selection constraints. Follow the transitivity example from the slides. The goal is to have only one universal quantifier left, which can then be implemented by iteration.

b) Implement the constraints in Gecode/J using the skeleton we provide. All the places where you should add code are marked with a comment.

Exercise 9.3 (Enumerating trees)

In this exercise, you will define a binary relation R that is constrained to have tree-shape. Think of R as the *reachability relation* in the tree. Each value stands for a node. For two nodes x and y, we have Rxy if and only if y is (reflexively, transitively) reachable from x.

More formally, a binary relation R is a tree if and only if it has the following three properties:

a) It is a partially ordered set, i.e., it is reflexive, transitive, and anti-symmetric.

b) For each value x, $\{y|Ryx\}$ is totally ordered.

c) There is exactly one value x such that for all y, Rxy.

You should have no difficulty implementing property a). For property b), first implement a method `public void total(Space home, SetVar s)` that posts the constraint that R is total on the set S, i.e., $\forall x, y : x \in S \wedge y \in S \wedge \neg Rxy \Rightarrow Ryx$ (use reification to achieve this). Now define the converse relation R^{-1}, and post the constraint that R must be total for each $R^{-1}x$.

For part c), use reification to implement $\forall x : Rx = U \Leftrightarrow x \in root$ for some set variable *root*. Constrain the result to have exactly one element.

You can use the branching method we provide in `RelVar.java`.

Use your program to find out how many trees of size 1 to 5 (according to this axiomatization) there are, and compare your result with the corresponding entry in the Database of Integer Sequences:

`http://www.research.att.com/~njas/sequences/`.