KTH Information and Communication Technology

Constraint Programming (ID2204), Spring 2015
Christian Schulte

Assignment 3

The different tasks on this assignment are marked with either **EXP** for an *exploration* task or **SUB** for a *submission* task. Only submission tasks are to be submitted. Submission for all these tasks is by Email to `cschulte@kth.se`:

- Use `ID2204: A3` as Email subject.
- Clearly state in the Email who you are (that is, name and personnummer). Do that for all group members.
- Make sure that all group members are also receivers of the Email (via cc). By that we can simply reply to the Email and everybody sees our comments.
- When you are asked to develop a program, you have to submit the actual source code in a single file without additional include files. For each task it is specified which name the file must have. If the task also has additional questions you have to answer these questions as comments in the source code!
- Submission deadline is **2015-05-11, 08:00**.

More information is available on the course's homepage.

1 Reification (EXP)

We briefly mentioned *reified* constraints in one of the earlier lectures. The reified constraint

$$(c \leftrightarrow b = 1) \wedge b \in \{0, 1\}$$

for the constraint c and the *control variable* b (b is a Boolean 0/1 variable) can be propagated as follows:

- If c holds, propagate $b = 1$.
- If c does not hold, propagate $b = 0$.
- If $b = 1$ holds, propagate c.

- If $b = 0$ holds, propagate the negation of c (propagate that c does not hold).

In Gecode, simple relation and linear constraints (and some others) have reified versions that take an extra `BoolVar` argument. The extra argument is the control-variable described above. Please consult the documentation for finding out which constraints have reified versions.

The main reason for reification is to combine constraints. The disjunction $c_1 \vee c_2$ of the constraints c_1 and c_2 can be expressed by reifying each constraint:

$$(c_1 \leftrightarrow b_1 = 1) \wedge b_1 \in \{0,1\} \qquad (c_2 \leftrightarrow b_2 = 1) \wedge b_2 \in \{0,1\}$$

together with

$$b_1 + b_2 \geq 1$$

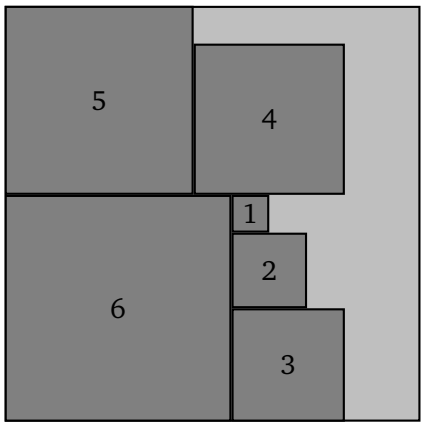
- How would you express the conjunction of two constraints via reification?
- Given are the reified constraints $(c_i \leftrightarrow b_i = 1) \wedge b_i \in \{0,1\}$ for $0 \leq i < n$. How do you express that at least $1 \leq k \leq n$ constraints hold? Which propagator would you use to implement this?

2 Square Packing (SUB, 3 points)

For a given n ($n > 1$), the square packing problem consists of finding an enclosing square of size $s \times s$ such that the n squares of sizes $1 \times 1, 2 \times 2, \ldots, n \times n$ are packed inside the large $s \times s$ square and that s is minimal. This assignment develops a solution using reified constraints to place all squares on the board such that they do not overlap and that the n squares fit inside the $s \times s$ square.

The goal is to write a script that finds a smallest value for s and computes for each square its position (that is, its x and y coordinate) on the enclosing $s \times s$ square.

An example best solution for $n = 6$ where $s = 11$ is:



We are going to use techniques for solving the square packing problem described in the following paper: Helmut Simonis, Barry O'Sullivan, *Search Strategies for Rectangle Packing*, in: Peter J. Stuckey, CP 2008, pages 52–66, 2008. You have to read the paper and try the techniques described in the paper. The paper and slides are available from Helmut Simonis' homepage.

The paper also addresses the more general case of packing n squares into a rectangle of dimension $w \times h$. To keep the assignment simple, we just consider the special case where $w = h$ (which we then call $s = w = h$).

Furthermore, the model described in the paper uses the `disjoint2` constraint for enforcing that the squares do not overlap, and the `cumulative` constraint as an implied constraint. In this assignment, we are going to use a naive formulation of the problem that expresses both constraints by a collection of simple reified constraints.

1. The script needs to create an `IntVar` that gives the size s of the surrounding square and two `IntVarArrays` of integer variables which give the respective x and y coordinates for each square to be packed.

   Think carefully how large a coordinate for a square can be! A good idea is to define a static member function `size` that returns the size of the square with number `i`. It is helpful for branching to make sure that the square with number 0 has size n, with number 1 size $n-1$, and so on.

2. Express with reification that no two squares overlap. Two squares s_1 and s_2 do not overlap, iff

   (a) s_1 is left of s_2, or
   (b) s_2 is left of s_1, or
   (c) s_1 is above s_2, or
   (d) s_2 is above s_1.

   To express these geometrical relations, use reified constraints taking the coordinates and sizes of the squares into account.

3. Consider a column with coordinate x. Then you see that the sum of the sizes of the squares occupying space at column x must be less than or equal to s.

   Express this for all columns by using reified constraints. Do the same also for all rows. As `s` might not be assigned a value yet, you might have consider all columns (and rows) from 0 to `s.max()`.

   Tip: The reified `dom` constraint might come in handy here.

4. Read the paper and think carefully which additional constraints you need to post. Remember, we only deal with the case where width and height have the same value s.

   You should at least incorporate the following ideas described in the paper:

(a) Problem decomposition (Section 2.1).

(b) Symmetry removal (Section 2.2).

(c) Empty strip dominance (only initial domain reduction, Section 4.1).

(d) Ignoring size 1 squares (Section 4.2).

Tip: the sum of the first n square numbers can be computed as:

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

5. Design a good branching heuristic. Aspects you want to take into account are:

   (a) Always branch on s first.

   (b) Try to first assign all x-coordinates, then all y-coordinates.

   (c) Try bigger squares first.

   (d) Try to place squares from left to right (and top to bottom, respectively).

   (e) Also read Section 3 in the paper and think which of the strategies can be easily expressed with normal variable value branchings. Some of them can not be expressed, but in the next assignment A04 you are going to develop a custom branching that follows some of the ideas described in Section 3.6 of the paper.

Submit as `square.cpp`.

3 Implementing a no-overlap propagator (SUB, 2 points)

In the previous task you have posted $O(n^2)$ reified propagators to enforce that n squares do not overlap. In this task you are going to implement a propagator for the constraint `nooverlap`(x, w, y, h) which enforces that a collection of rectangles at position (x_i, y_i) with width w_i and height h_i do not overlap.

On the course homepage you find a template file `no-overlap.cpp` for the propagator. You only have to implement the `propagate` function, all other functions are already provided.

Use the same logic for propagation as used for the reified constraints from before. For example, when your propagator finds out that two rectangles r_i and r_j overlap in x direction and that r_i cannot be above r_j, then the propagator must enforce that r_j must be below r_i.

Make sure that your propagator correctly reports subsumption (a criterion for subsumption that is easy to check is that no two rectangles overlap any longer. The check can be done while performing propagation).

The technical details on how to implement a propagator in Gecode are available in Part P of "Modeling and Programming with Gecode" available from Gecode's webpage.

Submit as `no-overlap.cpp`.

4 Magic sequence (EXP)

In this assignment you will implement several models for finding magic sequences. Apart from the basic model, you will get to add implied constraints and also implement your own propagator.

A magic sequence of length n is a sequence of numbers $\langle x_0, x_1, \ldots, x_{n-1}\rangle$ such that if the variable x_i has the value k, then the sequence contains exactly k i's. For example, the sequence $\langle 1, 2, 1, 0\rangle$ is a magic sequence, since there is one occurrence of 0 (and x_0 has the value 1), there are two occurrences of 1, one occurrence of 2, and no occurrences of 3.

1. Your first task is to implement a basic model for finding magic sequences. The script should take an argument specifying the length of the sequence. Use one `IntVarArray` containing n variables with an appropriate initial domain. To implement the constraint connecting variables to occurrences, you should use reification. This can be done for each value k by defining Boolean variables b_i, where the variable b_i is true if and only if the variable x_i has the value k. Then the sum of the Boolean variables will be the number of occurrences of the value k.

2. The next task is to improve the script by two implied constraints.

   - A simple argument will show you what the sum of all the variables in the sequence should be. Use this fact to add an implied constraint to the model.
   - Another interesting implied constraint is defined by the following equation.

     $$\sum_{i=0}^{n-1}(i-1)\cdot x_i = 0$$

     This equation holds for every magic sequence (why?).

3. Your third task is to implement a specialized propagator for the constraint that you expressed by reification in the first task. The constraint is called `exactly`, and posting the constraint for an `IntVarArray x`, a value `y`, and an IntVar `z` means that exactly `z` of the variables in `x` should take the value `y`. Note that you should *not* use the (more general) `count` constraint that is provided by Gecode. You should have the same search-space for the reified version with both the implied constraints, as you get by using

your own propagator instead of the reified constraints (but still with the implied constraints).

The technical details on how to implement a propagator in Gecode are discussed in detail in Part P of "Modeling and Programming with Gecode" available from Gecode's webpage.