Constraint Programming: Assignment no. 6

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

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

This is the third of four graded assignments. The submission deadline is Monday, June 4th, 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`.

In this week's assignment, you will implement branchings and search engines in Gecode/J.

Exercise 6.1 (Branchings, 10P)

Branchings are very similar to propagators. Instead of a propagate method, however, they have three methods:

boolean status(Space home)
: returns whether this branching can actually split the search space. Typically, you can just check whether at least one of the variables you are branching on is not yet assigned.

JavaBranchingDesc description(Space home)
: returns a *branching description*. Most branchings *select* one variable and one value. E.g., a naive branching selects the first unassigned variable of an array, and selects its minimum, and then posts the constraints that the selected variable is equal to its minimum or not. A `JavaBranchingDesc` reflects this: it is a pair of a position (the index of the selected variable) and the selected value.

ExecStatus commit(Space home, JavaBranchingDesc d, long alt)
: uses the branching description **d** to post the constraints that correspond to alternative **alt**. E.g., for a naive branching, if **d.pos()** = 3, **d.val()** = 2, and **alt** = **0**, you could post the constraint **x[3]** = **2**, whereas **alt** = **1** would correspond to **x[3]** ≠ **2**. Another branching might post **x[3]** < **2** and **x[3]** ≥ **2**.

a) Use the template from the file `BranchingTemplate.java` to implement a first-fail, split-min branching, i.e. select the unassigned variable with the smallest domain size, and split its domain in two halves. If there are two or more variables with minimal domain size, pick the first one.

b) Implement a branching that orders two activities which have to be disjoint. An activity is a pair $(start, end)$ of variables for the start and the end time. Activities are disjoint if either $a.end < b.start$ or $b.end < a.start$. Note that you don't need a branching description here, as there is no variable or value to select. Just return a dummy `JavaBranchingDesc` and ignore it in `commit`. You can test your branching with the meeting problem from assignment 2.

Hint: You should also have a look at the file `QueensJavaPropagator.java` from the Gecode/J distribution.

Exercise 6.2 (Depth-first search, 12P)

In this exercise, you will implement an iterative depth-first search engine with recomputation.

The fundamental idea is to combine the stack (needed for exploration) with the path (needed for recomputation): Our stack items are of the form (d, a), where d is the branching description for a choice node, and a is the *next* alternative of that choice to be explored. See Figure 1 for an example, and note that the first alternative has number 0.

Note that this is different from the stack we presented in the lecture! We do not push one item per alternative, but only a single item representing the current alternative.

A stack item has two methods: **next()** increments which alternative to consider next, and **done()** checks whether there are still alternatives left.

Note that getting a space for a stack item's current alternative means recomputation, and for recomputation keep in mind that all stack items point to the *next* alternative to be explored, not the current one!

a) Complete the code in the file `NaiveDFS.java` such that it implements iterative depth-first search with full recomputation.
b) In the recursive formulation of DFS you saw in the lecture, the current space was always used for the left alternative, and only the right branch had to be recomputed. Modify your implementation to include this important optimization.
c) Modify your code such that it implements *fixed recomputation* in a class `DFS`. The idea is to use items of the form (d, a, c), where c is a copy of a space or `null`. Now recomputation does not start at the copy of the root space, but at the highest copy in the stack (i.e. the one that was added last). Make sure the root stack

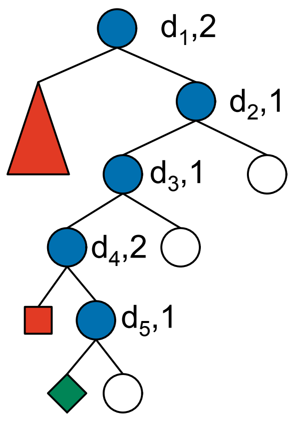

Figure 1: Symbols on the stack when exploration hits the solved node

item always contains a copy, and place a copy in every n-th stack item where n is configurable. Note that $n = 1$ amounts to full copying, whereas $n = \infty$ (i.e., large enough) means full recomputation.

You do not have to submit code for a), but please submit separate source code for b) and c).

Exercise 6.3 (Branch-and-bound search, 8P)

Modify the iterative DFS with fixed recomputation from the previous exercise to perform branch-and-bound search. The class should be called `BAB`. Each call to `next` should return the next solution (which is better than all previous solutions) or `null` (when no more solutions can be found).

Think about when you have to call the `constrain` method of a space. In a first step, you may call `constrain` on the same space with the same solution more than once (this is slightly easier to implement). In a second step, you should try to avoid this. You do not have to submit separate source code for the first step.