KTH Information and
Communication Technology

Constraint Programming (ID2204), Spring 2015
Christian Schulte

Assignment 4

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: A4` 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 gets the 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 **2015-06-09, 9:00**.

More information you can find on the course's homepage.

1 Golomb Rulers (EXP)

In the lecture you have seen the sketch of a model for Golomb rulers. This task lets you work out the details.

1. Give a script for the model introduced in the lecture. The distance variables are stored best in an `IntVarArray`.

   For a Golomb ruler with n ticks, there are $\frac{n^2-n}{2}$ distance variables. The distance variable d_{ij} should be stored at position

   $$\frac{i \times (2n-i-1)}{2} + j - i - 1$$

   in the `IntVarArray`.

2. Can you find any symmetry breaking constraints?

3. Try to find a good branching strategy.

4. Compare the effect of using the naive (`ICL_VAL` as last argument) or the domain-consistent (`ICL_DOM` as last argument) propagator for the distinct constraint.

5. Compare the effect of using the domain-consistent (`ICL_DOM`) or the bounds-consistent (`ICL_BND`) propagator for the distinct constraint. What do you observe?

6. Another observation is that d_{ij} must be at least the sum of the first $j-i$ integers. Explain why! Use this insight as a lower bound for the d_{ij}. **Note:** The sum of the first n integers is $n(n+1)/2$.

   This insight is taken from: Barbara Smith, Kostas Stergiou, Toby Walsh, *Modelling the Golomb Ruler Problem*. In IJCAI 99 Workshop on Non-binary Constraints, 1999. PDF

2 Branching for Square Packing (SUB, 2 points)

In this task we are going to reconsider the square packing problem from the previous assignment. The goal is to implement a custom branching that splits the coordinates of a square such that a certain part of the coordinates becomes obligatory (again, consult the paper mentioned in the previous assignment). Suppose a square of size 10×10 and that we are considering the obligatory part in x-direction. Assume that the x-coordinate can be between 0 and 100. Then, the square has no obligatory part. Now assume that the coordinate is between 0 and 4. Then, we know that the square can either occupy the x-coordinates 0–9, 1–10, 2–11, 3–12, or 4–13. With other words, no matter where the square is being placed, the coordinates 4–9 will always be occupied: hence, 4–9 is called the obligatory part.

1. Read Section 3.6, “Forcing Obligatory Parts” in the paper.

2. Read Section 9.2 “Hybrid recomputation” in “Modeling and Programming with Gecode”.

3. Read the first chapter in the Part “Programming branchers” (Chapter 31) in “Modeling and Programming with Gecode”.

4. Implement a branching, called `interval`, that takes an array of coordinate variables (either x or y coordinates), an array of dimensions (either width or height), and a percentage p (a `double` between 0.0 and 1.0) that does the following: provided that the domain of a coordinate variable is still large enough, it splits the coordinate interval into subintervals such

that the obligatory part in each subinterval is p percent of the rectangle's dimension.

You can decide yourself whether you want to split into several subintervals (that is, your brancher will create a choice with several alternatives), or whether you want to split into just two intervals, where only the interval corresponding to the first alternative forces the obligatory part to a certain percentage.

Note that you only have to split sufficiently large intervals, assigning the coordinates a value can be done by other branchings.

Use the file `interval.cpp` from the course webpage as a starting point.

Submit the file `interval.cpp`.

3 Linear Arithmetic (EXP)

Familiarize yourself with the algorithm for propagating linear equations as introduced in the lecture. An instructive example is propagating the linear equations

$$x - 2 \times y = 0 \qquad \text{and} \qquad x - y - y = 0$$

starting from the store $\{x \mapsto \{1, \ldots, 4\}, y \mapsto \{0, \ldots, 4\}\}$.

4 Understanding Régin's Algorithm (EXP)

Propagate the distinct constraint with Régin's algorithm on paper for an example.

5 Maximum Density Still Life (SUB, 3 points)

Conway's *Game of Life* is a zero-player game that is played on an infinite grid of cells that can be either occupied or empty (called *live* or *dead* respectively). Given an initial configuration, the rules specify how the grid evolves in time-steps. The state of each cell in the next generation is determined by its 8 neighbors in the current generation:

- A live cell with less than two live neighbors dies. (This is due to boredom)
- A live cell with more than three live neighbors dies. (This is due to overcrowding)
- A live cell with two or three live neighbors is alive in the next generation.
- A dead cell with precisely three live neighbors is alive in the next generation.

The rules are applied simultaneously to every cell in the pattern to get the next generation. As a simple example, consider the following sequence of patterns that shows part of a gliding pattern that will travel diagonally down the grid (squares not shown are considered to be empty):

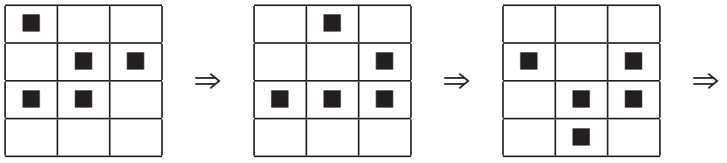

Table 1: Game of Life Glider

There are many interesting problems connected to the Game of Life. One is the problem of finding *Maximum Density Still Life patterns*. A still life pattern is a pattern of live cells that will not change from one generation to the next. The simplest such pattern is a two-by-two block of lives cells. In the maximum density still life problem of size n a pattern is sought that fits into a square of size $n \times n$, that is a still life pattern, and that has the maximum number of live cells. For example, a maximum density still life pattern of size 5 has 16 live cells.

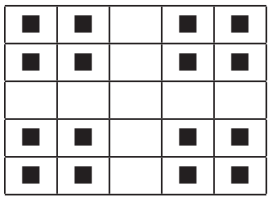

Table 2: Maximum density still life pattern of size 5

Your task is to implement a model that finds a maximum density still life pattern for a given value n. Proceed as follows

- Use a matrix of Boolean variables for the cells. A 1 indicates a live cell and a 0 indicates a dead cell.
- For each cell, add constraints that make sure that it stays the same in each generation.
- To ensure that the pattern does not spread (and for ease of implementation), add a border of size two around the pattern and set it to empty. That is, for a pattern of size 7, use a matrix of size 11-by-11. Add the constraints also for the inner border.
- Maximize the sum of values of the individual cells.

With this model, you should be able to solve the size 8 problem in about a minute on a modern computer.

To improve the model, we can use the following observation: for a 3-by-3 slice of a maximum-density pattern, the maximum number of lives cells is 6. Divide the pattern into such squares of size 3, and maximize the sum of the densities for the individual squares (where each square is known to have a density between 0 and 6). This gives a stronger bound on the optimization variable. With this model you should be able to solve the size 9 problem in about 5 minutes on a modern computer.
You should send your program and also maximum density still life patterns that you found of sizes 8 and 9 together with their densities.
For more information about the Maximum Density Still Life problem and constraint programming, see the following papers.

- Robert Bosch and Michael Trick, *Constraint Programming and Hybrid Formulations for Life*, Workshop on Modelling and Problem Formulation (Formul'01), CP'01, December 2001. PS
- Barbara Smith, *A Dual Graph Translation of a Problem in 'Life'*, Proceedings CP'02, 2002. PS
- Kenil C. K. Cheng and Roland H. C. Yap, *Applying Ad-hoc Global Constraints with the case Constraint to Still-Life*, Constraints, Volume 11 (2006), Pages: 91-114. PDF

Submit the file `life.cpp`.

6 Locating Warehouses (EXP)

A company needs to construct warehouses to supply stores with goods. Each warehouse possibly to be constructed has a certain capacity defining how many stores it can supply. Constructing a warehouse incurs a fixed cost. Costs for transportation from warehouses to stores depend on the locations of warehouses and stores.
Determine which warehouses should be constructed and which warehouse should supply which store such that overall cost (transportation cost plus construction cost) is smallest.
The cost of building a warehouse is 30. There are five warehouses $w_0, \ldots, w_4$ and ten stores $s_0, \ldots, s_9$. The warehouses have the following capacity:

w_0	w_1	w_2	w_3	w_4
1	4	2	1	3

The costs to supply a store by a warehouse are defined by a matrix c_{ij} ($0 \leq i \leq 9$, $0 \leq j \leq 4$) as follows:

	w_0	w_1	w_2	w_3	w_4
s_0	20	24	11	25	30
s_1	28	27	82	83	74
s_2	74	97	71	96	70
s_3	2	55	73	69	61
s_4	46	96	59	83	4
s_5	42	22	29	67	59
s_6	1	5	73	59	56
s_7	10	73	13	43	96
s_8	93	35	63	85	46
s_8	47	65	55	71	95

Implement a model for the problem as follows:

- Introduce for each warehouse a zero-one variable (called open) $Open_i$ which equals one, if the warehouse supplies at least one store.
- Introduce for each store a variable $Supplier_i$ taking values from $\{0,\ldots,4\}$ such that $Supplier_i = j$ if warehouse w_j supplies store s_i.
- Introduce for each store a variable $Cost_i$ which defines the cost for a store s_i to be supplied by warehouse $w_{Supplier_i}$ (note that you will have to use `element` here).
- For a given warehouse w_i the following must hold: the number of stores s_j supplied by w_i is not allowed to exceed the capacity of w_i. You can express this via reification or by using the propagator `count`.
- For a given warehouse w_i the following must hold: if the number of stores s_j supplied by w_i is at least one, then $Open_i$ equals one.
- The total cost is defined by the cost of open warehouses (that is, the sum of $Open_i$ for all warehouses multiplied with the cost for a warehouse) and the cost of stores (that is, the sum of the $Cost_i$ for all stores).
- The branching assigns values to the variables $Cost_i$ and follows the strategy of least regret: select the variable for which the difference between the smallest value and the next larger value is maximal.

Compute a solution with least cost.