Object-Oriented Constraint Programming in Java Using the Library `firstcs`

— An Introductory Tutorial —

Armin Wolf

Fraunhofer FIRST, Kekuléstr. 7, D-12489 Berlin, Germany `Armin.Wolf@first.fraunhofer.de`

Abstract. This tutorial shows how to use the object-oriented Java constraint programming library `firstcs` to solve constraint problems. Beyond the architecture of the system and the supported constraints, this presentation focuses on the implementation of new constraints, the modeling of problems and their solutions using constraint propagation and search. The presentation is completed a practical application realized with this constraint library.

1 Introduction

The object-oriented Java constraint programming library `firstcs` [1] differs from constraint logic programming (CLP) systems like CHIP, ECL^{i}PS^e or SICStus Prolog in some major topics:

- imperative versus rule-based programming,
- stateful typed variables and objects versus logic variables and terms,
- no pre-defined search versus built-in depth-first search.

Additionally, due to object-orientation in `firstcs` we have classes of objects and inheritance of concepts. Furthermore, in CLP systems choice-points and their maintenance is opaque, only their abdication has to be stated explicitly using the "cut". In `firstcs` the situation is reversed: choice-points has to be established an maintained explicitly. There are methods for setting, resetting, backtracking and committing, where the last has similar functionality like the "cut" in CLP.

Another difference is the explicit activation of constraint propagation. In CLP each time a constraint is established it is propagated, too. However, in `firstcs` we have the possibility to gather the constraints and propagate them "en bloc". This may result in reduced runtime, because the whole information about the modeled CSP is available during the first propagation.

The tutorial is organized as follows: First, we give an overview of `firstcs`' architecture, the available constraints as well as search and optimization methods. Then we show its extension by a new constraint. Further, we show how to model a constraint problem using the Golomb ruler problem as an example. Finally, we show the usage of choice-points within the implementation of a simple depth-first search.

2 An Overview on `firstcs`

The kernel of our Java constraint programming library `firstcs` is formed by a Java class called `CS` which is an acronym for the term *Constraint System*. Each object of this class is indeed a constraint system managing finite domain variables and constraints over these variables. Due to the object-oriented design, it is possible to generate and manipulate several constraint systems in a single application. In the current version these systems are independent.

There are the Java classes `Domain`, `Variable`, `Constraint` and the subclasses of `Constraint` around the kernel providing the tool box to model and solve constraint problems.

The class `Domain` implements the finite domains (fd) of the variables, i.e. finite integer sets represented by lists of integer intervals. There are several methods to manipulate these sets, i.e. the usual set operations. All these methods return a boolean value which is false if and only if the

manipulated set becomes the empty set. This information is used to detect inconsistencies during constraint propagation.

The class `Variable` implements the fd-variables representing the unknowns of a constraint problem. Their admissible values are restricted by their finite domains and their constraints. Thus, they are implemented as *attributed variables* [2]: Together with the domains, the constraints are attached to their variables. This construct is commonly used in constraint logic programming systems, e.g. like SICStus Prolog[1], too.

2.1 Built-In Constraints

The abstract class `Constraint` samples all the concrete constraint classes. Beyond other application-specialized constraints, these are:

- `AllDifferent` constrains an array of n variables to have pairwise different values, i.e $x_i \neq x_j$ for $1 \leq i < j \leq n$.
- `Before` constrains an activity to be finished before another one starts, i.e. $a.start + a.duration \leq b.start$.
- `Cumulative` constrains some activities to be scheduled on a common resource such that the sum of their capacity requirements never exceeds the capacity of the resource.
- `Disjoint2` constrains some rectangles to be placed non-overlapping in a 2-dimensional area.
- `Equal`, `Greater`, `GreaterEqual`, `Less`, `LessEqual`, and `NotEqual` relate two variables with respect to the corresponding arithmetic relation, i.e. $s < t$.
- `Kronecker` states that a variable has a given integer value if another boolean value is true (1) or false (0), i.e. $\delta_{v,i} = 1$ if $v = i$ and 0 otherwise.
- `SingleResource` and `AlternativeResource` state that some activities are processed either on an exclusively available resource or on alternatively available resources.
- `SetUpTime` and `SetUpCost` state sequence-dependent setup times or cost for activities that are processed successively on some resources.
- `Product`, `Sum` and `WeightedSum` establish arithmetic relations between variables, especially cost functions for optimization problems.

These constraint classes are extensions of the class `Constraint`; they inherit the basic concepts of a constraint implementation in `firstcs`.

2.2 Built-In Search and Optimization

For users that are not familiar with the implementation of search strategies, especially with the supported concept of choice-points, `firstcs` offers some pre-defined search strategies, i.e. subclasses of the class `AbstractLabel`. All these classes offer a method `nextSolution()` that allows an iteration over all solutions of a given constraint problem. Additionally, they inherit the methods `nextMinimalSolution(Variable objective)` and `nextMaximalSolution(Variable objective)`, allowing an iteration over *all* minimal and maximal solutions with respect to a given objective function. Therefore, the variable `objective` has to be constrained to the function's result. Beyond others `firstcs` comes with the following subclasses of `AbstractLabel`:

- `BtLabel` implements the `labeling` known from CLP systems and supports some well-known heuristics like first-fail or random variable order.
- `ResourceLabel` implements a specialized search for task scheduling problems. Before any labeling of the tasks' start times the search looks for a *linear order* of the tasks on the considered resource.
- `OrderLabel` implements the *Reduce-To-The-Max* search algorithm presented in [3,7] for contiguous task scheduling and optimization problems.

[1] See `http://www.sics.se/sicstus.html`.

The usage of any of these classes is quite simple (cf. Listing 1.1): Let a constraint system `cs` with constraints be given (cf. line 1) that restrict the values of the variables, given in an array `vars` (cf. line 2). Finding all solutions of this CSP by the use of the system-integrated labeling requires the creation of a `BtLabel` object with these arguments (cf. line 3). Then we define a search strategy: a combination of a random variable permutation (cf. line 4) and the first-fail principle (cf. line 5). Before performing an iteration in the loop over `nextSolution()` (cf. lines 7–11) that finds and prints all solutions, we have to initialize the search process, i.e. to set a choice-point (cf. line 6). Finally, we use this choice-point to reset the constraint system to the state that was valid before the search started (cf. line 12). It should be noted that the solutions are defined by the unary-valued domains of the variables (printed together with the variables in line 9).

Listing 1.1. Finding All Solutions of a CSP

```
CS cs = ...                                                          1
Variable[] vars = ...                                                2
BtLabel label = new BtLabel(cs, vars);                               3
label.useRandomVariables();                                          4
label.useFirstFail();                                                5
label.set();                                                         6
while (label.nextSolution()) {                                       7
    for (int i=0; i <= vars.length; i++) {                           8
        System.out.println(vars[i]);                                 9
    }                                                                10
}                                                                    11
label.reset();                                                       12
```

For optimization purposes, i.e. finding minimal or maximal solutions of a CSP with respect to an objective function we have to constrain a variable, say `objective`, to that function. Then we only have to replace `nextSolution()` in Listing 1.1 (cf. line 7) either by `nextMinimalSolution(objective)` or `nextMaximalSolution(objective)`. This also holds not only for `BtLabel` but also for all other search algorithm classes extending `AbstractLabel`. Further, there are generic implementations of some *branch-and-bound* principles in `AbstractLabel` using the specific implementations of `nextSolution()` in its subclasses. The implemented optimization algorithms

- are *incremental*, if the following two methods are implemented (cf. [6]):
  - `storeLastSolution()` for storing a solution and
  - `restoreLastSolution()` for partially restoring a solution,
- perform either *monotonic* or even *dichotomic* bounding,
- compute potentially *all* optimal solutions,
- are symmetric because a minimal solution with respect to an objective function f is maximal with respect to $-f$.

Incremental search uses the fact, that branches that were considered to find a *first* (improved) solution never contains an even better solution — otherwise we would have found it. Without loss of any better solution we can continue the search on the branch where we found the last solution.

The differences between monotonic and dichotomic bounding are shown for minimization in Figure 1 and Figure 2 and will be explained in more detail:

Monotonic bounding for minimization improves the upper bound upb of the objective function f monotonically: If there is a solution having an objective value obj then this is a new upper bound $upb' := obj$. It is tried to find an even better solution by adding the constraint $f < upb'$ that bounds the objective function f and cuts some branches in the search tree. This procedure is repeated until no (better) solution is found. Then, it follows that the last upper bound, say upb'' is also a lower bound of the minimal objective value, i.e. $lwb := upb''$. The most recently found solution (if any) and any other solution satisfying the additional constraint $f = lwb$ are then optimal solutions.

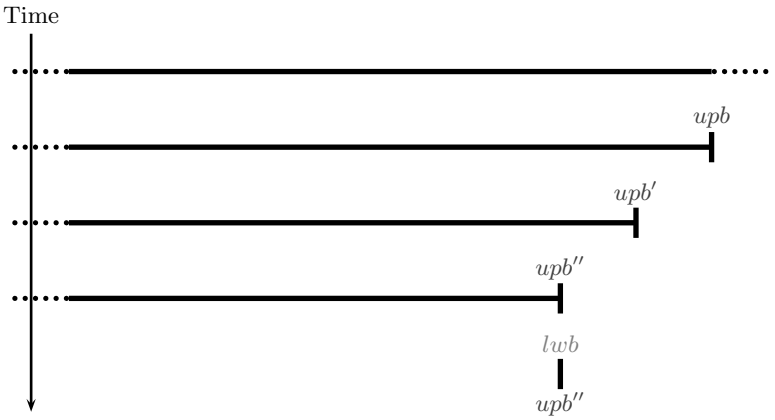

Fig. 1. Monotonic bounding.

Dichotomic bounding for minimization requires a (trivial) lower bound lwb and a (trivial) upper bound upb of the objective function f. The improvement of the bounds performs dichotomic. It is tried to find a solution having an objective value within the interval

$$[lwb, \lfloor \frac{lwb + upb}{2} \rfloor]$$

by adding the constraints

$$lwb \leq f \wedge f \leq \lfloor \frac{lwb + upb}{2} \rfloor \ .$$

If this trail fails the lower bound is updated to be

$$lwb' := \lfloor \frac{lwb + upb}{2} \rfloor + 1 \ .$$

Otherwise, if there is a solution, the upper bound is updated to be the objective value obj of this solution: $upb := obj$ and the search continues recursively until the lower bound is greater than the upper bound, say $lwb''' > upb''''$, holds. The most recently found solution (if any) and any other solution satisfying the additional constraint $f = upb''''$ are then optimal solutions.

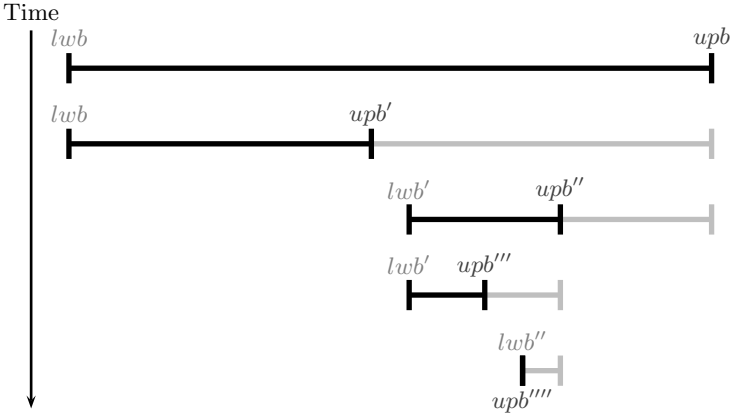

Fig. 2. Dichotomic bounding.

For example, dichotomic bounding for `BtLabel` is applied in Listing 1.1 if we add the method call `label.setBoundingScheme(DICHOTOMIC)` between line 3 and line 6.

3 Constraint Implementation

Any constraint in `firstcs` must be implemented as a subclass of the class `Constraint`. This common superclass defines some basic data-structures of any constraints and guides the implementation of any constraint-specific pruning algorithm.

The core of any constraint system is an iterative process that realizes constraint propagation. It schedules and activates the pruning algorithms of those constraints whose variable domains are changing. Thus it is important to know the variables that are constrained by an object of the class `Constraint` and how to react on any change of their domains. Therefore the class `Constraint` maintains a vector of each constraint's variables `linkedVars` (cf. Listing 1.2, line 6) and a method to add (or link) a variable to each constraint via this vector (cf. Listing 1.2, lines 11–13). These linked variables are used when a generated constraint is added to a constraint system `cs`, i.e. an instance of the class `CS` (cf. line 4). Then within this constraint system the constraint is registered under its variables and on how to react on any changes of their domains calling the method `register()` (cf. Listing 1.2, lines 19–24). By default, any change will trigger an activation of the registered constraints (cf. Listing 1.2, line 22). As will see later, it is possible to restrict the activation on specific changes, e.g. changes either of the lower or the upper bound of the domain.

Listing 1.2. Variable management of any constraint

```
...                                                                    1
public abstract class Constraint {                                     2
    ...                                                                3
    protected CS cs = null;                                            4
                                                                       5
    protected Vector linkedVars = new Vector();                        6
                                                                       7
    /**                                                                8
     * adds the given variable to this constraint's variables.         9
     */                                                               10
    public void addVar(Variable var) {                                11
        linkedVars.add(var);                                          12
    }                                                                 13
                                                                      14
    /**                                                               15
     * registers this constraint to its variables.                    16
     * - may be specialized                                           17
     */                                                               18
    public void register() {                                          19
        Iterator i = linkedVars.iterator();                           20
        while (i.hasNext()) {                                         21
            cs.link((Variable) i.next(), this);                       22
        }                                                             23
    }                                                                 24
    ...                                                               25
```

Concrete constraint classes must have constraint-specific implementations of the abstract method `activate()` defined in the class `Constraint` (cf. Listing 1.3). Its specific implementation has to perform local constraint propagation, i.e. the domains of the constraints' variables are pruned until a local fix-point is reached. Whenever any variable's domain becomes empty, an inconsistency is detected and an `InconsistencyException` has to be thrown. We use exceptions instead of returning a value, e.g. false, over several method calls to force an explicit and efficient handling of inconsistencies at any appropriate place in the program code. Especially, it is impossible to ignore inconsistencies avoiding senseless deductions, i.e. when *ex falso quod libet* holds.

Listing 1.3. Activation schema of any constraint's pruning algorithms.

```
    ...
    /**
     * activates this constraint to establish consistency
     * - must be implemented
     */
    public abstract void activate()
        throws InconsistencyException;
    ...
}
```

The implementation of binary `Less` constraints that constrains one variable to be smaller than another variable, say $LHS < RHS$, is an extension of the class `Constraint` (cf. Listing 1.4, line 5). Each `Less` constraint consists of two variables representing the left- and right-hand-side of the binary relation "<" (cf. Listing 1.4, line 8 and 11). The creation of an instance of a `Less` constraint is straightforward: An object for two variables to be constrained is generated, the variables are stored locally and are added to the recently generated instance (cf. Listing 1.4, lines 16–24).

Listing 1.4. The basics of the binary `Less` constraint.

```
package de.fhg.first.cs.constraint;

import ...
...
public class Less extends Constraint {

    // the left-hand-side variable, i.e. LHS:
    protected Variable lhs = null;

    // the right-hand-side variable, i.e. RHS:
    protected Variable rhs = null;

    /**
     * creates a new LHS < RHS constraint.
     */
    public Less(final Variable left, final Variable right) {
        // initialize the constraint's variables:
        lhs = left;
        rhs = right;
        // connect the constraint's variables
        // with the constraint itself:
        addVar(lhs);
        addVar(rhs);
    }
```

We know that bounds consistency for the binary constraint $LHS < RHS$ is equivalent to its local consistency. Thus, we only have to consider and update the bounds of the variables' domains to establish these consistencies. This pruning will be performed by the implementation of the pre-defined method `activate()` (cf. Listing 1.5). Therefore we have to perform two operations:

- the values of the right-hand-side variable must be greater than the smallest value of the left-hand-side variable and
- the values of the left-hand-side variable must be less than the greatest value of the right-hand-side variable.

The first adjustment is performed by calling a `Variable`'s method `greater()` that uses the method `min()` which returns the smallest value of its domain (cf. Listing 1.5, line 8). The method `greater()` prunes all values in its variable's domain not greater than the given value. It returns true if the domain is changed and throws an `InconsistencyException` if the reduced domain

becomes empty. The second adjustment is symmetric. There, the method `less()` prunes all values in its variable's domain not less than given value which is the greatest value of the right-hand-side variable (cf. Listing 1.5, line 11). The method call returns true if the domain is changed and throws an `InconsistencyException` if the reduced domain becomes empty.

Listing 1.5. The pruning performed by `Less` constraints.

```
/**                                                                    1
 * activate this constraint w.r.t the registration                     2
 * and prune its variables' domains.                                   3
 */                                                                    4
public void activate() throws InconsistencyException {                 5
    // adjust the values of the rhs to be greater                      6
    // than the minimum value of the lhs:                              7
    rhs.greater(lhs.min());                                            8
    // adjust the values of the lhs to be less                         9
    // than the maximum value of the rhs:                              10
    lhs.less(rhs.max());                                               11
}                                                                      12
```

It should be noted that any further adjustment of the right-hand-side's domain is only necessary if the smallest value of the left-hand-side's domain increases and vice-versa: the left-hand-side's domain must be only updated if the greatest value of the right-hand-side's domain decreases. We are able to reflect this situation in a specific implementation of `register()` shown in Listing 1.6. There, it is stated in line 8 and line 11 that pruning must only be re-activated if the minimum of the left-hand-side's domain and or the maximum of the right-hand-side's domain changes.

Listing 1.6. The specific registration of `Less` constraints.

```
/**                                                                    1
 * link this constraint via its variables                              2
 * to the constraint solver/system.                                    3
 */                                                                    4
public final void register() {                                         5
    // activate, if the minimum of the                                 6
    // left variable's domain changes:                                 7
    cs.link(lhs, this, AbstractPropagator.ACT_ON_MIN);                 8
    // activate, if the maximum of the                                 9
    // right variable's domain changes:                                10
    cs.link(rhs, this, AbstractPropagator.ACT_ON_MAX);                 11
}                                                                      12
```

Further, we observe that both adjustments of the left- and the right-hand-side's domain (cf. Listing 1.5, line 8 and 11) are independent from each other such that any further call of them will not change the adjusted domains, i.e. a local fix-point is reached. In general, if this is not valid we have to iterate over the pruning operations (e.g. `lessEqual()`, cf. line 5) until all variables' domains stay unchanged. Listing 1.7 shows our general approach to calculate a local fix-point and to force idempotency of the method `activate()`: We iterate over the pruning operations as long as at least one variable's domain has changed. This is signaled by a boolean variable `hasChanged`.

Listing 1.7. Forcing idempotency of the method `activate()`.

```
public void activate() throws InconsistencyException {                 1
    boolean hasChanged = true;                                         2
    while (hasChanged) {                                               3
        hasChanged = false;                                            4
        hasChanged |= var.lessEqual(...);                              5
        hasChanged |= ...;                                             6
    }                                                                  7
}                                                                      8
```

4 Customizing Constraint Processing

In contrast to other constraint programming systems, especially to constraint logic programming systems, the addition of a constraint does not automatically perform its activation, i.e. the propagation of its consequences. In `firstcs` any possible pruning of the variables' domains with respect to this constraint is delayed by default. It must be performed either via a general constraint propagation or explicitly by the call of the constraint's method `activate()`. Thus, it is possible to model a constraint problem to be solved completely before propagating the constraints' consequences. This may improve the overall performance of the constraint processing as the following example shows:

Example 1. Considering the constraint problem $var_{i-1} < var_i$ for $i = 1, \ldots, n-1$ any incremental constraint propagation is $\mathcal{O}(n^2)$: Any extension of $var_0 < \cdots < var_j$ to $var_0 < \cdots < var_j < var_{j+1}$ will adopt the domains of all considered variables. However, a delayed propagation triggered after adding all constraints might be $\mathcal{O}(n)$. The Java program in Listing 1.8 realizes both: incremental propagations immediately after adding constraint by constraint and a delayed propagation after adding all constraints – the boolean variable `IS_INCREMENTAL` triggers either case.

Listing 1.8. Incremental and delayed constraint propagation.

```
boolean IS_INCREMENTAL = ...                                    1
...                                                             2
CS cs = new CS();                                               3
Variable[] var = new Variable[n];                               4
for (int i=0; i<n; i++) {                                       5
    var[i] = new Variable(i, n);                                6
    if (i > 0) {                                                7
        cs.add(new Less(var[i-1], var[i]));                     8
        if (IS_INCREMENTAL) cs.activate();                      9
    }                                                           10
}                                                               11
if (!IS_INCREMENTAL) cs.activate();                             12
```

In line 12 `cs.activate()` activates all constraints that were added to the constraint system `cs` but not yet activated, i.e. delayed. In the incremental case only the most recently added constraint is activated (cf. line 9) which will activate those constraints whose variables' domain were changed until a new fix-point is calculated. Runtime experiments have shown that the delayed non-incremental propagation is in fact linear if the constraints are activated in a last-in-first-out order. □

5 Modeling and Solving a Constraint Problem

The *Golomb ruler problem* is a good example to show the modeling purposes of `firstcs`. Global constraints are combined with simple constraints and the basic model is extendible by symmetry-breaking and implicit constraints. Furthermore, the problem is known to be a hard *constraint optimization problem* (COP).

Golomb rulers are named after the mathematician *Solomon W. Golomb*. They are important in physics, radio-astronomy, crystallography where we are interested in rulers of *minimal* length. The difference between Golomb rulers and ordinary rulers is significant: all distances between its marks are different. Figure 3 shows a Golomb ruler with 5 marks and of minimal length 11.

The in the following presented modeling of the Golomb ruler (optimization) problem in `firstcs` is according to the considerations in [4,5]. The main program of our Golomb ruler problem solver is shown in Listing 1.9. In the header the constraint system (cf. line 2), the number of marks n and a heuristic upper bound for the length of minimal rulers nn (cf. line 3) as well as the marks and their differences (cf. line 4) are declared.

Fig. 3. A Golomb ruler with 5 marks and of minimal length 11.

The `main` method parses the number of marks from the command line input (cf. lines 6–8), creates a new corresponding Golomb ruler instance (cf. line 9), establishes the base constraint model (cf. line 10) as well as the extended model (cf. line 11) and finally finds all minimal solutions (cf. line 12).

Listing 1.9. The main program of the Golomb ruler problem solver.

```
public final class GolombRuler {                                          1
    CS cs;                                                                2
    int n, nn;                                                            3
    Variable[] marks, diffs;                                              4
    ...                                                                   5
    public static void main(final String[] args) {                        6
        // the Golomb ruler's number of marks:                            7
        final int num = Integer.parseInt(args[0]);                        8
        GolombRuler golombruler = new GolombRuler(num);                   9
        golombruler.establishBaseModel();                                 10
        golombruler.establishExtendedModel();                             11
        golombruler.findAllMinimalSolutions();                            12
    }                                                                     13
}                                                                         14
```

The creator method for Golomb rulers having `num` marks is shown in Listing 1.10. It creates a new constraint system for this ruler (cf. line 2) and a heuristic upper bound for its length which is the square of its number of marks (cf. line 4). Additionally, variables for the marks and the quadratic number of differences between them are created, too (cf. lines 5–6). They will be constrained by the corresponding models.

Listing 1.10. The class of Golomb rulers.

```
public GolombRuler(int num) {                                             1
    cs = new CS();                                                        2
    n = num;                                                              3
    nn = n * n; // a heuristic upper bound for its length                 4
    marks = new Variable[n];                                              5
    diffs = new Variable[(n * (n - 1)) / 2];                              6
}                                                                         7
```

The base constraint model is shown in Listing 1.11. It states that the first mark is determined to be 0 (cf. line 3) and that all other marks have values between 1 and the heuristic upper bound `nn` (cf. line 5). Furthermore, it is stated that the $i-1$th mark is less than the ith mark (cf. line 6) and that for $0 \leq j < i \leq n$ each variable `diff[ i*(i-1)/2 + j]` is equal to the difference between the jth and the ith mark (cf. lines 8–11). Finally, all differences are constrained to be pairwise different using one global `AllDifferent` constraint (cf. line 15).

Listing 1.11. The basic model of the Golomb ruler problem.

```
public void establishBaseModel() {                                        1
    int idiff = 0;                                                        2
    marks[0] = new Variable(0);                                           3
    for (int i = 1; i < n; i++) {                                         4
        marks[i] = new Variable(1, nn);                                   5
        cs.add(new Less(marks[i - 1], marks[i]));                         6
```

```
            for (int j = 0; j < i; j++) {                                    7
                // here it holds: idiff == i*(i-1)/2 + j                     8
                diffs[idiff] = new Variable(1, nn);                          9
                cs.add(new Sum(marks[j], diffs[idiff],                       10
                       marks[i]));                                           11
                idiff++;                                                     12
            }                                                                13
        }                                                                    14
        cs.add(new AllDifferent(diffs));                                     15
    }                                                                        16
```

The additional constraints in the extended constraint model is shown in Listing 1.12. The first constraint forces that the differences between the first and the second mark is less than the one between the last and the next to last avoiding symmetric solutions: pairs of rulers that results from each other by turning them 180 degrees (cf. line 2). Additional constraints state how the differences between the jth and ith marks are bound by the constant value $(n-1-i+j)\cdot(n-i+j)/2$ and the last mark (cf. lines 7–9 and [4]).

Listing 1.12. The extended model of the Golomb ruler problem.

```
public void establishExtendedModel() {                                       1
    cs.add(new Less(diffs[0], diffs[diffs.length - 1]));                     2
    int idiff = 0;                                                           3
    for (int i = 1; i < n; i++) {                                            4
        for (int j = 0; j < i; j++) {                                        5
            // here it holds: idiff == i*(i-1)/2 + j                         6
            cs.add(new Before(diffs[idiff],                                  7
                              (n-1-i+j) * (n-i+j)/2,                         8
                              marks[n - 1]));                                9
            idiff++;                                                         10
        }                                                                    11
    }                                                                        12
}                                                                            13
```

Last but not least all solutions of an optimal Golomb ruler are found by the method presented in Listing 1.13. It uses "standard" labeling with the dichotomic bounding scheme (cf. lines 2–3) shown in Figure 2. The iteration over all optimal solutions has the same structure as the general search presented in Listing 1.1. The only difference is that we call the method `nextMinimalSolution()` with the last mark as its argument because it defines the objective to be minimized (cf. line 5).

Listing 1.13. Finding all Golomb rulers of minimal length.

```
public void findAllMinimalSolutions() {                                      1
    BtLabel labeler = new BtLabel(cs, marks);                                2
    labeler.setBoundingScheme(BtLabel.DICHOTOMIC);                           3
    labeler.set();                                                           4
    while (labeler.nextMinimalSolution(marks[n - 1])) {                      5
        System.out.print("optimal␣marks:");                                  6
        for (int i=0; i < n; i++) {                                          7
            System.out.print("␣␣" + marks[i]);                              8
        }                                                                    9
        System.out.println();                                                10
    }                                                                        11
    labeler.reset();                                                         12
}                                                                            13
```

It should be noted that the runtime of our Java implementation is comparable to an equivalent SICStus Prolog implementation. However, SICStus Prolog offers only system predicates that finds one optimal solution.

6 Realizing Search

After the addition of all constraints defining a problem to be solved and their propagation a search process is used to find a solution of the considered problem. The constraint library `firstcs` supports search based on different pre-defined strategies but also the necessary basics for any user-defined search based on backtracking. These basics are choice-points, i.e. objects of the class `ChoicePoint`. For a given constraint system, we are able to generate several choice-points to store the system's state at a specific program state. Therefore, a choice-point is set with its method `set()`, signaling the system that the current state of the constraint system has to be stored for any future backtracking. The call of the method `backtrack()` for a previously set choice-point will restore the stored state at the program state where the choice-point was set. For any re-use of a choice-point at another program state it might be reset with its method `reset()`.

For simplicity, the usage of choice-points is illustrated for simple depth-first search that finds *one* solution. The Java code of the recursively implemented `labeling()` method is presented in Listing 1.14. The input of this `static` method which does not belong to any object consists of

- a constraint system `cs` defining a CSP,
- an array of fd variables `vars` which have to be labeled.

The output of the method is

- true, if a solution was found,
- false, otherwise.

In case of a solution its values are determined by the single-valued variables' domains. Otherwise the CSP, especially its variables' domains are unchanged. The method uses the globally defined integer variable `level` which is initialized with the value 0 (cf. line 1). Its value reflects the considered level within the search tree. Its depth is equal to the number of variables. If all variables are labeled its value is equal the search tree's depth. In this case, the recursive `labeling()` stops and returns true signaling that the variables are labeled with the values in their domains (cf. lines 3–5). Otherwise, a new choice-point is created and set (cf. lines 6–7) preparing the labeling of the variable at the current level in the search tree. Therefore it is iterated over all values in the domain of this variable[2] (cf. lines 8–25). During this iteration the actual variable is labeled with the current value (cf. line 11) and this additional constraint is explicitly propagated (cf. line 12). If this results in an inconsistency it will be caught. This catch causes backtracking, i.e. the labeling of the actual variable is annihilated and the next value – if any – is tried (cf. lines 21–24). If there is no inconsistency the level is increased and the labeling procedure is called recursively. If it succeeds the choice-point is reset and true is returned (cf. lines 14–16). If it fails, backtracking annihilates the labeling of the actual value and the level is decreased to its correct values (cf. lines 17–20).

Listing 1.14. A recursive depth-first search to find a solution of a CSP.

```
private static int level = 0;                                          1
public static boolean labeling(CS cs, Variable[] vars) {               2
    if (level == vars.length) {                                        3
        return true;                                                   4
    } // else:                                                         5
    ChoicePoint cp = new ChoicePoint(cs);                              6
    cp.set();                                                          7
    for (int val=vars[level].min();                                    8
         val<=vars[level].max(); val++) {                              9
        try {                                                          10
            vars[level].equal(val);                                    11
            cs.activate();                                             12
            level++;                                                   13
```

[2] and possibly over some more value if the domain is not an integer interval.

```
            if (labeling(cs, vars)) {                                    14
                cp.reset();                                              15
                return true;                                             16
            } else {                                                     17
                cp.backtrack();                                          18
            }                                                            19
            level--;                                                     20
        } catch (InconsistencyException e) {                             21
            cp.backtrack();                                              22
        }                                                                23
    }                                                                    24
    cp.reset();                                                          25
    return false;                                                        26
}                                                                        27
```

It should be noted that this labeling is rather rudimental: If a solution of the CSP is found, then it is impossible to find further solutions because it is impossible to access the locally defined and used choice-points. Even another call of `labeling()` returns with an unchanged solution.

7 Conclusion

As we have seen, the constraint library `firstcs` offers as an open toolbox a variety of functionalities for constraint programming. It supports the realization of new constraints, the modeling of CSPs or even COPs as well as their solution either by the use of built-in or user-implemented search or branch-and-bound algorithms.

References

1. Matthias Hoche, Henry Müller, Hans Schlenker, and Armin Wolf. firstcs - A Pure Java Constraint Programming Engine. In Michael Hanus, Petra Hofstedt, and Armin Wolf, editors, *2nd International Workshop on Multiparadigm Constraint Programming Languages – MultiCPL'03*, 29th September 2003. Online available at `uebb.cs.tu-berlin.de/MultiCPL03/Proceedings.MultiCPL03.RCoRP03.pdf`.
2. Christian Holzbaur. *Specification of Constraint Based Inference Mechanism through Extended Unification.* PhD thesis, Dept. of Medical Cybernetics & AI, University of Vienna, 1990.
3. Hans Schlenker. Reduce-To-The-Max: Ein schneller Algorithmus für Multi-Ressourcen-Probleme. In Francois Bry, Ulrich Geske, and Dietmar Seipel, editors, *14. Workshop Logische Programmierung*, number 90 in GMD Report, pages 55–64, 26th–28th January 2000.
4. Barbara M. Smith, Kostas Stergiou, and Toby Walsh. Modelling the golomb ruler problem. In *Sixteenth International Joint Conference on Artificial Intelligence (IJCAI 99), Workshop on Non Binary Constraints*, Stockholm, August 2 1999.
5. Barbara M. Smith, Kostas Stergiou, and Toby Walsh. Using auxiliary variables and implied constraints to model non-binary problems. In *Proceedings of the National Conference of the American Associtation for Artifical Intelligence (AAAI-00)*, pages 182–187, Austin, Texas, 2000.
6. Pascal van Hentenryck and Thierry le Provost. Incremental search in constraint logic programming. *New Generation Computing*, 9(3 & 4):257–275, 1991.
7. Armin Wolf. Reduce-to-the-opt — a specialized search algorithm for contiguous task scheduling. In K.R. Apt, F. Fages, F. Rossi, P. Szeredi, and J. Váncza, editors, *Recent Advances in Constraints*, number 3010 in Lecture Notes in Artificial Intelligence, pages 223–232. Springer Verlag, 2004.