Abstract
This document describes how to write Java programs in Cream (Constraint Resolution Enhancement And Modules) class library. In other words, how to make your Java creamy.

Contents
1 Features 1
2 Installation 1
3 Programming 1
  3.1 First Step ................................. 1
  3.2 Using Coroutining Facility ................. 4
  3.3 Using SolutionHandler ..................... 5
  3.4 Searching with Timeout ..................... 5
  3.5 Finding Optimal Solution ................... 6
  3.6 Using Local Search ......................... 6
  3.7 Using Multiple Local Search Solvers ....... 7
4 API 7
5 License 7
6 Acknowledgement 8
1 Features

Cream is a class library helping Java programmers to develop intelligent programs requiring constraint satisfaction or optimization on finite domains. The followings are features of Cream.

- **100% Pure Java**: Whole programs are written in Java.
- **Open source**: Cream is distributed as a free software with source code. Please refer to License section for more details about licensing issue.
- **Natural description of constraints**: Various constraints can be naturally described within Java syntax.
- **Easy enhancements**: Programmers can easily enhance/extend constraint descriptions and satisfaction algorithms.
- **Various optimization algorithms**: Various optimization algorithms are available, such as Simulated Annealing, Taboo Search, etc.

2 Installation

There is nothing special for installation. Please unzip the zip file you can obtained from the web page, then you are ready to run example programs in examples directory as follows.

- on Unix-like systems:
  ```
  java -classpath ./:../cream.jar FourColor
  ```

- on Windows systems:
  ```
  java -classpath .;./cream.jar FourColor
  ```

Please note that you need Java 2, Standard Edition to use Cream.

3 Programming

3.1 First Step

This section describes how to use Cream step-by-step.

Consider an old Japanese elementary school problem:
There are some cranes and tortoises. They are 7 in total, and their legs are 20 in total. How many cranes and tortoises are there?

To solve this problem in Cream, firstly you need to create a constraint network (an instance of `Network` class) consisting of variables and constraints over those variables.

```java
Network net = new Network();
```

Secondly, please declare and create variables for numbers of cranes (that is, `x`) and tortoises (that is, `y`).

```java
IntVariable x = new IntVariable(net);
IntVariable y = new IntVariable(net);
```

These variables are added to the constraint network by the constructor.

Thirdly, please describe constraint conditions over those variables, that is `x` ≥ 0, `y` ≥ 0, `x + y` = 7, and `2x + 4y` = 20. First two conditions can be written in Cream as follows.

```java
x.ge(0);
y.ge(0);
```

These constraints are also added to the same constraint network which the variables belong.

It is possible to write them (but lengthy) as follows.

```java
new IntComparison(net, IntComparison.GE, x, 0);
new IntComparison(net, IntComparison.GE, y, 0);
```

The latter two conditions `x + y` = 7 and `2x + 4y` = 20 can be written simply as follows.

```java
x.add(y).equals(7);
x.multiply(2).add(y.multiply(4)).equals(20);
```

It is possible to rewrite them as follows.

```java
// 7 == x + y
new IntArith(net, IntArith.ADD, 7, x, y);
// t1 == x * 2
IntVariable t1 = new IntVariable(net);
new IntArith(net, IntArith.MULTIPLY, t1, x, 2);
// t2 == y * 4
IntVariable t2 = new IntVariable(net);
new IntArith(net, IntArith.MULTIPLY, t2, y, 4);
// 20 == t1 + t2
new IntArith(net, IntArith.ADD, 20, t1, t2);
```
Now, pass the constraint network to DefaultSolver to solve the problem by constraint propagation and backtracking.

Solver solver = new DefaultSolver(net);

You can get a solution from the solver as follows.

Solution solution = solver.findFirst();

This code only finds the first solution, but it is sufficient in this case. To get values of the variables in the solution, getIntValue methods can be used.

int xv = solution.getIntValue(x);
int yv = solution.getIntValue(y);

The following is the whole program.

```java
import jp.ac.kobe_u.cs.cream.*;

public class FirstStep {
    public static void main(String args[]) {
        // Create a constraint network
        Network net = new Network();
        // Declare variables
        IntVariable x = new IntVariable(net);
        IntVariable y = new IntVariable(net);
        // x >= 0
        x.ge(0);
        // y >= 0
        y.ge(0);
        // x + y == 7
        x.add(y).equals(7);
        // 2x + 4y == 20
        x.multiply(2).add(y.multiply(4)).equals(20);
        // Solve the problem
        Solver solver = new DefaultSolver(net);
        Solution solution = solver.findFirst();
        int xv = solution.getIntValue(x);
        int yv = solution.getIntValue(y);
        System.out.println("x = " + xv + ", y = " + yv);
    }
}
```
The same program is in examples directory. You can compile and execute it as follows.

- on Unix-like systems:

```java
javac -classpath .:../cream.jar FirstStep.java
java -classpath .:../cream.jar FirstStep
```

- on Windows systems:

```java
javac -classpath .;..

cream.jar FirstStep.java
java -classpath .;..

cream.jar FirstStep
```

### 3.2 Using Coroutining Facility

If you want to find all solutions, you can use coroutining facility or SolutionHandler interface described in the next subsection.

The previous example program can be rewritten as follows.

```java
Solver solver = new DefaultSolver(net);
for (solver.start(); solver.waitNext(); solver.resume()) {
    Solution solution = solver.getSolution();
    int xv = solution.getIntValue(x);
    int yv = solution.getIntValue(y);
    System.out.println("x = " + xv + ", y = " + yv);
}
solver.stop();
```

The `start()` method starts the solver in a new thread, and immediately returns to the caller. The `waitNext()` method is used to wait the next solution. It returns `true` if the next solution is found, and returns `false` if there are no more solutions. The `getSolution()` method returns the solution. The solver is suspended when the solution is found, and it resumes the execution when the `resume()` method is called. The `stop()` method should be called so that the solver thread is disposed cleanly.

The invocation of the `stop()` method during the search results in the abortion of the solver execution.
3.3 Using SolutionHandler

SolutionHandler can be used to find all solutions. The previous example program can be rewritten as follows.

```java
Solver solver = new DefaultSolver(net);
solver.findAll(new FirstStepHandler(x, y));
```

The `findAll(SolutionHandler handler)` invokes `solved` method of the handler for each solution and at the end of the solver execution.

The following is an example implementation of the `SolutionHandler`.

```java
class FirstStepHandler implements SolutionHandler {
    IntVariable x;
    IntVariable y;

    public FirstStepHandler(IntVariable x, IntVariable y) {
        this.x = x;
        this.y = y;
    }

    public synchronized void solved(Solver solver, Solution solution) {
        if (solution != null) {
            int xv = solution.getIntValue(x);
            int yv = solution.getIntValue(y);
            System.out.println("x = " + xv + ", y = " + yv);
        }
    }
}
```

Another way is the use of `start(SolutionHandler handler)` method. It starts the solver in a new thread, and immediately returns to the caller. You can wait the end of the solver execution by `join()` method.

```java
Solver solver = new DefaultSolver(net);
solver.start(new FirstStepHandler(x, y));
/* other jobs can be performed in parallel */
solver.join();
```

3.4 Searching with Timeout

Above mentioned methods (`findFirst`, `findAll`, and `start`) have optional argument for specifying timeout in milliseconds.

- `findFirst(long timeout)`
- `findAll(SolutionHandler handler, long timeout)`
• `start(long timeout)`
• `start(SolutionHandler handler, long timeout)`

The solver stops the execution when the elapsed time exceeds the given timeout value.
It is also possible to specify the timeout value for the `waitNext` method.

### 3.5 Finding Optimal Solution

The following is an outline of a program to find the optimal solution by complete search.

```java
// set the objective variable
net.setObjective(v);

// set the solver to find the minimal value
Solver solve = new DefaultSolver(net, Solver.MINIMIZE);
for (solver.start(); solver.waitNext(); solver.resume()) {
    // find the next better solution
    Solution solution = solver.getSolution();
    ....
}

// get the best solution
Solution solution = solver.getBestSolution();
```

The method `getBestSolution` always return the best solution up to now.

### 3.6 Using Local Search

Currently, Cream can solve a problem by local search when the problem involves `Serialized` constraints.

• `Serialized(Variable[] s, int[] d)`
  All intervals \([s_i, s_i+d_i-1]\) are not overlapped each other
• Useful for scheduling problems

The following local search algorithms are available in Cream.

• `LocalSearch()`: Random Walk
• `SASearch()`: Simulated Annealing
• `TabooSearch()`: Taboo Search
• `IBBSearch()`: Iterative Branch and Bound
net.setObjective(v);

long timeout = 60000;

Solver solve = new SASearch(net, Solver.MINIMIZE);
for (solve.start(timeout); solve.waitNext(); solve.resume()) {
    // find the next neighbor solution
    Solution solution = solve.getSolution();
    ....
}
solve.stop();

// get the best solution
Solution solution = solve.getBestSolution();

3.7 Using Multiple Local Search Solvers

- Cream also allows to use multiple local search solvers working in parallel (multi thread).
- Each solver randomly post its current best solution
- Other solvers randomly adopt the posted solution to generate the next neighbor

Solver solver1 = new SASearch((Network).net.clone(), Solver.MINIMIZE);
Solver solver2 = new TabooSearch((Network).net.clone(), Solver.MINIMIZE);
Solver[] solvers = { solver1, solver2 };
Solver solver = new ParallelSolver(solvers);

for (solver.start(timeout); solver.waitNext(); solver.resume()) {
    Solution solution = solver.getSolution();
    ....
}
solver.stop();

Solution solution = solver.getBestSolution();

4 API

See Cream API Specification.

5 License

Cream (Class Library for Constraint Programming in Java)
Copyright (C) 2003, 2004 by Naoyuki Tamura (tamura@kobe-u.ac.jp)

Cream is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

6 Acknowledgement

Cream was developed as a part of HECS (HEterogeneous Constraint Solver) system which was supported in part by METI and IPA (The Information-technology Promotion Agency) under grant of 2002 Exploratory Software Project and 2003 Exploratory Software Project. Some improvement ideas of Cream are obtained through the discussion with Mr. Kino, a project manager of IPA Exploratory Software Project and also a developer of K-Prolog and ICS in Java.