Constraint Programming: Assignment no. 2

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

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

This is the first of four graded assignments.

Your task in this week's assignment will be to implement a small constraint modeling language on top of Gecode/J. The constraints we will be dealing with are *temporal constraints*. You will learn

a) how to model problems using temporal constraints.

b) how to compile temporal constraints into finite domain constraints.

c) what reified constraints are and how you can use them to cope with disjunction.

Please submit your solutions by Monday, April 30, 24:00 via email. We would like to encourage you to work in pairs – this will help you understand better what you're doing and will keep our workload on a sensible level.

Exercise 2.1 (Temporal constraints, introduction & warmup)

Temporal constraints state facts about *intervals*, which can be seen as intervals in time (events, activities), hence the name. We are only interested in relations between intervals, not in their actual numeric values. We can consider the following basic relations on intervals:

Relation	Example
x precedes y	xxxx yyyy
x meets y	xxxx yyyy
x overlaps y	xxxx yyyy
x during y	xxxx yyyyyyyyy
x starts y	xxx yyyyyyy
x finishes y	xxx yyyyyyy
x equals y	xxxx yyyy

a) What makes these seven relations *basic*?

b) In addition to intervals and basic relations, we can consider union, intersection and converse of relations. Using these ingredients, model the following problem[1]:

> The meeting ran non-stop the whole day. Each person stayed at the meeting for a continuous period of time. The meeting began while Mr Jones was present and finished while Ms White was present. Ms White arrived after the meeting had begun. In turn, Director Smith was also present but he arrived after Jones had left. Mr Brown talked to Ms White in presence of Smith. Could possibly Jones and White have talked during this meeting?

Hint 1: Use interval variables for the meeting and each person. You will need unions of basic relations – e.g., two events share a period of time if they overlap or one happens during the other or one starts or finishes the other or both are equal. Unions of relations are disjunctions of their semantics.
Hint 2: Think about how to integrate the question whether Jones and White could have talked into the model.

Exercise 2.2 (Compiling temporal constraints into integer constraints, 15P)

In a first step, you will now implement basic temporal constraints using integer variables and constraints.

We will use the following Java classes to write down temporal constraints:

```
import java.util.Vector;

public enum Rel { PRECEDES, MEETS, OVERLAPS, DURING,
                  STARTS, FINISHES, EQUALS};

public class BasicConstraint {
  String interval1; Rel rel; String interval2;
  BasicConstraint(String i1, Rel r, String i2) {...}
}

public class Conjunction {
  Vector<BasicConstraint> c;
  Conjunction add(String i1, Rel r, String i2) {
    c.add(new BasicConstraint(i1, r, i2));
    return this;
  }
}
```

[1] Taken from Apt: Principles of Constraint Programming

Interval variables are marked with strings, and the vector of basic constraints shall be interpreted as their conjunction. For example, the fact that some student *A* was not there when the lecture began, but *B* was there all the time could be modeled as

```
Conjunction c = new Conjunction()
  .add("A", Rel.FINISHES, "lecture")
  .add("B", Rel.EQUALS, "lecture");
```

We want to represent an interval variable as a pair of integer variables that stand for start and end time of the interval, respectively. The basic constraints can then be translated into integer constraints.

```
class Interval {
  public IntVar start; public IntVar end;
  public String name;
}
```

a) For each basic interval constraint, list the integer constraints that implement its semantics.
b) Given n basic constraints, what are the minimal start and maximal end times that you can use as initial domains for the integer variables?
c) Implement appropriate constructors and `toString` methods for the classes given above.
d) Implement a class `TemporalRel`, which is a script (and therefore extends `Space`) and can be constructed from a conjuntion:
```
public class TemporalRel extends Space {
  public TemporalRel(Conjunction c) {
    ...
  }
}
```
e) You need to map interval variables (identified by strings) to pairs of integer variables. To do this, create a `HashMap<String,Interval>`. Then fill this hash table with `Interval` objects for all the intervals in the conjunction. Implement a copy constructor that copies all integer variables contained in the `Interval`s.
f) For each basic relation, add a static method to the `Interval` class that takes two `Interval`s and posts the constraints for the corresponding relation. For example, add a method
```
static void precedes(Space home, Interval a, Interval b) {
...
}
```
Then use these methods to post the constraints for the given conjunction.
g) Implement a first-fail branching.

Exercise 2.3 (Disjunctive temporal constraints, 15P)

The compiler as you just wrote is functional but a bit limited in its use. As you have already seen, actual applications will always require disjunctive constraints like "either A precedes B or B precedes A".

One way of dealing with disjunction is to use *reified constraints*, as introduced in Lecture 2. They all take the form $c \Leftrightarrow b$, where c is some integer constraint and b is a 0/1 variable (called `BoolVar` in Gecode/J). Their semantics is that b is 1 exactly if the constraint c holds. As there are propagators for boolean connectives, we can now express disjunctive constraints: $(x = 3 \Leftrightarrow b_1) \wedge (x = 4 \Leftrightarrow b_2) \wedge (b_1 \vee b_2)$ implements $x = 3 \vee x = 4$ using temporary variables b_1 and b_2.

a) Write a Java method `static void disjoint(Space home, Interval a, Interval b)` that posts the constraint that two intervals do not overlap. Use reified constraints.

b) We change our specification for temporal constraints as follows:

```
public class Disjunction {
  Vector<BasicConstraint> d;
  Disjunction add(String i1, Rel r, String i2) {
    d.add(new BasicConstraint(i1, r, i2));
    return this;
  }
}
public class Conjunction {
  Vector<Disjunction> c;
  Conjunction add(Disjunction d) {
    c.add(d);
    return this;
  }
}
```

Temporal constraints are now interpreted as conjunctive normal forms (conjunctions of disjunctions of basic constraints). Extend your system accordingly. You will have to add reified versions of your temporal relation constraints, e.g.:

```
static void precedes(Space home, Interval a, Interval b, BoolVar b) {
...
}
```

c) Explain why you do not have to branch over the temporary Boolean variables.

d) Implement the Meeting problem.

e) Experiment with different heuristics for branching. Which one gives you the smallest search trees?