

Constraint Programming 2002: Laboration 1

Introduction: In this laboration we will take a closer look on the constraints and representation for these. We will also implement domain filtering, which is removal of inconsistent values from the domains of the variables. The following labs are clearly most easily solved using a higher-level language like **Prolog**. You are however free to choose which programming language you want to use for the implementation. Choosing a lower-level language like **C** or **C++** will give a better insight in how constraint systems are implemented in reality.

Laboration 1-1: The Amazing Tuplifier

The first exercise is pretty straightforward and serve as a warm-up for the later labs. The task is to implement a *tuplifier*, which converts constraints to the tuple notation (sets of *Compound Labels* in Tsang). The function should take as input a constraint represented by a boolean function, a predicate or something else convenient, the variables in the constraint, and the domains of the variables, and return the tuple representation of this. Observe that you are **not** allowed to use any provided constraint reasoning (like *ILOG Solver* or the constraint library `clp(fd)` in *SICStus*) in this lab!

If you prefer C

First download a lab skeleton from the homepage. The files are called `lab1_1.h` and `lab1_1.c`. You probably want to represent a CSP by something like the following.

```
typedef struct {
    int nVals; /* Number of values in domain */
    int *val; /* An array of integers */
} Domain;

typedef struct {
    int nLabs; /* Number of labels in compound label */
    int *var; /* Array of variables */
    int *val; /* array of values */
} CoLabel;

typedef struct {
    int nClabs; /* Number of compound labels in constraint */
    CoLabel **clab; /* Array of pointers to compound labels */
} Constraint;

typedef struct {
    int nVars; /* Number of variables */
    Domain **D; /* Array of pointers to domains of variables */
    int nCons; /* Number of constraints */
    Constraint **C; /* Array of pointers to constraints in problem */
} CSP;
```

Observe that variables are not represented at all. They are referenced using the numbers 0 to `nVars-1`. Take a look at the `main` function for some examples on how to use the skeleton.

You will (at least) implement the following function.

```
Constraint *makeBinaryConstraint(int(*c)(int,int),int x,Domain *dx,int y,Domain *dy);
```

The first argument `int(*c)(int,int)` takes a function pointer, which takes two integers as arguments (the values to test) and returns an `int` (serves as a boolean). An example of a call of this function:

```
int notequal(int x,int y) { return x!=y; }
...
C[0] = makeBinaryConstraint(notequal,0,D[0],1,D[1]);
```

Function pointers are pretty easy as can be seen in the example above. In the function body, you make a call to a function pointer `c` as in the following example.

```
if( (*c)(1,2) ) { ... } else { ... }
```

Here `c` is called with the arguments 1 and 2.

Note that it should be possible to use constraints with other arities than 2 (binary)! You will construct a constraint solver in the following labs, and it is expected that you are able to convert any constraints to tuple notation using what you do in this lab.

If you prefer C++

Download the same files as in the C section above. You should have no problem using the same stuff, but consider using STL instead of the ugly C-hacks I've been using.

If you prefer Prolog

You probably want to represent a CSP by something like this:

```
X=msp(
  vars(X),
  domains(D),
  constraints(C))
```

where `X` is a list of terms (variables), `D` is a list of lists of values (domains), and `C` is a list of lists of terms. Each term (compound label, tuple) could be something like `tuple(a(0),b(1))`, representing the tuple $\langle\langle a,0 \rangle\langle b,1 \rangle\rangle$. Another approach is to merge the domain and variable parts and put each domain in the corresponding variable.

You will (at least) construct the predicate `makeBinaryConstraint(C,X,DX,Y,DY,?Trep` or something equivalent. The predicate takes a constraint `C`, two variables `X` and `Y`, two domains `DX` and `DY`, and binds the variable `Trep` to a list of terms (or whatever you choose) representing a constraint. For example, you should be able to use

```
:- makeBinaryConstraint(notequal,a,[1,2,3,4],b,[1,2,4,5],T).
```

This should bind `T` to the tuple representation of `notequal` considering the domains `[1,2,3,4]` and `[1,2,4,5]` for `a` and `b` respectively.

Note that it should be possible to use constraints with other arities than 2 (binary)! You will construct a constraint solver in the following labs, and it is expected that you are able to convert any constraints to tuple notation using what you do in this lab.

Laboration 1-2: Node Consistency

Implement one of the node consistency algorithms from Tsang or the course material (slides). Your function / predicate / whatever should take a CSP as input, and apply the algorithm on it to make it node consistent. The output should be either nothing (modify the input CSP) or a new domain-reduced CSP.

Laboration 1-3: Arc Consistency, AC-1

Implement AC-1. As in the previous lab, the input to the algorithm is a CSP, and the output should be either nothing (modify the input CSP) or a new domain-reduced CSP.

Laboration 1-4: More Arc Consistency

Implement a more efficient arc-consistency algorithm from the course material or from articles (AC-3, AC-4, AC-5, AC-6, AC-7 and AC-8 are all available, in roughly increasing order of efficiency).

Laboration 1-5: Comparison

Run your two algorithms on the following problem P .

$$\begin{aligned} P &= (X, D, C) \\ X &= \{v_1, v_2, \dots, v_{100}\} \\ D_{v_1} &= D_{v_2} = \dots = D_{v_{100}} \\ &= \{1, 2, \dots, 100\} \\ C &= \{C_{v_i, v_j} \mid i < j \wedge v_i, v_j \in X\} \\ \forall v_i, v_j \in X : i < j \rightarrow C_{v_i, v_j} &= v_i < v_j \quad (v_i \text{ should be less than } v_j) \end{aligned}$$

The problem consists of 100 variables, each with a domain of 100 values ranging from 1 to 100. For each variable pair v_i, v_j where $i < j$, there is one constraint stating that the value of v_i should be less than the value of v_j .

Run your algorithms on the problem, and note down your results.

1. Which algorithm were faster? Why is that?
2. How much was the problem reduced?
3. Suppose we have 99 constraints, one for each pair of variables v_i, v_{i+1} stating that $v_i < v_{i+1}$.
 - (a) Are the problems equivalent?
 - (b) Solve the new problem using one of your methods. Do you note any differences in the results and the runtime?