600.432/632 Declarative Methods - Prof. Jason Eisner
Answers to "CSP Encodings" discussion problems

1. (a) Any method that prevents two x,y pairs from having the same values is fine. My favorite:

```
alldifferent([Ax*4+Ay, Bx*4+By, ...])
```

(Yes, you actually can write this in ECLiPSe. It is equivalent to alldifferent([A,B,...]), A #= Ax*4+Ay, B #= Bx*4+By, ...)

(b) Expand the placement area to allow additional space beyond the 6x4 suitcase. Instead of requiring the x variables to be in 0..5, allow them to be in 0..12.

Or you could allow the y variables to be in 0..11. Or both.

(Why 0..12? Since the total width of the example pieces is 13, the hard constraints will certainly be satisfiable now. Why not go all the way up to 0..infinity? Because this raises a danger that the solver might backtrack forever because there are infinitely many places to put the pieces. Although I suspect that any decent constraint propagation strategy would still allow branch-and-bound to terminate in finite time.)

(c) The tricky part here is that you're not trying to get as many little squares into the suitcase as possible. You're trying to get a complete piece fully into the suitcase. If any little bit of it sticks out, in either the x or the y dimension, you're sunk.

Here's the easiest approach. Allow the x variables to be in 0..12, and continue to require the y variables to be in 0..3. A piece failed to get packed iff its rightmost edge is out of the suitcase, i.e., > 5. So we have

```
Cost #= (A #> 5) + (H #> 5) + (L #> 5) + (P #> 5) + (S #> 5)
```

2. (a) Some of you forgot the domain and alldifferent constraints, which are the key to encoding the problem. Beyond these, the 14 no-crossing constraints below iillustrate a systematic, mechanical way of doing the encoding. (For example: for each pair of vertices in the top layer, such as B and C, write down that B's neighbors < C's neighbors iff B < C. Or: iterate systematically through all pairs of disjoint edges, writing down one constraint for each.)

```
[A,B,C,D,E]::1..5,
[W,X,Y,Z]::1..4,

alldifferent([A,B,C,D,E]),
alldifferent([W,X,Y,Z]),

(A < C) #= (X < Y),   % AX, CY
(A < C) #= (X < Z),   % AX, CZ
(A < E) #= (X < Z),   % AX, EZ
(A < B) #= (X < W),   % AX, BW
(D < C) #= (X < Y),   % DX, CY
(D < C) #= (X < Z),   % DX, CZ
(D < E) #= (X < Z),   % DX, EZ
(D < B) #= (X < W),   % DX, BW
(C < E) #= (X < Z),   % CX, EZ
(C < B) #= (X < W),   % CX, BW
(C < E) #= (Y < Z),   % CY, EZ
(C < B) #= (Y < W),   % CY, BW
```

```
     (C < B) #= (Z < W),   % CZ, BW
     (E < B) #= (Z < W),   % EZ, BW

     labeling([A,B,C,D,E,W,X,Y,Z]).   % ok to skip this line
```

(Question: Can you guess how many labelings there are?
Use ECLiPSe to find out if your guess was right ...)

(Question: What do you think of using "relaxation" on
such a problem? That is, try solving with no constraints;
then add constraints to block the crossing edges that you
actually got; rinse & repeat. Is this efficient?)

Some of you tried to write a SINGLE alldifferent constraint
over all 9 variables. But you can't have 9 variables all with
different integer values in 1..5!

Some of you tried to abbreviate

```
       (A < C) #= (X < Y),
       (A < C) #= (X < Z),
```

as

```
       (A < C) #= ((X < Y) and (X < Z)).
```

But this is not equivalent! It would incorrectly allow
(A > C), (X < Y), (X > Z). The equation holds because both sides
are false. The problem is that ((X < Y) and (X < Z)) can be
false without BOTH of its conjuncts being false.

(b) The question is, what values of the triple (A,C,E) allow
these two constraints to be satisfied?

The solver could infer that this is possible only for triples
such that

```
   (C < E) #= (A < C)
```

So if the original two constraints are satisfied, this new one
must be satisfied too.

What does the constraint mean in English? Either both sides of
the #= equation are true (so A < C < E) or else both sides are
false (so A > C > E). So the constraint means that C is
between A and E.

You can see by staring at the graph that this betweenness will
have to be true. We derived the betweenness from the four
edges CX, EZ, AX, CZ that caused the original constraints. So
in fact it follows from just the arrangement of those edges:

```
              A  C   E
            \ / \ /
      X   Z
```

Remark: Notice that the solver can perform this inference by
doing symbolic reasoning (using the transitive property of
equality). In class, we considered both symbolic reasoning and
exhaustive checking strategies. How would exhaustive checking
do this job? It would consider *all* values of (A,C,E,X,Y)
that satisfy both of the original constraints. It would then
see which (A,C,E) triples survive. However, it might have
trouble recognizing that these (A,C,E) triples could be
described by the new constraint (C < E) #= (A < C). And it

could be very slow if the domains were large (for a bigger graph) or infinite (for another problem). (On the other hand, exhaustive checking's brute force might be able to discover things that a symbolic reasoner wouldn't.)

(c) For each ordered pair of distinct variables in the same layer, P and Q, create a boolean variable with the name P<Q. (We will also create one with the name Q<P.)

Our old constraints can now be written like this:

```
    (A < C) #= (X < Y)    becomes   A<C iff X<Y
    (D < C) #= (X < Y)    becomes   D<C iff X<Y  (notice the negation)
```

This is all fine, but the use of the character "<" is purely mnemonic at this point. What guarantees that the assignment to these booleans can really be decoded into a real ordering?

Nothing yet! The above constraints could easily be satisfied by making all the variables true (or false).

We need additional constraints that characterize the axiomatic properties of <. For every pair of distinct variables in the same layer, add the constraint

```
P<Q <--> !Q<P    % exactly one of P<Q, Q<P is true
```

and for every triple of distinct variables in the same layer, we need

```
P<Q ^ Q<R --> P<R   % transitivity of <
```

This was not necessary in part (a), where we had numeric variables and ECLiPSe guaranteed the real meaning of "<" with its usual properties. But the SAT solver would be happy to violate those properties if you didn't state them explicitly.

The problem only asks for SAT, not CNF-SAT, so you're not required to eliminate the <--> and --> symbols. It's fine to keep them.

(d) For (c), you should use a MAX-SAT solver. Constraints like transitivity are hard constraints and should get a weight of infinity. Put a weight of 1 on each of the no-crossing constraints, such as A<C <--> X<Y. Then the total weight of unsatisfied clauses (if finite) will be the number of crossings. Maximizing the number of *satisfied* clauses will minimize the number of crossings.

Typically, a MAX-SAT solver works only with CNF formulas. You'd want to rewrite A<C <--> X<Y as two clauses (~A<C v X<Y) ^ (A<C v ~X<Y). Then you can put a weight of 1 on each clause. This has the same effect as before: if the edges don't cross, they contribute 0 unsatisfied clauses, and if they do cross, they contribute 1 unsatisfied clause.

For (a), you could use a branch-and-bound solver. Replace the crossing constraints with a cost:

```
    Cost    #= ((A < C) #\= (X < Y))
            + ((A < C) #\= (X < Z))
                      + ((A < E) #\= (X < Z))
                      + ...
```

Use the solver to minimize this Cost (subject to the remaining constraints, such as alldifferent).

Note that the equalities #= have been replaced with inequalities #\=, since the cost is the number of original constraints that are NOT satisfied.

3. (a) Start by defining a---the number of correct colors in the correct position:

```
a #= (P1 #= Q1) + (P2 #= Q2) + (P3 #= Q3),
```

Now let's move to c---the total number of **matched** colors (both in correct and wrong position). Remember that, for each color, the number of matched letters cannot exceed the number of letters of a that color in P:

```
c #= MatchGreen + MatchYellow + ... ,
```

where

```
min([PGreen,QGreen], MatchGreen),
min([PYellow,QYellow], MatchYellow),
...
```

and

```
PGreen = (P1 #= green) + (P2 #= green) + (P3 #= green),
QGreen = (Q1 #= green) + (Q2 #= green) + (Q3 #= green),
...
```

(b) Each challenge gives some constraints on P. Put all of these constraints together. Run backtracking search on the resulting constraint program to count the number of solutions and see whether it equals 1.

This search can be sped up using arc consistency each time a value is assigned, to determine early whether the partial assignment is UNSAT.

(c) Input: k triples (Q1,a1,b1), (Q2,a2,b2), ... (Qk,ak,bk), where each Qi is a challenge and (ai,bi) is the response.

Output: Yes (or SAT) if there exists any secret code P such that each (ai,bi) is the appropriate response to the corresponding challenge Qi. No (or UNSAT) otherwise.