600.432/632 Declarative Methods - Prof. Jason Eisner
Answers to "More Miscellany" discussion problems

1. (a) We'll have one variable per vertex, which is 0 for a red vertex and 1 for a blue vertex. Since these are boolean variables, we have a SAT problem. The reason that we only need boolean variables is that we are only doing 2-coloring (k=2).

The reason that it is 2-SAT (rather than 3-SAT, etc.) is that we only have binary constraints: each edge only connects 2 variables.

(In other words, the "2" part of "2-SAT" comes from the fact that we have a graph, i.e., binary constraints. What k=2 contributes is rather the "SAT" part of "2-SAT," i.e., the fact that the variables are boolean. Relaxing EITHER of these restrictions would make the problem NP-complete.)

Here are some of the binary constraints:

A xor B (since A and B are connected)
B xor C (since B and C are connected)

And each of those binary constraints can be turned into a pair of binary CNF constraints: for example, (A xor B) becomes

(A v B) ^ (~A v ~B)

Note: Especially in the case of graph coloring, you may prefer to think of these constraints as implications. Instead of A xor B, you could equivalently write

(A <-> ~B)

which turns into

(~B -> A) ^ (A -> ~B)

which turns into

(B v A) ^ (~A v ~B)

which is equivalent to the above.

(b) Notice that arc consistency is equivalent to unit propagation in this case.

We must make just one decision for each connected component of the graph -- in the example, just 2 decisions.

Once we pick a color for any vertex in that component (decision variable), arc consistency forces assignments to the rest of the vertices.

In the example, if we color D=blue in the first component, then its neighbors A,C,I (at distance 1 from D) are forced to be red, so their neighbors B,E,H (at distance 2 from D) are forced to be blue, etc., until the whole component is colored.

We can then pick J or K in the next component as our next decision variable.

If the graph is 2-colorable (SAT), as in the example, we will find a solution without backtracking. Variable ordering doesn't matter in this case.

If it is not 2-colorable (UNSAT), we will have to backtrack. You can see from the explanation above that unsatisfiability must always be the result of an odd-length cycle in some component, e.g., some vertex is at both even and odd distances from the decision vertex D.

For example, if we assign D=blue and it runs into a conflict, we'll backtrack and try D=red, which won't work either. If that still doesn't work, we may backtrack into previous components that were successfully colored, and try flipping their variables as well! So if there were 99 previous components, then we may try 2^100 combinations of decision variables (one per component) before concluding that the problem is UNSAT!

Adding backjumping to the solver will help with this because it tells us that changing decision variables in to previous components won't help us solve the conflict in this component. After trying D=blue and D=red, we can just backjump over all the previous decision variables and return UNSAT.

Regardless of whether we have backjumping, variable ordering does matter in the UNSAT case. If our first decision variable is in an uncolorable component, we will be able to "fail fast" by detecting UNSAT quickly right on the first decision variable, without even looking at other components.

(c) The constraints involving D are

```
D != A
D != C
D != I
```

To eliminate D, we join these constraints and project onto (A,C,I) to get a new constraint, namely:

```
A = C = I
```

This can be represented graphically by gathering {A,C,I} into a single complex node that will get a single color; it is connected to E, H, and B, requiring them to have a different color than {A,C,I} does.

If we find a coloring of the reduced graph that satisfies the new constraint A=C=I, then we will be able to add D back in and successfully give it the opposite color.

So how do we find a coloring of the reduced graph?

If we eliminate A, the constraints involving A are

```
A != E
A != B
A = C = I
```

Joining these and projecting, we get a new constraint on (E,B,C,I) that says E and B must be red while C and I are blue, or vice-versa.

Again, this can be represented graphically by gathering {E,B} into a complex node and {C,I} into another complex node that is connected to {E,B}.

Actually the {C,I} node is just the same {A,C,I} node as before; it's just that A has just been eliminated from it. The

reason that we merged {E,B} is that they are the surviving neighbors of A.

So when applied to graph 2-coloring, variable elimination can be regarded as gathering the variables into clusters, each of which is uniformly colored. This is fast (unlike variable elimination in the general case).

Once we are down to a single variable, we can un-eliminate the other variables and assign them in a backtrack-free way. The first variable we assign within a connected component can be freely chosen, and then the rest will all be prescribed.

How do we detect UNSAT (in the case of an odd-length cycle)? This happens when we eliminate a variable that is connected to two clusters that are connected to each other. In other words, the original odd-length cycle has been shrunk down to a 3-cycle, and now the resulting elimination is trying to shrink it down to a 1-cycle. The resulting join will produce an unsatisfiable constraint, leaving empty domains for the variables involved, at which point we can safely say the problem is UNSAT.

2. This is a typical example of a hard (probably NP-hard) problem that you might face in the real world. I was quite surprised at how few people mentioned any concepts from our course! Some of you proposed ad hoc procedural heuristics, rather than using the declarative tools you've learned. Some of you suggested using machine learning, but as the problem is stated, there is nothing to learn -- only something to optimize.

   First let's formulate it as an ECLiPSe-style problem:

   For each i from 1 to n, let integer variable

       M[i] :: 1..m represent

   the processor where process i will run. The idea is to pick a low-cost assignment to the M[i] variables. We will write the global cost of the assignment below in terms of those variables.

   This problem's only hard constraints are the domain constraints saying that M[i] :: 1..m and M[i] is an integer. Everything else is in the soft objective.

   To minimize the global cost of the assignment, you could use branch and bound (e.g., the minimize method in ECLiPSe).

   But it might be faster to use a technique like hill-climbing, simulated annealing, or Gibbs sampling -- this will get you a pretty good answer pretty fast. In other words, you can try picking different variables M[i] and changing their values, thereby tentatively assigning process i to a new processor. Your technique should favor changes that reduce the cost function.

   To be explicit, the cost function looks something like this:

       migration cost + communication cost + inequality cost

   where

       migration cost = sum_i (S[i] * (M[i] != L[i]))
       communication cost = sum_i sum_j C[i,j] * (M[i] != M[j])
       inequality cost = sum_k (N[k] - n/m)**2

(note that N_k = n/m would represent a fair allocation), with

```
N[k] = sum_i (M[i]==k)
```

(Of course, you are free to describe this sum as a bunch of soft constraints whose costs are added together. For example, the summand S[i]*(M[i] != L[i]) could be incorporated via a constraint M[i] #= L[i] with weight S[i].)

To make techniques like simulated annealing efficient, you need to be able to quickly compute how much the cost will change if you change the value of M_i from k to k'. But this change only affects at most 1 summand in the migration cost, at most n−1 summands in the communication cost, and at most two summands in the inequality cost (by changing N_k and N_k'). So the trick is to see how much only those summands change, rather than recomputing the cost function from scratch.

Now, suppose you wanted to do this using ILP. The first issue is that you'd like a good LP relaxation. So it probably doesn't make sense to have M[i] :: 1..m, since there's no meaning to processor 3.14 as being partway between processor 3 and processor 4: they're not all in a line. Instead, let's have M[i,j] be a binary variable that is 1 if process i is on processor j. The ZIMPL program is something like this:

```
param n, m;
set Process := {1..n};
set Processor := {1..m};

var M[Process*Processor] binary;    # new assignment
var N[Processor];                   # number of processes per processor
var migrate[Process];               # let process i migrate?
var network[Process*Process];       # let two processes talk over network?
var contention[Processor];          # excess load on processor

param L[Process*Processor] ...;  # current assignment
param S[Processor] ...           # sizes
param C[Process*Process] ...     # communication from i to i2 (asymm.)

# each process on one processor, but N[j] processeses on processor j
subto forall <i> in Process: (sum <j> in Processor: M[i,j]) <= 1;
subto forall <j> in Processor: (sum <i> in Process: M[i,j]) == N[j];

# process shouldn't appear on a new processor
# but migrate[i] is a slack variable
forall <i,j> in Process*Processor: M[i,j] <= L[i,j] + migrate[i];

# if process i is on processor j, then process i2 should be there too
# but network[i,i2] is a slack variable
forall <i,j,i2> in Process*Processor*Process:
    M[i,j] <= M[i2,j] + network[i,i2];

# no processor j should have too many processes
# but contention[j] is a slack variable
forall <j> in Processor: N[j] <= n/m + contention[j];

# Total cost.  Note that we use a linear penalty for overload, not
# quadratic as before, since this is integer LINEAR programming.
    minimize cost:
        (sum <i> in Process: migrate[i]*S[i])
        + (sum <i,i2> in Process*Process: network[i,i2]*C[i,i2])
        + (sum <j> in Processor: contention[j]);
```

3. member(3,Xs).

Returns:

```
Xs = [3 | _]
Xs = [_, 3 | _]
Xs = [_, _, 3 | _]
...
```

where all of the _ symbols are internal variables.

4. (a) A=5, B=2;
A=2, B=5

(b) Here is one possible solution:

```
indomain(0).
indomain(1).
indomain(2).
indomain(3).
indomain(4).
indomain(5).

solve(A,B) :- indomain(A), indomain(B), 7 is A+B, 10 is A*B.
```

or, using Peano integers instead of the "is" operator:

```
indomain(z).
indomain(s(z)).
indomain(s(s(z))).
indomain(s(s(s(z)))).
indomain(s(s(s(s(z))))).
indomain(s(s(s(s(s(z)))))).

% Addition: add(X, Y, Z) means X + Y = Z
add(z, Y, Y).
add(s(X), Y, s(Z)) :- add(X, Y, Z).

% Multiplication: mult(X, Y, Z) means X * Y = Z
mult(z, _, z).
mult(s(X), Y, Z) :- mult(X, Y, W), add(Y, W, Z).

solve(A, B) :-
    indomain(A),
    indomain(B),
    add(A, B, s(s(s(s(s(s(z)))))))),      % A + B = 7
    mult(A, B, s(s(s(s(s(s(s(s(s(z)))))))))).  % A * B = 10
```

(c) Approximately n^2, since it would consider all (A,B) pairs

(d) Both A and B's domains have been reduced to 2..5 from the domain constraints together with the constraint A+B #= 7. The constraint A*B #= 10 does not reduce the domains further.

(e) The additional constraint A #=< B would prevent symmetric solutions.

(f) Unification. Prolog can't do a constraint like A+B #= 7, which ensures that if we learn something about A, then we will learn something about B. But Prolog can do A=B or f(A)=B, which also ensures that if we learn something about A then we will learn something about B (not to mention learning something about f(B,C)).

(g) Valid solutions are things like

```
01100 + 10011 = 11111 = 2^5 - 1
10001 + 01110 = 11111 = 2^5 - 1
```

Note that carrying is not possible in such a sum. (The reason is that carrying only happens when we add 1+1=10. The rightmost instance of such a carry would give a 0 bit in the sum, whereas we are looking for a sum that is all 1 bits.) This simplifies things.

In fact, what we need is very simple: for A to have 0 bits wherever B has 1 bits, and vice-versa.

Let a1, a2, ..., an be the bits of A,
and b1, b2, ..., bn be the bits of B.
We encode a bit value of 0 or 1 as false or true respectively.

We need constraints of the form

```
(a1 xor b1) ^ ... ^ (an xor bn)
```

or equivalently

```
(a1 <-> ~b1) ^ ... ^ (an <-> ~bn)
```

(h) This can be done through Peano integers:

```
add(A, B, s(s(s(s(s(s(z)))))))).
```

where addition is defined for example by

```
add(z, B, B).
add(s(A), B, s(Sum)) :- add(A, B, Sum).
```