600.325/425 Declarative Methods - Prof. Jason Eisner
Answers to "Miscellaneous Problems" discussion problems

1. (a) All variables now have domain [0..30].

To see that these domains are bounds-consistent, consider A. Its maximum value of 30 is supported in the sense that for each other constraint, we can pick in-domain values for B,C,D,E that satisfy that constraint (in this case, picking, B=C=D=E=0 always works). Similarly, the maximum and minimum values of all domains are supported.

We arrived at these domains by reducing the original domains [0..infinity] only as much as needed to ensure bounds consistency.

(b) A+B+C+D+E #= 65

(c) The [0:30] domains are not reduced further, since they are still bounds-consistent with respect to the new constraint as well. For example, if A=30, then we could again choose B=C=D=E=0 to satisfy the new constraint.

(d) You have to show both (i) and its converse (iii).

That is, the 2 new constraints on (B,C,D,E) should be equivalent to
"There exists an A in [0..infinity]
such that A+B+C <= 30 and A+B+C+D+E = 65."

I promised to show that this was true. To show (i), observe that if there exists such an A, then the 2 new constraints follow:

B+C = (A+B+C)-A = ((something <= 30) - (something >= 0)) <= 30.
D+E = (A+B+C+D+E)-(A+B+C) = (65 - (something <= 30)) >= 35.

And conversely, to show (iii), if the two new constraints hold over (B,C,D,E), then we can extend the assignment by setting A = 30-(B+C), and the original constraints on (B,C,D,E) will be satisfied.

(e) i. D #>= 5, E #>= 5. In other words, the domains are reduced to [5..30].

The original domains [0..30] are no longer bounds-consistent: the values 0..4 are no longer supported. For example, we know that D can't be 4 because then E would have to be at least 31, which is out of domain.

ii. D :: [5..30] is not bounds consistent with respect to C+D+E <= 30. (Why not? D can't take the value 30 because then we cannot choose C >= 0 and E >= 5 such that C+D+E <= 30.)

(As a result, one step of bounds propagation will reduce D to [5..25], and E similarly. But now the domains of D and E are no longer bounds-consistent with respect to D+E #>= 35; they must be reduced to [10..25]. Continuing bounds propagation in this way will shrink the domains to emptiness, which proves UNSAT.)

2. (a) VERSION 1:

For each of the possible ways to satisfy ONLY the 3-clauses, we'll run 2-SAT to try extending this partial assignment (which covers only the <= 9 variables in the 3-clauses) to satisfy the

entire formula.

There are only <= 2^9 partial assignments to try, a constant number. For each, we construct a 2-SAT formula and try solving it. This 2-SAT formula consists of all the original 2-clauses, plus unit clauses such as (A), (~B), ... that enforce our assignment A=1, B=0, ... that satisfies the 3-clauses.

VERSION 2 (similar):

Pick one arbitrary variable from each 3-clause. Assuming for simplicity that there are three 3-clauses, let's call these variables A, B, C.

Each of the 2^3 possible assignments to A, B, C can be used to simplify the original formula to a 2-SAT formula, which you can solve with your 2-SAT solver. For example, if A=1, you can cross off all clauses that contain A, and shorten all of the clauses that contain ~A. Either way, you have gotten rid of the 3-clause that originally contained A or ~A.

Another way of carrying out this simplification is to add 3 unit clauses that encode the assignment to A, B, C, just as in version 1, and then run unit propagation. This will cross out and shorten clauses exactly as in the previous paragraph.

Thus, another way of thinking about this solution is that you are just running DPLL on your 2-and-a-few-SAT problem, with A, B, C coming at the start of your variable ordering. Once the backtracking search has recursed 3 levels down, it has assigned A, B, C and carried out unit propagation. It is then left with a 2-SAT problem that it solves by recursively calling DPLL, which solves 2-SAT fast.

(In fact, I suspect that for ANY variable ordering, DPLL is guaranteed to solve 2-and-a-few-SAT in polytime, just as it does for 2-SAT. Can you prove this?)

VERSION 3 (doesn't work):

Some of you said you'd solve just the 2-SAT part, and then test the resulting assignment to find out whether it also satisfied the 3-clauses. The problem is that if it doesn't, you have to backtrack to try another solution to the 2-SAT problem. And there might be exponentially many such solutions. :-(

So this DOESN'T give a polynomial-time solution. Note that both Versions 1 and Version 3 break the problem into two "big constraints," namely "satisfy all the 3-clauses" and "satisfy all the 2-clauses." The difference is in which of the "big constraints" they try to handle first.

VERSION 4 (doesn't work):

Many of you tried to convert the 3-clauses to 2-clauses by using switching variables. Not a workable approach!

As a reminder, you can
convert a 4-clause to 3-clauses with a switching variable Z:
    (A v B v C v D)  becomes  (A v B v Z) ^ (~Z v C v D)
but that doesn't work for converting a 3-clause to 2-clauses:
    (A v B v C)  becomes  (A v B v Z) ^ (~Z v C)
which still contains a 2-clause.

VERSION 5 (doesn't work):

MANY of you tried to construct other tricks for turning the 3-clauses into 2-clauses.

But none of the tricks you tried could possibly have worked either! All of them were methods for cheaply replacing a 3-clause with a constant number of 2-clauses. This could be used to turn any 3-SAT problem into a 2-SAT problem that is only bigger by a constant factor ... thus proving that P=NP.

This should have set off big alarm bells in your head.

In principle, there might be a reduction that DOUBLES the formula length every time it eliminates a 3-clause. This wouldn't set off alarm bells, because it would give you a polytime solution to 2-and-a-few-SAT, but an exponential-time solution to 3-SAT. However, I can't find such a reduction, and I have strong reasons to doubt that one exists.

(b) There were several different kinds of wrong answers:

- Some of you tried to reduce 2.5-SAT to 3-SAT -- which is backwards -- despite the clear hints in the problem.

- Some of you tried to show why the method of part (a) would not help you solve a 2.5-SAT problem fast. But that certainly doesn't mean that no fast solver exists for 2.5-SAT! It only means that the particular method of (a) is not fast.

- Some of you referred to "the 2.5-SAT problem" as if one already existed. It doesn't! You are given a 3-SAT problem and you're asked to construct the 2.5-SAT problem.

Here is a reduction of 3-SAT to 2.5-SAT as needed:

If the 3-SAT problem is a conjunction of m 3-clauses, then add m copies of the 2-clause (X v ~X). This 2-clause is trivially true, so it does not further constrain the variables or change the satisfiability of the formula.

But what if some clauses in the 3-SAT problem are 2-clauses or 1-clauses? (Only a few of you thought about this possibility.) Then just pad them out to 3-clauses by repeating literals. For example, (A v ~B) becomes (A v ~B v ~B). Again, this does not change the satisfiability. Then you can apply the technique in the previous paragraph to finish the conversion to 2.5-SAT.

3. (a) Yes. For example, add(z,foo,foo) is provable even though foo is not a Peano integer. (It is only the first argument to add that must be a Peano integer.)

(b) negate(pos(X),neg(X)).
negate(neg(X),pos(X)).

Note: The problem was originally phrased so that you also needed the following clauses to be strictly correct -- this is a special case for zero:

```
negate(pos(z),pos(z)).
negate(neg(z),neg(z)).
```

(c) Suppose we are called with signed_add(XX,YY,ZZ), where XX, YY, and ZZ are all signed Peano integers. We will divide up systematically into a few cases

according to which arguments are pos or neg.

```
% If XX is pos, we have 3 cases as follows.

   % If YY is also pos, everything is easy.
   % For example, pos(5) + pos(2) = pos(7).
   % We just use add to find that 5+2 = 7.
signed_add(pos(X),pos(Y),pos(Z)) :- add(X,Y,Z).

   % If YY is slightly negative, the result will be
   % positive.  For example, pos(5) + neg(2) = pos(3).
   % To get that 3, we have to find 5-2.
   % This subtraction is accomplished by calling add(2,Z,5).
signed_add(pos(X),neg(Y),pos(Z)) :- add(Y,Z,X).  % fails unless Y <= X

   % If YY is slightly negative, the result will be
   % positive.  For example, pos(5) + neg(9) = neg(4).
   % To get that 4, we have to find 9-5.
   % This subtraction is accomplished by calling add(5,Z,9).
signed_add(pos(X),neg(Y),neg(Z)) :- add(X,Z,Y).  % fails unless Y >= X

% The above handle all the cases where XX is pos.
% If XX is neg, we flip the signs of all three arguments,
% and call the above code.

signed_add(neg(X),Y,Z) :-
    negate(Y,SY), negate(Z,SZ), signed_add(pos(X),SY,SZ).
```

Remark: It is possible for more than one of these cases to succeed. That's because we have two representations of zero and both are correct. For example, any of these queries

```
    signed_add(pos(s(s(z))), neg(s(s(z))), Result)
    signed_add(pos(s(s(z))), Result, pos(s(s(z))))
    signed_add(Result, pos(s(s(z))), pos(s(s(z))))
```

will return

```
    Result=pos(z) ;
    Result=neg(z)
```

which is fair enough. If you really wanted to block that behavior, the easiest way is to add a cut at the end of each signed_add rule, so that if any rule succeeds we don't consider the others. But it might be better to pick a representation that only has 1 representation of 0 in the first place. (For example, you could use a functor neg_s, so that neg_s(z) represents -1, neg_s(s(z)) represents -2, etc. So now the only way to represent 0 is pos(z). On the other hand, that would slightly complicate the code below.)

4. (a) Because that query should always return No.
And No is already the default -- things only become provable if we include rules that make them so.

We don't want to include any rules that would prove false facts matching deletefirst(Z,[],Ys).

(b) You'd get Z=1, Ys=[2,3].
The "missing" answers are Z=2, Ys=[1,3] and Z=3, Ys=[1,2].

(c) Some of you wrote

```
reverse([],[]).
reverse([A|As],Bs) :- reverse(As,[A|Bs]).
```

But this can't be right, because the two arguments always ought to be the same length, and the recursive call shortens one argument while lengthening the other.

Nonetheless, this is on the right track. You do indeed want to pull elements off the front of As and stick them on the front of something. But you want that something to start out as the empty list.

Remember that we managed to do reversal in class using an incorrect version of append/3, which we called appendrev/3.

```
reverse(As,Bs) :- appendrev(As,[],Bs).
appendrev([],Bs,Bs).
appendrev([A|As],Bs,Result) :- appendrev(As,[A|Bs],Result).
```

Example showing the recursive calls:

```
reverse([1,2,3],Bs)
  appendrev([1,2,3],[],Bs)
    appendrev([2,3],[1],Bs)
      appendrev([3],[2,1],Bs)
        appendrev([],[3,2,1],Bs)
          succeeds with Bs=[3,2,1] as desired
```

Runtime is O(n).

There are other solutions that are less efficient. Some of you wrote something like this:

```
reverse([],[]).
reverse([A|As],[Bs|A]) :- reverse(As,Bs).
```

Unfortunately, this means that reverse([1,2,3],Bs) gives Bs = cons(cons(cons([],3),2),1), which is not a well-defined list -- the second argument to cons is supposed to be a list. But the solution can be fixed as some of you did:

```
reverse([],[]).
reverse([A|As],Bs) :- reverse(As,Temp), append(Temp,[A],Bs).
```

Unfortunately this is slower than necessary -- it takes O(n^2) time, because when reversing [1,2,3] it builds up successive lists [], [3], [3,2], [3,2,1] from scratch. But it is correct.

(d)
```
deletelast(Z,Xs,Ys)
   :- reverse(Xs,RXs),deletefirst(Z,RXs,RYs),reverse(RYs,Ys).
```

The total time is O(n). Each of the three constraints succeeds in O(n) time in finding a unique ground value for its last argument.

Some of you forgot to reverse the result of deletefirst back again.
Some of you put the two reverse constraints before the deletefirst constraint; that is pretty but unfortunately it won't terminate under the conditions given (i.e., you can't efficiently reverse a list of unknown length).
Some of you put the deletefirst constraint before the reversals; that won't work either since deletefirst assumes that its second argument is ground as well.

(e)
```
deletelast(Z,[X|Xs],[X|Ys])
    :- member(Z,Xs), !, deletelast(Z,Xs,Ys).
deletelast(Z,[Z|Xs],Xs).
```

This takes runtime O(n^2). It is only able to proceed by either verifying that there is a later copy of Z to delete, or by deleting the first element if it is the

last copy of Z.

You could make it a bit more efficient, though still O(n^2), by only doing the member check if the input list starts with Z -- otherwise you already know you can't delete the starting element.

```
deletelast(Z,[Z|Xs],[Z|Ys]) :- member(Z,Xs), !, deletelast(Z,Xs,Ys).
deletelast(Z,[Z|Xs],Xs) :- !.
deletelast(Z,[X|Xs],[X|Ys]) :- deletelast(Z,Xs,Ys).
```

This code can be read: "If Z is the first element but there's a later copy of Z, delete the last copy. Else, if Z is the first element (but no later copy), delete the first element. Else, Z doesn't appear anywhere in the input list, so

(f) The simplest and most elegant solution is to recognize that the recursive call ALREADY indicates whether it managed to delete Z -- by failing if it can't!

Thus, the solution above can be simplified to just

```
deletelast(Z,[X|Xs],[X|Ys]) :- deletelast(Z,Xs,Ys), !.
deletelast(Z,[Z|Xs],Xs).
```

In other words, if we can delete Z from the tail, do that. Otherwise try to delete Z from the head. Otherwise we're done.

Below are some solutions that have an explicit return code -- but they are really just complexified versions of the elegant solution above. They make use of a predicate deletelastmaybe/4 that always succeeds (at least when called in mode +,+,+,-) but uses an extra argument to indicate whether it "really" succeeded.

```
deletelast(Z,Xs,Ys) :- deletelastmaybe(Z,Xs,Ys,1).
```

Here deletelastmaybe(Z,Xs,Ys,P) will try to find a Z in Xs. If it finds any copies of Z in Xs, it deletes the last copy to get Ys, and binds P=1. If Z does not appear in Xs, it binds Ys=Xs and P=0. So P is a kind of success code. Note that the call above insists on success (P=1), since we are supposed to fail if there were no copies of Z to delete.

```
deletelastmaybe(Z, [], [], 0).
deletelastmaybe(Z, [X|Xs], Result, P)
   :- deletelastmaybe(Z,Xs,Ys,P1),
      ok(Z,X,P1,P,Ys,Result).
ok(Z,Z,0,1,Ys,Ys) :- !.
ok(Z,X,P,P,Ys,[X|Ys]).
```

Notice that to get the O(n) runtime, we are careful to only recurse once. We then check the returned success code P1 to figure out what to do next. The first "ok" clause says that if the input list starts with Z and the recursive call didn't delete any later copy of Z, then we should delete that first Z. In fact, the cut says that we MUST do so in this case. The second "ok" clause says that in all other cases, we should just return without any further deletion, preserving the list's first element X, and returning whatever success code (0 or 1) was returned by the recursive call.

Here's a simpler approach:

```
deletelastmaybe(Z, [], [], 0).
```

```
deletelastmaybe(Z,[X|Xs],[X|Ys],1) :- deletelastmaybe(Z,Xs,Ys,1), !.
deletelastmaybe(Z,[Z|Xs],Xs,1) :- !.
deletelastmaybe(Z,Xs,Xs,0).
```

This manages also to recurse once, by taking advantage of the fact that if deletelastmaybe doesn't manage to delete Z from Xs, then it doesn't change Xs at all. The above code can be read like this:

- Base case.
- If recursion manages to delete a later copy of Z, go with that.
- Otherwise, if the input list starts with Z, then delete that Z.
- Otherwise, we conclude there are no copies of Z anywhere in the input list, so return the input list with a code saying we haven't deleted anything.

Here's another approach from one of you (slightly edited);

```
deletelast(Z, [Z|Xs], Ys, P)
  :- !, deletelast(Z, Xs, Ys1, P1), ok(Z, P1, P, Ys1, Ys).
deletelast(Z, [X|Xs], [X|Ys], P) :- deletelast(Z,Xs,Ys,P).
ok(Z, 1, 1, Ys1, [Z|Ys1]).
ok(Z, 0, 1, Ys1, Ys1).
```