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

1. a. B >= A, C >= A

   b. B + C >= A, A >= B, A >= C

   c. r + M * (1-p) >= 20

      where M is large enough that r+M will always be >= 20 anyway
      even if r is very negative;
      that is, M is an upper bound on 20-r

   d. s <= u, t <= u, s <= v, t <= v

   e. x <= 10 + s   (where s is a slack variable)
      cost == 3*x + 2*s

      The question assumed that you are trying to minimize cost; it should have made this specific. In that case, s will be kept down to x-10 and the cost function will be accurate.

      Note that this is an example of minimizing a piecewise linear convex cost function ("diseconomies of scale") as shown in class.

      Another solution is
          cost >= 3*x
          cost >= 3*x + 2*(x-10)
      which creates the convex feasible region simply by giving two hyperplanes (lines). If x <= 10, then the first constraint is active and the second constraint is redundant. If x >= 10, it's the other way around.

2. a. Mixed integer programming (i.e., linear programming where some variables are forced to be integers; the constraints and objective are linear).

      As noted in the problem, for two rectangles X and Y to avoid overlapping, X has to be fully to the left OR fully to the right OR fully above OR fully below Y. You need integer variables to handle this disjunctive condition (in fact, the problem is NP-complete).

      You also need real variables for the various rectangle positions. Finally you need a real "totalheight" variable for the total height. Your objective is to minimize totalheight (a very simple linear objective).

      Conditioned on the values of the integer variables, all of the constraints are linear inequalities (enforcing the geometric relations required by the integer variables and by the totalheight variable).

   b. Now the objective function becomes quadratic (height * width), while the constraints are still linear. So this is a quadratic programming problem with some integer variables, properly called a quadratic mixed integer programming problem.

      (Unfortunately this quadratic objective height*width is not convex anywhere, not even in the sensible region where height and width are positive ...)

   c. Mixed integer programming, just as in part a. The only difference is that you need additional binary variables to

encode which boxes have been turned on their sides.

d. This is a quadratically constrained quadratic program. In other words, the objective is quadratic (as in b.) and so are the constraints. The constraints are now very simple, like this:

```
set I := {1..n};   # indices of the circles
param r[I] := ...  # radii of the n circles (given)
var x[I];          # centers of the n circles (to be determined)
var y[I];
var totalheight;

minimize cost: totalheight;

# distance between centers of any two distinct circles is at least
# the sum of their radii
subto no_overlap:
    forall <i,j> in I*I with i != j:(x[i]-x[j])**2 + (y[i]-y[j])**2 >= ⸗
(r[i]+r[j])**2;

# top of each circle must be below height
subto keep_low: forall <i> in I: y[i]+r[i] <= totalheight;
```

Of course, SCIP is not willing to solve this one because it can't handle QCQP.

(In fact the simplicity of this program is deceptive, however, because quadratic constraints are very powerful, even when limited to be convex. In particular, a quadratic constraint z(z-1)==0 would force z to be 0 or 1, so quadratic constraints are at least as powerful as integrality constraints.)

e. This question asked you to reduce one problem to another.

SOLUTION 1. Because the quadratic objective in part b. is so simple, simply picking a fixed value for width reduces it to a linear objective as in part a. Thus, we can try different widths, calling the MIP solver as in part a. for each width separately.

As you increase the allowed width, the optimal height found by the solver will decrease (not gradually, but in jumps). If you are content to try many different widths, then you can find the width that minimizes the product (width)*(optimal height). (Such one-dimensional minimization problems can be addressed reasonably by brute force or by various techniques including simulated annealing, even when they have local minima, as here.)

SOLUTION 2 (DOESN'T WORK). A few of you suggested that you could call the solver to minimize height (for width 100?) and then again to minimize width (for what height?), but you didn't explain how you could combine those two results into something that would minimize width*height!

SOLUTION 3. Another possible solution is to approximate this quadratic objective with a piecewise linear objective. We were able to do this in class using only LP, since in our example there, each of the nonlinear objective terms depended on only a single variable (this is called a "separable" objective function).

The case here is more complicated because the objective is not separable: it depends on two variables. (Thus, the cost per unit of slack in one variable is not constant but depends on

another variable.) However, it can still be done.

In fact, if the cost function were convex, we could generalize the LP construction from class, approximating the cost function using a collection of hyperplanes that were tangent to it at various selected points (x0,y0). (We would minimize the approximate cost subject to the constraint that it is above each of these hyperplanes individually. Each hyperplane gives a global lower bound on the true cost function, thanks to convexity, and that lower bound is tight near the selected point (x0,y0), thanks to tangency. Specifically, for a cost function f(x,y), the following constrains the approximate cost to be above the hyperplane that is tangent at (x0,y0):

```
cost >= f(x0,y0) + xslope*(x-x0) + slope*(y-y0)
```

where f1 and f2 are the partial derivatives of f(x,y) at (x0,y0) with respect to x and y, respectively; this is just the start of the Taylor expansion about (x0,y0).)

Unfortunately the width*height cost function used here is NOT convex. But we can still fall back on integer variables, which let us do almost anything! (We already needed some other integer variables for part a, so why not use more?) Here's one way:

```
set Heights := {1..maxheight-1};
var hi[Heights] binary;   # hi[3] will be true if height > 3

subto setw: forall <h> in Heights: height <= h + maxheight*hi[h];

minimize approx_area: sum <h> in Heights: hi[h]*width;
# for example, if height=3.5, then hi[1],hi[2],hi[3] will
# all be 1, so approx_area will be 3*width.  This provides
# an incentive to keep both height and width small.  Of
# course we have other constraints requiring height and
# width to be big enough to fit all the rectangles.
```

That solution is tailored to this specific situation, but there are general methods for using mixed integer programming to approximate arbitrary nonlinear constraints and functions.

f. Here is some ZIMPL (see part a. for the basic idea):

```
set I := { 1..n };

param totalwidth := 100;
param width[I] := ...   # sizes of the n rectangles (given)
param height[I] := ...

var x[I];  # upper right corners of rectangles (>= 0; to
var y[I];  # be determined; will be printed by the decoder)

var totalheight;
minimize cost: totalheight;

# Make sure each rectangle individually fits in
# the width*height space.
subto right:  forall <i> in I: x[i] <= totalwidth;
subto left:   forall <i> in I: x[i] >= width[i];
subto top:    forall <i> in I: y[i] <= totalheight;
subto bottom: forall <i> in I: y[i] >= height[i];

# Now prevent overlap.  This is a disjunctive requirement,
# for which we need integer variables!
```

```
            var left[I*I] binary;  # left[i,j]==1 if i is REQUIRED to be left of j
            var below[I*I] binary; # below[i,j]==1 if i is REQUIRED to be below j

            subto no_overlap:
                forall <i,j> in I*I with i != j: left[i,j] + left[j,i] + below[i,j] + ↙
below[j,i] >= 1;
            # for any two distinct rectangles, we must force
            # them to be ordered horizontally or vertically or both

            subto horiz_order:
                forall <i,j> in I*I: (x[j] - x[i]) >= width[j] + totalwidth*(1-left[i,j]);
            # rectangle i must be fully to the left of rectangle j,
            # unless left[i,j] is false, in which case we allow
                # sufficient slack that this constraint goes away

            param maxheight := sum <i> in I: height[i];
            subto vert_order:
                forall <i,j> in I*I: (y[j] - y[i]) >= height[j] + maxheight*(1-below[i,j]);
            # similarly.  Note how we computed maxheight to provide a
                # sufficient amount of slack.
```

g. An easy option is to solve the problem once with

```
    minimize cost: totalheight;
```

Let h be the cost that was thus achieved. Now solve the problem again with

```
    minimize cost: sum <i> in I: y[i];
        subto oldcost: totalheight <= h;
```

A disadvantage of this solution is that it requires calling the solver twice, so the combined objective is not really part of the model. There is not a single linear function that expresses the quality of the rectangle-packing. This might make it difficult to build a more complex model (perhaps over more variables) where the quality of the rectangle-packing was just one term in the more complex objective.

An alternative is to express the single combined objective by changing the line

```
    minimize cost: totalheight;
```

to

```
    minimize cost: totalheight + epsilon*(sum <i> in I: y[i]);
```

This combined objective says "try to keep totalheight small and ALSO keep all the blocks low."

The value of epsilon gives the relative importance of these two objectives when they're in conflict. The difficulty here is in setting epsilon small enough so that our first priority is to minimize totalheight and the second term is only used to break ties.

If all of the block heights are integers, then setting epsilon is not hard. The optimal totalheight will also be an integer (the height of the tallest stack). So we should just set epsilon such that the second term can only vary by < 1 and therefore can't justify increasing totalheight to the next integer. This can be done by setting epsilon < 1/(n*maxheight), for example.

If the block heights are rationals or fixed-point reals, then they can be transformed to integers before the problem is solved, by multiplying them all by an approriate large integer k. Then the previous paragraph can be used.

3. (a) As a warm-up, suppose we wanted to require "Angela" and "Dan" to be close together, e.g., |pos["Angela"]-pos["Dan"]| <= 3. This is enforced by the pair of constraints

```
pos["Angela"] - pos["Dan"] <= 3;
pos["Dan"] - pos["Angela"] <= 3;
```

For our situation, we ask all <x,y> pairs to be as close together as possible, but we only penalize violations of that constraint (i.e., slack) to the extent that <x,y> have a meaningful adjacency:

```
subto getclose:
    forall <x,y> in Names*Names: pos[y] - pos[x] <= 1 + slack[x,y];

minimize cost: sum <x,y> in Names*Names: slack[x,y]*a[x,y];
```

Notice that the "forall" will generate a constraint for both <"Angela","Dan"> and another for <"Dan","Angela">, just as in our warm-up. And notice that at least one of these constraints won't need any slack in the solution, since one of "Angela" and "Dan" must come first in the ordering, and therefore one of pos["Angela"]-pos["Dan"] and pos["Dan"]-pos["Angela"] must be <= 0.

We could require that all positions are in 0,1,2,...n-1:

```
subto bounds: forall <x> in Names: pos[x] <= n-1;
```

I'll point out later why that's not really necessary, although conceivably it is helpful.

There's one hard constraint we really do need, though. So far, our objective is minimized by putting all the names on top of one other, with all the pos variables equal! Obviously, that would create lots of meaningful adjacencies :-), but it wouldn't be very readable. To decode into an ordered list, we need

```
forall <x,y> in Names*Names: pos[x] != pos[y];
```

But we can't write !=, so we need to encode this explicitly. The != indicates a disjunction: x can't be on top of y, but must be either before or after. We handle this disjunction in ILP with a binary variable:

```
var precedes[Names*Names] binary;
```

When precedes[x,y] is true, we'll impose the constraint

```
pos[y]-pos[x] >= 1, and when it's false, we'll impose
pos[x]-pos[y] >= 1.  (Thus, the difference can be any
```

positive or negative integer, but not 0.) Like this:

```
subto before:
    forall <x,y> in Names*Names with x != y: pos[y]-pos[x] + (1-precedes[x,y])*n >= 1;
subto after:
    forall <x,y> in Names*Names with x != y: pos[x]-pos[y] + precedes[x,y]*n >= 1;
```

Notice that n is large enough so that the "before" constraint has no constraining effect if precedes[x,y]==0. In that case, it just says that even if x isn't before y, it should be at most n-1 positions after y, which is always true.

Notice that we were careful to avoid requiring "Angela" to precede "Angela" or vice-versa.

Notice that our encoding is a little bit wasteful since we have separate variables for precedes["Angela","Dan"] and precedes["Dan","Angela"]. But it'll work fine.

As a final note, we could legitimately drop the constraints that the pos variables are in the range [0,n-1] -- and even the constraint that they are integers! Thanks to the binary precedes variables), the before/after constraints require distinct names to be separated by at least 1. Once we have that, the cost function will want them as close together as possible, so the optimal solution will separate adjacent names by 1. That is, the solver will prefer to encode the ordering "Dan","Angela","Brady","James"... by giving those names consecutive positions of 0,1,2,3 -- or equivalently 2,3,4,5 or even 2.2, 3.2, 4.2, 5.2 -- rather than farther-apart positions like 2.2, 3.3, 4.4, 5.5 or 2,5,8,12. It is true that there might be some blocks of names that can float apart arbitrarily far because there are no meaningful adjacencies between them, but we can still decode by sorting the pos variables in the solution. Or if we want to enforce that they're consecutive and start at zero, we can just add a slight bias toward having all names close together and toward the left:

```
minimize
    (sum <x,y> in Names*Names: slack[x,y]*a[x,y])
    + (sum <x,y> in Names*Names: slack[x,y]*epsilon)
    + (sum <x> in Names: pos[x]*epsilon);
```

where epsilon > 0 is set small enough that it can't change the ordering.

(b) This is pretty straightforward. Just convert the previous constraints:

```
var precedes[Names*Names] binary;

minimize cost: sum <x,y> in Names*Names: slack[x,y]*a[x,y];

subto getclose:
    forall <x,y> in Names*Names: dist[x,y] <= 1 + slack[x,y];

subto before:
    forall <x,y> in Names*Names with x != y: dist[x,y] + (1-precedes[x,y])*n >= 1;

subto after:
    forall <x,y> in Names*Names with x != y: dist[y,x] +    precedes[x,y] *n >= 1;
```

However, notice that these constraints are compatible with saying that dist[x,y]==1 for all x,y pairs! That solution would give us 0 cost, but it obviously doesn't give a total ordering.

This highlights that we must also require that the inter-position distances could be derived as the differences between some set of positions pos[x], pos[y], etc. on the real line. The following certainly holds if that is true:

```
    subto additive:
        forall <x,y,z> in Names*Names*Names: dist[x,y]+dist[y,z] == dist[x,z]
```

That's enough to reconstruct the positions in a consistent way, up to a constant. Here's one way of thinking about it: For any given x, knowing all the values like dist[x,y] and dist[x,z] tells us where all of the names are relative to x, and therefore where they are relative to one another. The "additive" axiom ensures that the positions we'd reconstruct by distance from x are compatible with the ones we'd reconstruct by distance from y.

As a further check that our constraints are enough, let's directly check that they do give us a total ordering of the names (implying that we can decode the output into a list).

Remember that a total ordering must be reflexive, antisymmetric, transitive, and trichotomous. In the case of the Linear Ordering Problem, we were working directly with the BOOLEAN variables precedes[x,y], and we had to enforce the first three of those constraints explicitly. But here, we get them from just the single "additive" constraint over the NUMERIC variables dist[x,y].

TRANSITIVITY: We want to show that if x precedes y and y precedes z, then x precedes z. In other words, that if dist[x,y] > 0 and dist[y,z] > 0, then dist[x,z] > 0. That is a consequence of "additive."

REFLEXIVITY: A special case of "additive"
```
    dist[x,x]+dist[x,x] == dist[x,x]
```
which implies dist[x,x] == 0.

ANTISYMMETRY: Another special case of "additive" is
```
    dist[x,y]+dist[y,x] == dist[x,x]
```
which implies dist[x,y] == -dist[y,x].

TRICHOTOMY: Our "before"/"after" constraints ensure that for any two different x,y, either dist[x,y] > 0 or dist[y,x] > 0.

Again, you could let the dist variables be real numbers if you wanted, as long as the precedes variables are binary.

(c) You may think that the two encodings are basically the same, modulo a linear transform, and it was good to say so. But there are two issues that might affect efficiency, and we hoped you'd call attention to them.

First, the position-based encoding is smaller because it has O(n) variables and O(n^2) constraints, whereas the distance-based encoding has O(n^2) variables and O(n^3) constraints, which will potentially slow things down, although presolving may be able to eliminate many of them.

Second, the two encodings have different integer variables. One has pos and precedes; the other has dist and precedes. This is potentially important because the branch-and-bound tree will be able to branch on these variables.

(It's also possible to write a version with pos AND dist and preceds as integers; or to relax the integrality of pos or dist as noted above.)

It is difficult to think about the effect of the extra

variables on branch and bound. But first let's think about doing the problem by hand, on index cards. You'd probably start out by arranging subgroups like Angela and her friends, and then positioning the subgroups with respect to one another. That corresponds to picking different distances, not picking different absolute positions.

From that point of view, you might not want to branch on the position variables. It's hard to derive good bounds on pos["Angela"], because there are lots of pretty good solutions where Angela is at different positions in the list and yet manages to be close to her colleagues. So if you're branching on position variables, you will not be able to prune many nodes. Putting Angela at position 25 and putting her at position 1259 will give LP relaxations of about equal quality, so you won't be able to prune either node. And there are certainly a lot of positions to try!

By contrast, branching on distance variables might be helpful. In that case, a node of the branch-and-bound tree may have fixed some local configuration of Angela and her colleagues, without yet committing to where that configuration will fall in the overall list.

Finally, it is possible to branch on precedes variables. Note that when precedes variables are not constrained to be 0 or 1 (by branching on precedes, pos, or dist), they are essentially no help in the LP relaxation. Two names <x,y> will be able to land on top of each other as long as precedes[x,y] and precedes[y,x] are in the range [1/n, 1-1/n].

(d) i. Yes. (Look at the graph)
ii. If it weren't, we'd need more integer variables to divide it into convex regions.
iii. We already have

```
subto getclose:
    forall <x,y> in Names*Names:
        dist[x,y] <= 1 + slack[x,y];
```

Let's add

```
subto getprettyclose:
    forall <x,y> in Names*Names:
        dist[x,y] <= 2 + slack2[x,y];
```

Now change our objective

```
minimize cost:
    sum <x,y> in Names*Names:
        slack[x,y]*a[x,y];
```

to

```
minimize cost:
    sum <x,y> in Names*Names:
        slack[x,y]*a[x,y]
        + slack2[x,y]*100;
```

(e) i. Quadratic programming -- or for full credit, mixed integer quadratic programming.

The objective function is now quadratic since it has a term slack2[x,y] ^ 2.
There are still no quadratic constraints, but there are

integrality constraints.

ii. Yes. In fact, the objective is still convex, so we don't need any new binary variables. We can approximate the objective with a piecewise linear function by continuing what we did in part (d), e.g., having additional constraints and slack variables for dist[x,y] <= 3, <= 4, etc. Since the distances are integer, this approximation is actually exact at all feasible points anyway. However, it does require a lot of new constraints and slack variables.

(f) Suppose precedes[x,y], which already ensures dist[x,y] >= 1. If also b[x,y]=1, then we want to say that dist[x,y] >= 2.

```
subto badadj:
    forall <x,y> in Names*Names:
        dist[x,y] + n*(1-b[x,y]) + n*(1-precedes[x,y]) >= 2.
```

Remark: We could make this a soft constraint by adding slack, with a slack penalty in the objective.