Modeling and Solving Nontraditional Optimization Problems

Session 3a: Discrete Models

Robert Fourer

Industrial Engineering & Management Sciences
Northwestern University

AMPL Optimization LLC

4er@northwestern.edu — 4er@ampl.com

Chiang Mai University International Conference
Workshop
Chiang Mai, Thailand — 4-5 January 2011

Session 3a: Discrete Models

Focus

- Modeling language support of discrete optimization

Topics

- Current difficulties
- Nontraditional extensions

When is a Model "Discrete"?

Feasible region is not

- convex
- smooth
- connected

Objective is not

- convex
- differentiable
- continuous

Problem cannot be solved by

- linear programming methods (or extensions)
- derivative-based methods

Challenges to Modeling Systems

Ideal

- Express models as people think of them
- Automatically convert to forms required by solvers

Reality

- Require formulation in terms of integer variables
  - with a few exceptions as previously seen
- Support only local optimization of smooth functions
- Allow formulations that cannot be solved
  - rejected by solvers
  - attempted by solvers without success
  - incorrectly solved

Conversion Confusion, Ex 1

I am trying to solve the problem below, which has some non-linear components to it. However, it appears to me that it is quite a simple problem to solve (computationally) if I use an iterative algorithm.

But I am wondering if anybody can help me formulate the problem such that it can be solved by lpsolve, if possible or definitely say that it cannot be done with lpsolve?

1. Given c_1 a constant, P_k another constant 2.
Given A lookup table of Price and Discounting coupon

   P1 -> c1
   P2 -> c2
   P3 -> c3
   ...
   Pn -> cn

   This could potentially be modeled by SOS2.

Conversion Confusion, Ex 1 *(cont'd)*

3. Given A non-linear function

$$P = (k1 + x\ b1)/\ c + (k2 + x\ b2)/\ c^2 + (k3 + x\ b3)/\ c^3 + \ldots + (kn + x\ bn)/\ c^n$$

   where, P is the price, c is the discounting coupon, P can lie anywhere on the linearly interpolated curve given by the lookup table in 2.

   All other ki's and bi's are known.

   x is the unknown and needs to be solved for.

   Given a value (Pi, ci) on the linearly interpolated curve in 2, it is trivial to solve for x (will reduce to a very simple linear equation)

4. Constraint

   $x >= c\_l$

5. Objective

   $max\ (\ P + (x - c\_l) * P\_k\ )$

   where P is the solution to the nonlinear function in 3 for some c and x, subject to the price-coupon curve in 2, and constraint 4.

Conversion Confusion, Ex 1 *(cont'd)*

> If I iterate over the points of the piecewise linear curve in 2 (by discretising the linear curve at some precision), it easy to solve for x at each point (Pi, ci) on the interpolated curve; discard those x which don't satisfy constraint in 4 and evaluate objective in 5 in each case and take the best solution.
> This will work and should be quite fast too. But I am wondering if we can model this in a clever way so that it can be solve by lpsolve.
>
> Any thoughts ?

Conversion Confusion, Ex 2

```
I have a problem need to add a such kind of constraint:
   Max[ sum(Pi * Hi) ];  i is from 1 to 24;
   which Pi are constant and Hi are need to be optimized
Bound is -180 <= Hi <= 270
One of constraints is
   sum(Ci) = 0; here Ci = Hi if Hi > 0 and Ci = Hi/1.38 if Hi <0
is it possiable to solve this kind of problem with lp_solve?
and how to setup the constraint?
```

Conversion Confusion, Ex 3

I need to solve the following optimization problem:

Minimize -|x1|-|x2|

subject to

x1-x2=3

Do you know how to transform it to standard linear program:

In some textbooks I see in the case x1, and x2 are free in the constrain set as in this problem, we set x1=u1-v1, and x2=u2-v2.

However how to present the objective function with absolute values of x1 and x2 using the auxiliary variables?

I am really confused. Can you please help?

Conversion Confusion, Ex 3 *(cont'd)*

```
If the problem was to *minimize* |x1| + |x2|, then writing |x1| = u1 +
v1 and |x2| = u2 + v2 will work---and it is what the textbooks
suggest. It works because for any value of x1, u1 + v1 will be
minimized by having at least one of u1 = 0 or v1 = 0. For example, if
x1 = 2.76, then to make u1 + v1 as small as possible, subject to u1 -
v1 = 2.76, we take u1 = 2.76 and v1 = 0; hence u1 + v1 = 2.76. But, if
x1 = -2.76, take u1 = 0 and v1 = 2.76; we have u1 - v1 = -2.76, but u1
+ v1 = +2.76 = |-2.76|. To get it to work, it is CRUCIAL that you be
minimizing a positive weighted sum of absolute values. If fails if you
are maximizing (even if one places bounds on the xi that make the
problem finite---unlike your example); you should think about why this
is the case, perhaps by constructing for yourself a little one- or two-
variable example.
```

Conversion Confusion, Ex 3 *(cont'd)*

Thanks, my professor also answered me the same as you do.

Do you still remember the textbook which says about the technique you mentioned (for the case of absolute variables). I read some textbooks and they are listed out some limitted cases to convert to standard linear program:

1. Max to min
2. Greater or equal, Less or Equal cases for contrains
   Ax>=b or Ax<=b
3. Free variables.

Sometimes I meet some more complicated cases such as the case I asked you and I don't know how to solve.

Conversion Confusion, Ex 3 *(cont'd)*

I think something that he can do, though, to solve his "maximize |x1| + |x2|" problem, is repeatedly solve feasibility problems with a lower bound on |x1| + |x2|, and find the maximum value of that lower bound for which the problem remains feasible. I.e. you would solve:

```
minimize q1 + q2
s.t.
q1 >=  x1
q1 >= -x1
q2 >=  x2
q2 >= -x2
x1 - x2 = 3
q1 + q2 >= K
```

repeatedly for different values of K. Find a low value of K for which the problem is feasible and a high value of K for which the problem is infeasible and then use bisection.

Conversion Confusion, Ex 3 *(cont'd)*

```
Nevermind, on second thought I see that this idea will not work. Once
the q variables are used in additional constraints (in this case q1 +
q2 >= K) there is no longer a guarantee that at optimum they will be
tight with one of their corresponding x variable expressions. So the
optimization can give a nonsense answer.
```

Conversion Confusion, Ex 3 *(cont'd)*

```
OK, so you suggested |x1| is replaced by u1+v1, and |x2|=u2+v2.
I am still confused in the constrain equations, what  x1 and x2
would be?
```

```
They should just be x1 and x2, although you *could* eliminate them and
write u1-v1 instead of x1 and u2-v2 instead of x2. For example, the
problem

   min |x1| + |x2|,
   subject to 3*x1 - 2*x2 = 7, x1 >=-5, x2 free

can be written as

   min u1 + v1 + u2 + v2
   subject to u1 - v1 = x1, u2 - v2 = x2, 3*x1 - 2*x2 = 7,
              v2 <= 5, u1,u2,v1,v2 >= 0.

You could, of course, re-write the constraints as 3*(u1-v1)-2*(u2-v2)
= 7, v2 <= 5, ui, vi >= 0, and drop all reference to x1 and x2. After
solving the u,v problem, you would just _calculate_ x1 and x2 as ...
```

Conversion Confusion *(abs, ceil)*

```
Could you tell me how can I calculate

minimize obj:
max({i in 1..4}: abs(x[i]));

or

minimize obj:
max({i in 1..4}: min(x[i]);
```

```
Following is my question

ampl: expand MultiplePath;

subject to MultiplePath[1,13]:
   ceil(df[1,13,1]) + ceil(df[1,13,2]) <= 2;

Above is the output of the expand command the value of variable df (my
decision variable) is between 0 and 1. Ceil function does not work and
i want to count all df variables which are greater than zero so than i
can restrict to choose only two paths. Any alternative solution will
also be appreciated.
```

Conversion Confusion *(if)*

how can i linearize this constraint because CPLEX said that my model contains nonquadratic non-linear constraint

this is the part of the model

```
var TOTCOST {c in CAMPAIGN} = sum{(i,j,k) in ROUTES}
   cost[i,j,k] * insertion[i,j,k,c];

var COUNT {c in CAMPAIGN} = sum {(i,j,k) in ROUTES} insertion[i,j,k,c];

var DISQTY1 {c in CAMPAIGN} =
   if COUNT[c] >= limit1 then TOTCOST[c] * disc1;

var DISQTY2 {c in CAMPAIGN} =
   if COUNT[c] >= limit2 then TOTCOST[c] * disc2 else DISQTY1[c];
```

Conversion Confusion *(if, or)*

```
I have been trying to write a stepwise function in AMPL
but I have not been able to do so:

     fc[wh] = 100 if x[wh] <=5
            300 if 6 <= x[wh] <=10
            400 if 11 <= x[wh]

where fc and x are variables.
```

```
I have a set of nonlinear equations to be solved, and variables
are binary.  Even I have an xor operator in the equations.
How can I implement it and which solver is suitable for it?
```

```
I have a variable with following bounds:

   x > 0.8 or x < 0.2.

I appreciate any suggestions how I can formulate above constraint
in lp-solve.
```

Conversion Confusion *(if, or)*

I'm a recent IE grad with just one grad level IE course under my belt. ... I'm just trying to find the x,y coordinate that minimizes the number of individuals in one of those groups (we'll call it GroupA) that are outside the area if a circle centered at (XGrpA, YGrpA) with a radius of Ra. ...

```
minimize Moves: sum{emp in GROUPA}
   (if Sqrt((XEmpA[emp] - XGrpA)^2 +
            (YEmpA[emp] - YGrpA)^2) > Ra then 1 else 0)
```

Is there some documentation on when you can and cannot use the if-then statements in AMPL (looked through the related forum posts but still a bit confused on this)?

... is there a way to write a simple "or" statement in AMPL like in Java or C++?

Alternatives for Discrete Modeling

A simple example

- Job sequencing with setups

A survey of possibilities

- Logical conditions
- Cardinality restrictions
- Matching
- Assignment
- Selection
- Sequencing

Example: Job Sequencing with Setups

Given

- **A set of jobs,** with
  production times, due times and earliness penalties
- **One machine** that processes one job at a time
- **Setup costs and times** between jobs
- **Precedence relations** between certain jobs

Choose

- A sequence for the jobs

Minimizing

- Setup costs plus earliness penalties

C. Jordan & A. Drexl, A Comparison of Constraint and Mixed Integer Programming Solvers for Batch Sequencing with Sequence Dependent Setups. *ORSA Journal on Computing* **7** (1995) 160–165.

Example: Variables and Costs

Either way

- `ComplTime[j]` is the completion time of job `j`
- Earliness penalty is the sum over jobs `j` of

```
duePen[j] * (dueTime[j] - ComplTime[j])
```

Integer programming formulation

- `Seq[i,j]` equals 1 iff `i` immediately precedes `j`
- Setup cost is the sum over job pairs `(i,j)` of

```
setupCost[i,j] * Seq[i,j]
```

More natural formulation

- `JobForSlot[k]` is the job in the `k`th slot in sequence
- Setup cost is the sum over slots `k` of

```
setupCost[JobForSlot[k],JobForSlot[k+1]]
```

Example: Production Constraints

Integer programming formulation

For each job `i`, `ComplTime[i]` ≤ `dueTime[i]`

For each job pair `(i,j)`,

`ComplTime[i] + setupTime[i,j] + procTime[j] ≤`

`ComplTime[j] + BIG * (1 - Seq[i,j])`

More natural formulation

For each slot `k`,

```
ComplTime[JobForSlot[k]] = min (
  dueTime[JobForSlot[k]],
  ComplTime[JobForSlot[k+1]]
      - procTime[JobForSlot[k+1]]
      - setupTime[JobForSlot[k],JobForSlot[k+1]] )
```

Example: Sequencing Constraints

Integer programming formulation

For each job `i`,

```
sum {j in JOBS} Seq[i,j] = 1
```

For each job `i`,

```
sum {j in JOBS} Seq[j,i] = 1
```

More natural formulation

```
alldiff {k in SLOTS} JobForSlot[k]
```

General Principles

Variables

- Use extended operators and expressions to avoid defining new zero-one variables
- Define fewer variables having larger domains

Constraints

- Use structure (global) constraints to replace large collections of simpler constraints

traditional model

alternative model

Logical Conditions

Every job either precedes or follows every other job

```
subj to NoConflict12
  {i1 in JOBS, i2 in JOBS: ord(i1) < ord(i2)}:

    Start[i2] >= Start[i1] +
      setupTime[i1,i2] - BIG * (1 - Prec[i1,i2]);

subj to NoConflict21
  {i1 in JOBS, i2 in JOBS: ord(i1) < ord(i2)}:

    Start[i2] >= Start[i1] +
      setupTime[i2,i1] - BIG * Prec[i1,i2];
```

```
subj to NoConflict
  {i1 in JOBS, i2 in JOBS: ord(i1) < ord(i2)}:

   Start[i2] >= Start[i1] + setupTime[i1,i2] or
   Start[i1] >= Start[i2] + setupTime[i2,i1];
```

Logical Conditions *(cont'd)*

No one should feel isolated within an assigned group

```
subj to NoIso0 {(i1,i2) in TYPE, j in GROUP}:
  Assign[i1,i2,j] <= upperbnd[i1,i2,j] * Any[i1,i2,j];

subj to NoIso1a {(i1,i2) in TYPE, j in GROUP}:
  Assign[i1,i2,j] >= Any[i1,i2,j];

subj to NoIso1b {(i1,i2) in TYPE, j in GROUP}:
  Assign[i1,i2,j] +
    sum {ii1 in ADJ[i1]: (ii1,i2) in TYPE} Assign[ii1,i2,j]
      >= 2 * Any[i1,i2,j];
```

```
subj to NoIso {(i1,i2) in TYPE, j in GROUP}:
  not (Assign[i1,i2,j] = 1 and
    sum {ii1 in ADJ[i1]: (ii1,i2) in TYPE} Assign[ii1,i2,j] = 0);
```

Logical Conditions *(cont'd)*

No one should feel isolated within an assigned group

```
subj to NoIso0 {(i1,i2) in TYPE, j in GROUP}:

  Assign[i1,i2,j] <= upperbnd[i1,i2,j] * Any[i1,i2,j];

subj to NoIso1a {(i1,i2) in TYPE, j in GROUP}:

  Assign[i1,i2,j] >= Any[i1,i2,j];

subj to NoIso1b {(i1,i2) in TYPE, j in GROUP}:

  Assign[i1,i2,j] +
    sum {ii1 in ADJ[i1]: (ii1,i2) in TYPE} Assign[ii1,i2,j]
      >= 2 * Any[i1,i2,j];
```

```
subj to NoIso {(i1,i2) in TYPE, j in GROUP}:

  Assign[i1,i2,j] = 1 ==>

    sum {ii1 in ADJ[i1]: (ii1,i2) in TYPE} Assign[ii1,i2,j] >= 1;
```

Cardinality Restrictions

A warehouse may not serve too many customers

```
var Serve {WHSE,CUST} binary;

subj to UDef {i in WHSE, j in CUST, p in PROD}:
   sum {p in PROD} Trans[i,j,p] <= limit[i,j] * Serve[i,j];

subj to MaxServe {i in WHSE}: sum {j in CUST} Serve[i,j] <= mxsrv;
```

```
subj to MaxServe {i in WHSE}:
   count {j in CUST} (sum {p in PRD} Trans[i,j,p] > 0) <= mxsrv;
```

```
subj to MaxServe {i in WHSE}:
   atmost mxsrv {j in CUST} (sum {p in PRD} Trans[i,j,p] > 0);
```

Matching

Assign each job to a different machine

```
var Assign {JOBS,MACHINES} binary;

subj to OneJobPerMachine {k in MACHINES}:
  sum {j in JOBS} Assign[j,k] = 1;
```

```
var MachineforJob {JOBS} in MACHINES;

subj to OneJobPerMachine:
  alldiff {j in JOBS} MachineForJob[j];
```

Assignment

Assign a limited number of jobs to each machine

```
var Assign {JOBS,MACHINES} binary;

subj to CapacityOfMachine {k in MACHINES}:
  sum {j in JOBS} Assign[j,k] <= cap[k];
```

```
var MachineforJob {JOBS} in MACHINES;

subj to CapacityOfMachine {k in MACHINES}:
  numberof k in ({j in JOBS} MachineForJob[j]) <= cap[k];
```

Selection

Select locations to minimize fixed plus variable costs

```
var Serve {CLI,LOC} binary;
var Open {LOC} binary;

minimize TotalCost:
   sum {i in CLI, j in LOC} srvCost[i,j] * Serve[i,j] +
   bdgCost * sum {j in LOC} Open[j];

subject to OneEach {i in CLI}:
   sum {j in LOC} Serve[i,j] <= 1;

subject to OpenDefn {i in CLI, j in LOC}:
   Serve[i,j] <= Open[j];
```

```
var Serve {CLI} in LOC;
var Open {LOC} binary;

minimize TotalCost:
   sum {i in CLI} srvCost[i,Serve[i]] +
   bdgCost * sum {j in LOC} Open[j];

subject to OpenDefn {i in CLI}: Open[Serve[i]] = 1;
```

Sequencing

Choose job completion times consistent with setup and due times

```
subj to JobDueTime {j in JOBS}:
  ComplTime[j] <= dueTime[j];

subj to JobSetupTime {j1 in JOBS, j2 in JOBS: j1 <> j2}:
   ComplTime[j1] + setupTime[j1,j2] + procTime[j] <=
   ComplTime[j2] + BIG * (1 - Seq[j1,j2]);
```

```
subj to ComplTimeDefn {k in 1..nSlots}:
  ComplTime[JobForSlot[k]] =
    min( dueTime[JobForSlot[k]],
         ComplTime[JobForSlot[k+1]]
           - procTime[JobForSlot[k+1]]
           - setupTime[JobForSlot[k],JobForSlot[k+1]] )
```

Knapsack

Choose the most valuable subset of objects that fit

```
var In {OBJECTS} binary;

maximize Total_Value:
  sum {i in OBJECTS} value[i] * In[i];

subj to Weight_Limit:
  sum {i in OBJECTS} weight[i] * In[i] <= cap;
```

```
var KNAPSACK within OBJECTS;

maximize Total_Value:
  sum {i in KNAPSACK} value[i];

subject to Weight_Limit:
  sum {i in KNAPSACK} weight[i] <= cap;
```

Assignment *(again)*

Assign students to suitable project groups

```
var Assign {STU,PRJ} binary;

subject to Assign_Students {s in STU}:
  sum {p in PRJ} Assign[s,p] = 1;

subject to Assign_Projects {p in PRJ}:
  min_team[p] <= sum {s in STU} Assign[s,p] <= max_team[p];

subject to Enough_Cars {p in PRJ}:
  sum {s in STU} car[s] * Assign[s,p] >= cars_needed[p];
```

```
var MEM {p in PRJ} within STU;

subject to Assign_Students: alldisjoint {p in PRJ} MEM[p];
subject to All_Students: sum {p in PRJ} card(MEM[p]) = card(STU);

subject to Assign_Projects {p in PRJ}:
  min_team[p] <= card(MEM[p]) <= max_team[p];

subject to Enough_Cars {p in PRJ}:
  card(MEM[p] inter HAVE_CARS) >= cars_needed[p];
```

Remaining Issues

Which expressions are implemented?

- Many discrete operators in AMPL
  - **`and`**, **`or`**, **`not`**, **`if-then-else`**
  - **`count`**, **`atmost`**, **`atleast`**, **`numberof`**
  - **`alldiff`**
- Further support in various solver-specific languages

What solvers could handle them?

How could they be communicated to solvers?