Formulations and Reformulations in Integer Programming

Michael Trick

Tepper School of Business, Carnegie Mellon, Pittsburgh, PA USA, 15213
trick@cmu.edu*

Abstract. Creating good integer programming formulations had, as a basic axiom, the rule "Find formulations with tighter linear relaxations". This rule, while useful when using unsophisticated branch-and-bound codes,is insufficient when using state-of-the-art codes that understand and embed many of the obvious formulation improvements. As these optimization codes become more sophisticated it is important to have finer control over their operation. Modelers need to be even more creative in reformulating their integer programs in order to improve on the automatic reformulations of the optimization codes.

1 Introduction

Integer programming has shown itself to be an effective mechanism for solving a wide variety of difficult combinatorial optimization problems of practical interest. While no technique can solve every instance of such problems quickly, integer programming has been robust and effective enough to play a key role in solving problems in applications such as airline crew scheduling, combinatorial auction winner determination, telecommunications network design, sports scheduling and many other applications.

Despite the practical success of integer programming, initial forays into this area are often full of frustration: seemingly obvious formulations "don't work", leading to excessive computation time for even small instances. Success with integer programmming seems to be a hit-or-miss proposition, with more misses than hits.

In this note, I examine two problems of practical interest: a transportation design problem and a sports scheduling problem. We will show that key to the successful application of integer programming to these problems is the choice of formulation. In both cases, initial formulations lead to intractible instances, while "good" formulations can be solved very quickly with modern software. However, the "good" formulations have to be very creative, since modern software embeds most of the obvious formulation improvements. –

The general issue of formulations in integer programming has been little studied. Textbooks generally provide lots of examples in the hope that readers will

* Thanks to DASH Optimization who provided the XPRESS-MP software under their Academic Partner Program.

be able to find generalizations. One exception is the book by Williams [6], which does concentrate on formulations and provides some broad perspective on integer programming formulations. Otherwise the integer programming literature contains a vast number of formulations, many with computational experience, with few generalizations on what leads to a successful formulation. The few generalizations we have are so well understood that they are included in modern software (as we will see) to the extent that model formulations do not need to include the "improvements": the software will generate them itself.

So, integer programming formulations often "don't work", taking excessive time to find and prove optimal solutions, but modern software already includes some of the obvious improvements. What is a modeler to do?

By closely examining these two cases, I believe that there are general things to be learned. First, I think the integer programming paradigm where models are given by the variables, objective, and linear constraints can be greatly enhanced by learning from the constraint programming field whereby models are often given by higher-level constructs. As we will see, within integer programming, there is a huge difference between the linear constraint

$$4x_1 + 10x_2 + 7x_3 + 5x_4 + 8x_5 \leq 17$$

and what might be denoted the "knapsack" constraint

$$\texttt{knapsack}([4, 10, 7, 5, 8], x, 17)$$

with all the implications that come from our understanding of knapsack constraints. Constraint programmers understand this difference, while integer programmers tend to muddy up the distinction (or leave it to software to handle).

Second, there is still room to provide better formulations to software in the standard integer programming sense: formulations with better relaxations. It must be understood, however, that software is already pretty good at "tightening" formulations, so the modeler has to be quite creative to get beyond what the software can do. This leads to the interesting question of what can be embedded in software: in the race between modelers and software, will there always be a role for modelers or will software be able to include everything a modeler can think of?

Third, one area where modelers have a advantage is in the creation of problems with a huge number of constraints or variables. Such formulations can be very powerful, but are difficult for integer programming codes to generate since they involve the concept of a cut (or variable) generation algorithm, rather than generating the cuts or variables themselves. Given the power of such formulations, is there any mechanism for automatically generating these models, or will human modelers always be required to provide guidance here?

2 Example 1: Transportation Design

My first example is a transportation design problem that came from a consulting project I did a year or two ago. The company sends packages between pairs of

cities. The amount it sends is high volume (multiple trucks per day), so its trucks go simply between pairs (there are no complicated routing issues) in one-way trips. For a pair of cities (A,B), the company has a set of packages to be sent from A to B. Each package has a size, a time for which is available to load at A, and a time for which it is needed at B. Trucking firms have provided the company with a set of truck choices. Each "choice" consists of a truck of a particular size, leaving A at a particular time, and arriving at B at a particular time. Each choice has a cost. The goal of the company is to choose a set of trucks that can hold all of the packages and gets them to B on time. A package cannot be split among multiple trucks.

Naturally the real problem is more complex, with more cities, complicated routing, multiple capacity constraints, splittable packages, and other aspects, but this simplified model has most of the critical features.

The natural integer programming formulation for this has a set of binary (0-1) variables for the decision on whether to use a particular truck (indexed by i) and a binary variable for whether package j goes onto truck i. We handle the timing issues by an array `can_use(i,j)` which is 1 if truck i can handle package j (that is, j is available at A before i departs, and i arrives at B before j is required there). This results in the formulation in Figure 1 (written in the language Mosel [7]).

Don't worry if you are not familiar with Mosel: this is a straightforward integer programming formulation. Constraints (1) ensure that the total size of the packages assigned to a truck is no more than the capacity of the truck. Constraints (2) ensure that `x(j,i)` is 0 whenever `y(i)` is (`NUM_PACKAGE` is the number of packages in the instance). While (2) might look to be a strange formulation of the constraint, this is a "standard" integer programming approach to handling this requirement. Constraints (3) for every package to go on some truck. (4) and (5) enforce the integrality restrictions.

I will illustrate the effect of various formulations with a single 10 truck, 20 package instance (the real examples are at least an order of magnitude larger). The formulation above with this instance solved with XPRESS-MP [7] (Optimizer version 15.20.05) results in 11.2 seconds of computation time (3Gz, Intel/Windows machine, 2Gb memory, default settings), with 31,825 nodes in the branch-and-bound tree. This time is not extreme, but it is much larger than we would want with such a small instance.

Now, it is a fundamental tenet of integer programming that the key to a successful formulation is a "tight" linear relaxation. The linear relaxation of the above model replaces (4) and (5) with

```
forall (i in TRUCKS)
             y(i) <= 1                          ! (4')
forall (i in TRUCKS, j in PACKAGES)
             x(j,i) <= 1                        ! (5')
```

(Note that nonnegativity of the variables is assumed). This results in a linear program (the `x` and `y` variables can take on fractional values). A formulation

```
model "Transportation Planning"
uses "mmxprs"


declarations
TRUCKS = 1..10
PACKAGES = 1..20
capacity: array(TRUCKS) of real
size: array(PACKAGES) of real
cost: array(TRUCKS) of real
can_use: array(PACKAGES,TRUCKS) of real
x: array(PACKAGES,TRUCKS) of mpvar
y: array(TRUCKS) of mpvar
end-declarations

capacity:= [100,200,100,200,100,200,100,200,100,200]
size := [17,21,54,45,87,34,23,45,12,43,
 54,39,31,26,75,48,16,32,45,55]
cost := [1,1.8,1,1.8,1,1.8,1,1.8,1,1.8]
can_use:=[1,1,1,1,1,1,0,0,0,0,   1,1,1,1,0,0,0,0,0,0,
 1,1,1,1,1,1,1,1,0,0, 1,1,1,1,1,1,1,0,0,0,
 0,1,1,1,1,0,0,0,0,0, 0,1,1,1,1,1,1,0,0,0,
 0,0,1,1,1,1,1,1,1,1, 0,0,1,1,1,1,1,1,0,0,
 0,0,1,1,1,1,0,0,0,0, 0,0,0,1,1,1,1,1,1,0,
 0,0,0,1,1,1,1,0,0,0, 0,0,0,1,1,1,0,0,0,0,
 0,0,0,0,1,1,1,1,1,0, 0,0,0,0,1,1,1,1,0,0,
 0,0,0,0,1,1,1,1,1,1, 0,0,0,0,0,1,1,1,1,1,
 0,0,0,0,0,1,1,1,1,0, 0,0,0,0,0,0,1,1,1,1,
 0,0,0,0,0,0,0,1,1,1, 0,0,0,0,0,0,0,0,1,1]


Total := sum(i in TRUCKS) cost(i)*y(i)
forall(i in TRUCKS)
             sum(j in PACKAGES) size(j)*x(j,i) <= capacity(i)
                                                  ! (1) Packages fit
forall (i in TRUCKS)
             sum (j in PACKAGES) x(j,i) <= NUM_PACKAGE*y(i)
                                                  ! (2) use only
                                                  ! paid for trucks
forall (j in PACKAGES)
             sum(i in TRUCKS) can_use(j,i)*x(j,i) = 1
                                                  ! (3) every
                                                  ! package on truck
forall (i in TRUCKS)
             y(i) is_binary                       ! (4) no partial trucks
forall (i in TRUCKS, j in PACKAGES)
             x(j,i) is_binary                     ! (5) no package splitting

minimize(Total)
end-model
```

Fig. 1. Transportation formulation

with linear relaxation F_1 is tighter than another with relaxation F_2 if every fractional feasible solution to F_1 is also a fractional feasible solution to F_2 and the reverse is not true. Note that tightness is a property of the linear relaxation of a formulation.

Every integer programmer will look at the formulation given and immediately identify improvements. The main issue is in the constraints (2). These are well-known "weak" constraints. For example, it is straightforward to see that a package can be assigned to a truck whose corresponding `y` value is as small as $\frac{1}{\texttt{NUM\_PACKAGE}}$. We can "cut off" this sort of solution by replacing (2) with the constraints

```
forall (i in TRUCKS, j in PACKAGES) x(j,i) <= y(i)
                                          !(2') tighter formulation
```

Now, if `x(j,i) = 1` for a particular `i,j` then the corresponding `y(i)` must also be 1. The addition of these constraints leads to a tighter formulation. Note that the new formulation is quite a bit larger: instead of one constraint for every truck, we have a constraint for every (truck,package) pair. While this makes it slower to solve the linear program at each node of the branch-and-bound tree, the resulting decrease in size of the tree far outways this.

Further improvements can be had by replacing

```
forall(i in TRUCKS)
             sum(j in PACKAGES) size(j)*x(j,i) <= capacity(i)
                                               ! (1) Packages fit
```

with

```
forall(i in TRUCKS)
             sum(j in PACKAGES) size(j)*x(j,i) <= capacity(i)*y(i)
                                               ! (1') Packages fit
```

Again, depending on the exact coefficients, this can lead to a tighter formulation.

At this point, integer programmers step back, look self-satisfied, and move on to other problems.

Unfortunately, when put into XPRESS-MP (other sophisticated codes will work similarly), the results are not what was expected. The time for our instance goes *up*, doubling to 22.1 seconds with 50,631 nodes in the branch-and-bound tree.

What has happened? The primary point is that the relaxation solved by XPRESS-MP or any other top-quality code is not the same as the naive relaxation. The code already has the ability to identify "obvious" tightenings. In the case of constraints (2), there is a technique during preprocessing that sets each binary variable to 1 and determines any variable fixing that might occur (exactly as would happen with constraint propagation in constraint programming). If variable y must be 1 once x is 1, then the constraint $y \geq x$ can be added. This

will generate constraints (2') from (2) automatically. From a modeler's point of view, there is no need to add (2'): that "trick" is already known to the software.

Now, if the code is unsophisticated, it is important to add tightenings such as 2'. Turning off preprocessing and cut generation from XPRESS-MP leads to a formulation and solution code that takes 1851 seconds and 2.4 million nodes with 2'. Without 2', after the same 1851 seconds, branch-and-bound has taken 5 million nodes (since the linear program is smaller) but still has a duality gap with a lower bound of 1.22 and an upper bound (feasible solution) of 8.4 (8.2 is optimal). Time to optimality is measured in days.

It is somewhat mysterious why adding 2' to the sophisticated code actually slows things down for this instance. At this point, I have no better answer than the XPRESS-MP is a collection of heuristics: heuristics to find feasible solutions, heuristics to find tightening constraints, and heuristics to search the tree. Given the role of the heuristics, it is perhaps not surprising that a better formulation can sometimes lead to worse times once all of the preprocessing and cut identification is done.

This interaction of formulation with solution code is shown even stronger with the addition of the constraint:

```
sum(i in TRUCKS)
            capacity(i)*y(i) >= sum (j in PACKAGES)size(j)
                                        ! (6) Have sufficient capacity
```

This constraint says simply: the total capacity of the trucks chosen must be sufficient to handle the total size of the packages to be transported.

This constraint does not tighten the linear relaxation: it is a linear combination of previous constraints, so it cannot improve the relaxation. Standard IP formulation approaches would therefore not include the constraint.

Aardal [1] noted the surprising result that if you include this redundant constraint into the formulation, sophisticated codes solve instances much faster (they worked on a closely related location problem, where the timing aspects of the packages do not come into play).

For our instance, solution is instantaneous, and no branching is done: the problem is solved at the initial relaxation. How can adding a constraint that does not improve the relaxation affect the solution process to such an extreme extent?

Again, the key is that XPRESS-MP (or any other sophisticated code) does not solve the naive relaxation. In this case, the constraint (6) is recognized as a specially structured constraint, called a knapsack constraint. A tremendous amount is known about knapsack constraints (they form the basis for the groundbreaking work of Crowder, Johnson and Padberg [3] who used an understanding of knapsack constraints to solve general integer programs, a fundamental breakthrough in computational integer programming), and that knowledge is embedded in current codes. In particular, a set of constraints called *cover inequalities* are known to provide a much tighter formulation than just the linear knapsack

constraint alone. The constraints are added "automatically" by XPRESS-MP, resulting in an extremely tight formulation that is solved without branching.

This is an example of an interaction between the human modeler and the software. The modeler is needed to identify the knapsack inequality (at least current software does not automatically identify the redundant knapsack) but then the software is able to bring in all of its knowledge about knapsack constraints.

This development is not surprising (I believe) for constraint programmers. In constraint programming, it is common to add redundant constraints in order to improve propagation. In integer programming, however, it is unusual to add a constraint that doesn't improve the linear relaxation in order to take advantage of the automatic cut generation available in the software.

This would be more obvious to integer programmers if the solution codes offered more flexibility in the handling of cover and other inequalities. Currently, for every code I am aware of, you can do no more than set a level of aggressiveness in searching for cover inequalities (0=no inequalities, 1=1 round of search, etc.). It is not possible to identify some constraints as good prospects for finding cover constraints and others as poor areas. If integer programming codes were like constraint programming codes, then it would be possible to write (6) as something like

```
knapsack(capacity,y,'>',sum(j in PACKAGES) size(j))
                 with STRONGCUTS
```

or

```
knapsack(capacity,y,'>',sum(j in PACKAGES) size(j))
                 with FASTCUTS
```

or

```
knapsack(capacity,y,'>',sum(j in PACKAGES) size(j))
                 with NOCUTS
```

where the '>' denotes a "$\geq$" knapsack and STRONGCUTS, FASTCUTS, and NOCUTS tell the optimizer which cut generation routine to us. This will guide software in the amount of cuts to generate for this particular constraint, rather than the current approach of setting the generation for all constraints at once.

To review for this problem, a naive software implementation of branch-and-bound for a simple formulation doesn't work: solutions take hours or days. Tightening the formulation in the traditional sense only works for simple codes: sophisticated codes already include standard tightening. The best formulation requires understanding the capabilities of the software and adding a seemingly irrelevant constraint.

3 Example 2: Sports Scheduling

For our second problem, I will discuss some experiments on the Traveling Tournament Problem. The Traveling Tournament Problem (TTP), introduced by

Easton, Nemhauser, and Trick [4], is a simplification of a sports scheduling problem that arose in the scheduling of Major League Baseball (MLB). The requirements for MLB take many pages to describe, but the key aspects of a "good" MLB schedule is flow (the number of consecutive home or away series a team plays) and distance traveled (how far teams must fly in the schedule). Additional "real world" constraints include stadium availability, the scheduling of key rivals, holiday requirements and much more. The TTP ignores most of these requirements and concentrates on flow and distance.

Given n teams with n even, a double round robin tournament is a set of games in which every team plays every other team exactly once at home and once away. A game is specified by an ordered pair of opponents. Exactly $2(n-1)$ slots or time periods are required to play a double round robin tournament. Distances between team sites are given by an n by n distance matrix D. Each team begins at its home site and travels to play its games at the chosen venues. Each team then returns (if necessary) to its home base at the end of the schedule.

Consecutive away games for a team constitute a road trip; consecutive home games are a home stand. The length of a road trip or home stand is the number of opponents played (not the travel distance).

The Traveling Tournament Problem is defined as:

Input: n, the number of teams; D an n by n integer distance matrix; L, U integer parameters.

Output: A double round robin tournament on the n teams such that

- The length of every home stand and road trip is between L and U inclusive, and
- The total distance traveled by the teams is minimized.

The parameters L and U define the trade off between distance and pattern considerations. For $L = 1$ and $U = n - 1$, a team may take a trip equivalent to a traveling salesman tour. For small U, teams must return home often, so the distance traveled will increase. In this paper, we will concentrate on $L = 1$ and $U = 3$, which corresponds with the MLB ideal.

In addition, a "no-repeaters" constraint can be added: if team A plays at team B in slot t, then B does not play at A in slot $t + 1$.

Instances of the TTP seem very difficult, even for relatively small n. For $n = 4$, optimal solutions are relatively easy to find, but even $n = 6$ is nontrivial. The largest instances solved to optimality are at $n = 8$ (in contrast, MLB has two leagues: one with $n = 14$ and one with $n = 16$).

Many researchers have worked on heuristics for this problem, but there has been relatively little work on complete (or provably optimal) approaches.

The most direct formulation for this problem as an integer program defines a variable `plays(i,j,t)` which equals 1 if team i plays at team j in slot t. In this way, we can ensure a double-round robin structure with constraints like (where TEAMS is defined to be the range $1 \ldots n$ and SLOTS is $1 \ldots 2n-2$:

```
forall (i in TEAMS, t in SLOTS)
 plays(i,i,t) = 0 ! (1) no team plays itself

forall (i in TEAMS, t in SLOTS)
 sum(j in TEAMS) (plays(i,j,t)+plays(j,i,t)) = 1
                     ! (2) team i plays one team in each slot

forall (i,j in TEAMS | i <> j)
 sum (t in SLOTS) plays(i,j,t) = 1
                     ! (3) team i plays at team j exactly once
```

Handling the "no more than 3 home or away in a row" can be handled with constraints like

```
 forall (i in TEAMS, t in 1..2*n-5)
  1 <= sum(j in TEAMS) (plays(i,j,t)+plays(i,j,t+1)+
                   plays(i,j,t+2)+plays(i,j,t+3)) <= 3
                         ! (4) no more than 3 away in a row
```

These variables are not sufficient for the objective function, however. To get the distance traveled, additional variables are needed. Define `location(i,j,t)` to be 1 if team i is in location j in slot t (so `location(i,i,t)=1` implies i is home in slot t. Define `follows(i,i1,i2,t)` to be 1 if team i travels from location $i1$ to location $i2$ between slots t and $t+1$. Then the following constraints links all the variables together:

```
forall (i,j in TEAMS, t in SLOTS)
  if (i=j) then
    location(i,i,t) = sum(k in TEAMS) plays(k,i,t)
  else
    location(i,j,t) = plays(i,j,t)
  end-if
  ! (5) define location in terms of plays

forall (i in TEAMS)
  forall (j1,j2 in TEAMS, t in 1..2*n-3)
    follows(i,j1,j2,t) >=
         location(i,j1,t)+location(i,j2,t+1) - 1
  ! (6) define follows in terms of location
```

Now the total distance traveled is

```
 Total := sum(i,j,k in TEAMS, t in 1..2*n-3) DIST(j,k)*follows(i,j,k,t)+
  sum(i,j in TEAMS) DIST(i,j)*location(i,j,1)+
  sum(i,j in TEAMS) DIST(j,i)*location(i,j,9)
! (7) Distance traveled
```

The "no-repeaters" requirement is

```
forall (i,j in TEAMS, t in 1..2*n-3)
  plays(i,j,t)+plays(j,i,t)+
        plays(i,j,t+1)+plays(j,i,t+1) <= 1
  ! (8) no repeaters
```

This gives a complete formulation for the TTP. Unfortunately, putting this formulation into XPRESS-MP gives very poor result, even for $n = 6$. The initial relaxation value for that instance (letting XPRESS-MP be aggressive in adding initial cuts) is only 2186, while the optimal value is 23,916. After 1800 seconds, the lower bound has improved to only 5434 , while the best feasible solution found is 25650 . Again, running to optimal takes days.

To improve this formulation, it might be possible to add constraints to give a better linear relaxation. For instance, since the assignment for every week corresponds to a matching problem, Trick [5] suggests adding the "odd-set" constraints for each week.

An alternative (and better) approach is to reformulate by redefining the variables. The formulation given seems quite complicated because multiple types of variables are needed to correctly model the "distance traveled" aspects. Instead of using `plays(i,j,t)` as a fundamental variable, we can formulate this problem using variables corresponding to each road trip and home stand. Define `trips1(i,i1,t)` to be 1 if team i makes a trip to team $i1$ in slot t, and then returns home. Let `trips2(i,i1,i2,t)` be 1 if team i makes a trip to $i1$ in slot t then on to team $i2$ in slot $t+1$ and then returns home. `trips3(i,i1,i2,i3,t)` is the corresponding variable for length-3 trips: first to $i1$, then $i2$, then $i3$ before returning home. Similarly, `home1(i,t)` corresponds to a length-1 homestand in slot t; `home2(i,t)` a length-2 homestand in t and $t+1$; and `home3(i,t)` a length-3 homestand beginning at t.

Each road-trip variable has a cost, corresponding to the distance traveled. This gives an objective function of

```
Total := sum(i,i1 in TEAMS,t in SLOTS) cost1(i,i1,t)*trips1(i,i1,t)+
  sum(i,i1,i2 in TEAMS, t in SLOTS) cost2(i,i1,i2,t)*trips2(i,i1,i2,t)+
  sum(i,i1,i2,i3 in TEAMS, t in SLOTS) cost3(i,i1,i2,i3,t)*trips3(i,i1,i2,i3,t)
```

For constraints, we still have constraints that require each team to play at most one game in each slot. This looks like the following (the constraint for slots 1 and 2 is slightly different, based on which trips are feasible for the slot):

```
forall (i in TEAMS, t in 3..10)
  sum(i1 in TEAMS) trips1(i,i1,t) +
  sum (i1,i2 in TEAMS) (trips2(i,i1,i2,t)+trips2(i,i1,i2,t-1)) +
  sum (i1,i2,i3 in TEAMS) (trips3(i,i1,i2,i3,t)+trips3(i,i1,i2,i3,t-1)+
                           trips3(i,i1,i2,i3,t-2)) +
  home1(i,t) +
  home2(i,t)+home2(i,t-1) +
  home3(i,t)+home3(i,t-1)+home3(i,t-2) = 1
```

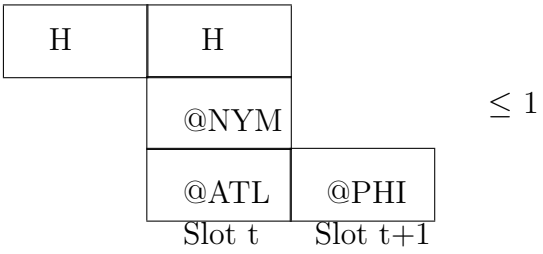

Fig. 2. Constraint: One Game per Slot

This is illustrated in Figure 2.

There are also constraints that either i is away in slot t or some team is away and playing i in that slot (the subscripts get a little messy):

```
forall (i in TEAMS, t in 3..10)
  sum(i1 in TEAMS) trips1(i,i1,t) +
  sum (i1,i2 in TEAMS) (trips2(i,i1,i2,t)+trips2(i,i1,i2,t-1)) +
  sum (i1,i2,i3 in TEAMS) (trips3(i,i1,i2,i3,t)+trips3(i,i1,i2,i3,t-1)+
                           trips3(i,i1,i2,i3,t-2)) +
  sum(i1 in TEAMS) trips1(i1,i,t)+
  sum(i1,i2 in TEAMS) trips2(i1,i2,i,t-1)+
  sum(i1,i2 in TEAMS) trips2(i1,i,i2,t)+
  sum(i1,i2,i3 in TEAMS) trips3(i1,i2,i3,i,t-2) +
  sum(i1,i2,i3 in TEAMS) trips3(i1,i2,i,i3,t-1) +
  sum(i1,i2,i3 in TEAMS) trips3 (i1,i,i2,i3,t)= 1
```

It is also necessary to ensure that no away trip for team i is followed immediately by another away trip:

```
forall (i,i1 in TEAMS, t in 3..2*n-3)
  trips1(i,i1,t)+trips1(i1,i,t)+
  trips1(i,i1,t+1)+trips1(i1,i,t+1)+
  sum(i2 in TEAMS) (trips2(i,i2,i1,t-1)+trips2(i1,i2,i,t-1)) +
  sum(i2 in TEAMS) (trips2(i,i1,i2,t+1)+trips2(i1,i,i2,t+1)) +
  sum(i2,i3 in TEAMS) (trips3(i,i2,i3,i1,t-2)+trips3(i1,i2,i3,i,t-2)) +
  sum(i2,i3 in TEAMS) (trips3(i,i1,i2,i3,t+1)+trips3(i1,i,i2,i3,t+1)) <= 1
```

Figure 3 illustrates this constraint.

Additional constraints preclude a home-stand after a home-stand and repeaters.

This formulation is inspired by the "variable generation" formulations useful in airline crew scheduling and many other applications (see Barhart et al. [2] for a fine survey). By encapsulated complicated structure (in this case, the distance traveled) in an expanded variable definition, we can create formulations with

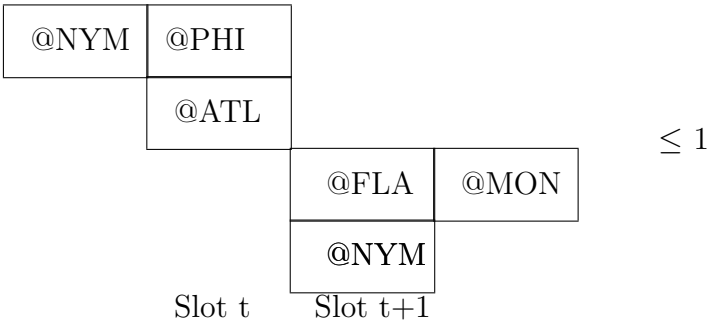

Fig. 3. Constraint: No away after away

tight relaxations. In this case, we do not have to resort to branch-and-price since the number of variables is still relatively small (4400 for the $n = 6$ case).

The strength of this formulation is shown immediately by XPRESS-MP. The initial relaxation value for $n = 6$ is 21624.7, an order of magnitude larger than that of our initial formulation. In fact, we obtain the optimal solution for this instance after "merely" 4136 seconds and 66,000 nodes in the tree.

Despite the improvement, the time required is still quite long, and does not bode well for solving larger instances. There are some "obvious" strengthenings available. For instance, for the "no away trip after away trip" constraint, it is possible to add more variables to the constraint. This is illustrated in Figure 4.

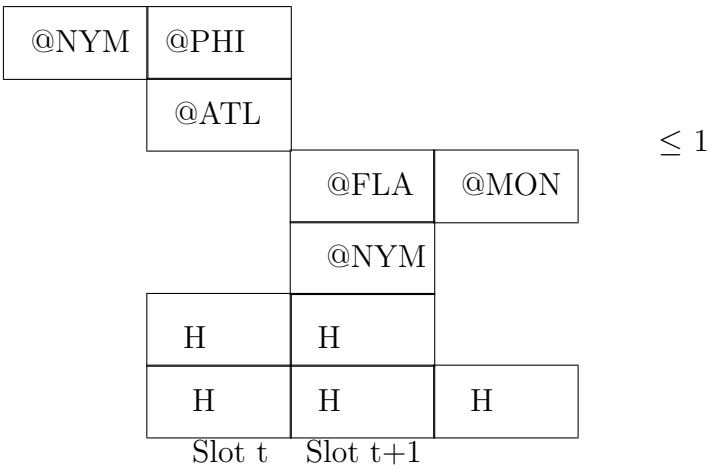

Fig. 4. Constraint: No away after away (strengthened)

This clearly is a strengthening, and resulted in significant improvement in previous versions of XPRESS-MP. Putting this constraint in the current version of XPRESS-MP, however, leads to another nasty surprise: the initial relaxation

value is identical, and the overall solution trajectory is a little worse (taking 15 seconds longer to find and prove an optimal solution).

What has happened? Again, I have added a strengthening that the system already knows about: the "strengthened" constraint is known as a "clique inequality" and is part of the XPRESS-MP repertoire. XPRESS-MP can generate that strengthening on its own: my strengthening of the constraint did not help. In fact, it slightly slowed the solution, for reasons that are unclear. If I want to improve my formulation, I need to find constraints or other reformulations that the sophisticated software package does not know about.

To summarize this example, again we have an initial formulation that is hopeless, and XPRESS-MP (other any other package, I believe) cannot improve on it. By reformulating the model using different variables, we ended up with a much better formulation. Trying to improve that model, however, let to overlap with the optimization package's knowledge, and led to no improvement.

4 Conclusions

Through two examples, I have argued that traditional approaches to "reformulation" in integer programming are not practical, since modern, sophisticated software already understands and implements obvious modeling "tricks". In order to improve on a formulation it is necessary to understand what the software knows and to provide insight beyond that knowledge base. For the transportation problem, this insight was in the form of a "redundant" but very important knapsack constraint that was "hidden" in the formulation. Adding this constraint allowed the software to add additional constraints, greatly improving the formulation. For the sports scheduling problem, the added knowledge was in the form of reformulating the variables of the problem to better encapsulate complicated structure. This reformulation was much better than the initial approach, though still not sufficient to solve even small instances (like the $n = 8$ instance).

These experiences suggest that, at least for integer programs, the art of improving formulations is getting more complicated: the simple rules of the past ("find formulations with better relaxations") are becoming less relevant as the relaxation used by the software is often not the relaxation given by the model. Understanding the software sufficiently to provide improved relaxation relative to the solved-relaxation requires highly sophisticated knowledge, and knowledge that can go out of date with every version released of the software.

But there still is room for the modeler to improve the formulations. Is it possible that the software packages will eventually "find" the knapsack constraint needed for the transportation problem? Probably. Can the software do the variable reformulation needed for the sports scheduling problem? Probably not, and almost certainly not if the integer program is only given the formulation in terms of variables, linear constraints, and linear objective. This sort of reformulation requires a deeper understanding of the problem structure.

In order to further develop "reformulations" as a research area and an area of practical interest, it would be useful to have more control over the solving of

models. It is in that spirit that I proposed the concept of defining some linear constraints as `knapsack` constraints while others are just linear constraints: this would define to the solver where the modeler thinks it is likely there are useful strengthenings (such as cover constraints).

Further, while most work in integer programming formulations have tried to find *one* integer programming formulation based on a higher level description of a problem, perhaps it is useful to come up with approaches that can generate multiple formulations for experimentation. Can we create a system that begins with a high level description of a problem and generates a series (or continuum) of formulations, perhaps based on the number of variables or constraints?

At this point, sophisticated software has embedded a lot of the simple reformulation rules integer programmers have developed. We now are challenged to find more sophisticated approaches to spur on the software.

References

1. Aardal, K. (1998). Reformulation of capacitated facility location problems: how redundant information can help, *Annals of Operations Research* **82** 289-308.
2. Barnhart C., Johnson E.L., Nemhauser G.L., Savelsbergh M.W.P. and Vance P.H. (1998), Branch-and-Price: Column Generation for Huge Integer Programs, Operations Research 46, 316.
3. Crowder H., Johnson E.L. and Padberg M.W. (1983), Solving Large Scale Zero-One Linear Programming Problems, *Operations Research* **31**: 803-834.
4. Easton, K., G.L. Nemhauser, and M.A. Trick (2003), Solving the Traveling Tournament Problem: A Combined Integer and Constraint Programming Approach, in *PATAT'2002*, E. Burke and P. Causmaecker (eds), Springer Lecture Notes in Computer Science **2740**, 63–77.
5. Trick, M.A. (2003). Integer and Constraint Programming Approaches for Round Robin Tournament Scheduling, in *PATAT'2002*, E. Burke and P. Causmaecker (eds), Springer Lecture Notes in Computer Science **2740**, 63–77 (2003).
6. Williams H.P. (1999), *Model Building in Mathematical Programming*, Wiley, New York.
7. XPRESS-MP Extended Modeling and Optimisation Subroutine Library, Reference Manual (2004), Dash Associates, Blisworth House, Blisworth, Northants.