Tower Puzzle Mixed Integer Programming Formulation

iMath

1MAR18

1 Abstract

This is an analysis of the Tower Puzzle introduced by the most recent Treasure Trail updates in the game RuneScape. We first briefly describe some general concepts and motivations behind mixed integer programming formulations and solution techniques. Using these concepts as a baseline, we discuss their applicability to the Tower Puzzle, and provide a formulation and general solution technique for solving any size instance of the puzzle.

2 Mixed Integer Programming

Mixed integer programming (MIP) problems are a specific type of decision problems in which some of the decision variables take on integer values. They are broadly useful for problems like scheduling, production, and routing; any time problem values are discrete quantities, MIPs tend to be appropriate modeling choices.

In general, integer programming problems are in NP. For sufficiently small problems, however, and for certain types of MIPs, solution times in commercial solvers tend to be extremely fast. As a result, modeling using MIPs tends to be both straightforward and suited to most applications.

3 Tower Puzzle Formulation

We will be modeling the Tower Puzzle with a MIP formulation. We will employ some relatively advanced techniques, but we will focus more heavily on the simpler portions of the model.

Standard MIP formulations can generally be broken down into the following components: an objective function, a set of decision variables, and a set of constraints. We will look at each of these as they correspond to our formulation, with the exception of the objective function. As we are not trying to maximize or minimize anything, we do not actually need an objective function; the constraints and variables will be sufficient to determine an answer.

3.1 Decision Variables/Constraints

The Tower Puzzle has two sets of information. First, we have the numbers around the outside, which we will call the "vision numbers." These describe how many towers can be seen in the row or column adjacent to the number. The second set of information is the values inside the boxes; in other words, the solution to the puzzle. These values are what the MIP solver will determine, and we refer to them as decision variables.

Given the name "mixed integer program," one might expect that the decision variables are simply each square, perhaps x_{ij}, where any x is the value in square (i, j). One might also expect that we will force the x_{ij}s to be integer, and specifically that each x_{ij} will belong to the set $\{1, 2, 3, 4, 5\}$. This is almost correct. We will indeed have variables x_{ij}, but we will actually not restrict these to be integer valued. Instead, we will create a set of binary variables, which are variables that can only take on a value of 0 or 1. These binary variables will be of the form $b_{i,j,k}$, where, as before, (i, j) will refer to square (i, j) of the solution. The index k will range from 1 to 5 (in general, from 1 to n, where n is the maximum number present in the puzzle - for example, a 3x3 grid will have $n = 3$, a 4x4 grid will have $n = 4$, etc.), and will control the number in square (i, j). This is a common technique, and allows us to enforce the puzzle constraint that each number 1 through 5 (or 1 through n) can appear just once per row or column. If we did not have these binary variables, and if we only had integer variables x_{ij}, we would need to find some way to make sure that no other x in row i or column j took on the same value. Suffice it to say that this is difficult, and we have made the correct choice by introducing the variables b.

The variables b_{ijk} need to follow the rules of the puzzle. Consider the variable b_{123}, and let us give it a value of 1. What does this mean? If $b_{ijk} = b_{123}$, and $b_{123} = 1$, this tells us that square (i, j), in this case $(1, 2)$, has a value of k, in this case 3. If square (1,2) has value 3, then no other square in row 1 can have a value of 3, and neither can any square in column 2. We can enforce this by adding up all of the values of b_{1j3} for row 1 (similarly the values of b_{i23} for column 2), and setting that sum equal to 1. Because the variables b_{ijk} can only take on values 1 or 0, and because we already have a variable equal to 1 in that row (and column), every other b_{1j3} (and b_{i23}) must have a value of 0. Additionally, if a square (i, j) has a value k, no other k can be active in square (i, j) - it already has a value! We generalize a bit across the indices, and this gives us our first three sets of constraints:

$$\sum_{i=1}^{n} b_{ijk} = 1 \quad \forall j \in 1..n, k \in 1..n,$$

$$\sum_{j=1}^{n} b_{ijk} = 1 \quad \forall i \in 1..n, k \in 1..n,$$

$$\sum_{k=1}^{n} b_{ijk} = 1 \quad \forall i \in 1..n, j \in 1..n^*.$$

* - It turns out we do not need this last set of constraints, as it is implied by the next set, and the values we restrict x to.

We still have yet to tie the values of the variables b_{ijk} to the values of x_{ij}, and we do this with our next set of constraints. Notice how we said that the index k in square (i, j)

determines the value of x_{ij} - but the values of b_{ijk} are 0 or 1, not 1 through n! It turns out there is a simple way to do this. All we need to do is multiply the value of b_{ijk} by k to represent the value x should take on in square (i, j) if $b_{ijk} = 1$. These constraints look like this:

$$x_{ij} = \sum_{k=1}^{n} k \cdot b_{ijk} \quad \forall i \in 1..n, j \in 1..n.$$

As a quick example, if we are looking at x_{34}, the value of x in the square $(3, 4)$, and if $n = 5$, we want the following to be true:

$$x_{34} = 1 \cdot b_{341} + 2 \cdot b_{342} + 3 \cdot b_{343} + 4 \cdot b_{344} + 5 \cdot b_{345}.$$

From the first three sets of constraints we generated, we know that exactly one of the b_{34k}s is equal to 1, so x *must* be in the set $\{1, 2, 3, 4, 5\}$. We are almost done! We just need to build in the set of constraints that correspond to the vision numbers.

3.2 Vision Number Constraints

Unfortunately, here is where it starts to get technical. For the purposes of this document, we will not get too much into specifics, so do not fret.

Notice that the vision numbers (1 through n) each refer to a specific set of numbers, regardless of whether the vision number corresponds to a row or a column. If $n = 3$, and if the vision number is 3, we know that the numbers we see must be in the order (1,2,3). If $n = 3$ and the vision number is 2, we must be seeing numbers from the following set: $\{(1, 3, 2), (2, 1, 3), (2, 3, 1)\}$. For any n and for any vision number, we can generate the set of number that we must see. These sets form a partition of the permutations; every set has exactly one vision number. If we take each vision number, and consider all of the sets to which it corresponds, we have a set of integer points that completely describe that vision number (bear with me).

There is some polyhedral theory that lets us retrieve a set of inequalities from a convex set; basically, because we have a set of integer points, we can retrieve some inequalities that describe that set of points. If we take these inequalities and associate them with the correct indices in our formulation, we can actually completely describe the Tower Puzzle problem and retrieve a solution!

Luckily for us (and for whoever is implementing this formulation as a solution technique), there is a package called Polyhedra.jl in the software language Julia that will generate exactly those inequalities that we need. As there are rather a lot of inequalities in some of the sets, and because they are difficult to interpret, we will use some set notation to make our formulation much simpler. The only thing we still need at this point is to take in the vision numbers as input. There are four sets of vision numbers, from the top, right, bottom, and left, so we will refer to them respectively as v_t, v_r, v_b, and v_l. We differentiate between them because they each affect different sets of indices; this is clearer in the attached code. Using these vision numbers, we generate the set of constraints using the Julia package, and refer to the body of constraints as P_v; if the vision number v_l is 3, for example, the set of constraints applying to all of the x_{ij}s in row 3 is P_3 (P being the common way to represent a polytope

for anyone familiar with the subject). This gives us our final sets of constraints - notice that the indices on the xs contain the signs $-$ and $\bullet$. The sign $-$ represents iterating through the indices backwards (if we are looking at a right or bottom vision number, we want to impose constraints on the xs from largest index to smallest), and the sign $\bullet$ indicates that we want to apply the constraints to every x in the row or column. The constraints are below:

$$\begin{aligned} x_{\cdot j} &\in P_v \quad \forall j \in 1..n, v \in v_t, \\ x_{-i\cdot} &\in P_v \quad \forall i \in 1..n, v \in v_r, \\ x_{\cdot -j} &\in P_v \quad \forall j \in 1..n, v \in v_b, \\ x_{i\cdot} &\in P_v \quad \forall i \in 1..n, v \in v_l, \end{aligned}$$

There is some necessary bookkeeping to keep all the indices straight, but we'll ignore that for now.

3.3 Formulation

We can finally put it all together and retrieve our formulation. Without further ado:

$$\begin{aligned} \min\ & 0 \\ \text{subject to}\ & \sum_{i=1}^{n} b_{ijk} = 1 \quad \forall j \in \{1..n\}, k \in \{1..n\} \\ & \sum_{j=1}^{n} b_{ijk} = 1 \quad \forall i \in \{1..n\}, k \in \{1..n\} \\ & x_{ij} - \sum_{k=1}^{n} k \cdot b_{ijk} = 0 \quad \forall i \in \{1..n\}, j \in \{1..n\} \\ & x_{\bullet j} \in P_v \quad \forall j \in 1..n, v \in v_t \\ & x_{-i\bullet} \in P_v \quad \forall i \in 1..n, v \in v_r \\ & x_{\bullet -j} \in P_v \quad \forall j \in 1..n, v \in v_b \\ & x_{i\bullet} \in P_v \quad \forall i \in 1..n, v \in v_l \\ & x_{ij} \leq n \\ & x_{ij} \geq 1 \\ & b_{ijk} \in \{0, 1\} \end{aligned}$$

4 Implementation and Results

If you've read this far, congratulations! We've been through quite a bit, so I'll make this section brief. As mentioned in the previous section, I used a Julia package called Polyhedra.jl to generate the constraints in the set P_v. The rest of the code is also in Julia, and basically

generates the other constraints described in the full formulation. The code will be attached in the Reddit post, and you can ask any questions of me via Discord, username Roscroft8299.

I also interfaced the solver with my clan's Discord bot. Unfortunately, I cannot make it widely available because of my MIP solver. I used Gurobi, which requires an academic license, and I am not able to distribute the license necessary to make the solver work anywhere. It is possible to use a free solver, but I found out today that Alt1 has already implemented a solver and so I imagine the demand for a Discord bot solution will be quite low.

To its credit, however, this integer program executes *extremely* quickly; the solver output says it takes 0.00 seconds to find a solution (which means it's about as quick as it can be) for $n = 5$. In my first iteration, I actually regenerate the constraints every time (which in general takes a nontrivial amount of time) and still managed to get near-instantaneous solutions. So it works!

I understand that this got pretty technical, so if anyone has any questions, please feel free to ask. This area is my research focus, and solving problems like this one is a lot of fun. Thanks for reading!