Nonokenken: Solving Puzzles Using Integer Programs

Kira Goldner, Ryo Kimura, Ari Schwartz

1 Abstract

We examine methods of integer programming in order to solve two of our favorite puzzles: nonograms and KenKen. The former is a puzzle in which each cell of a board must be colored either black or white. The clues provided in the puzzle indicate the number and length of black-cell clusters that must appear in each row and column of the board. The latter, KenKen, is a puzzle that requires the solver to fill an $n \times n$ grid of cells with integers from 1 to n such that the solution is a Latin square and adheres to mathematical operation constraints within designated blocks of cells. This paper addresses an IP formation for solving nonograms, various methods which generate nonogram puzzles from images, and an IP formulation for solving KenKen puzzles.

2 Solving Nonograms using IP

2.1 Defining the Nonogram

A typical nonogram puzzle consists of a blank $m \times n$ grid of squares and a sequence of cluster-sizes for each row and column. We define a *cluster* as simply a set of consecutive squares in a single row or column that are all colored black, and the *size* of the cluster as the number of consecutive squares in the cluster.

The goal is to color each square in the grid either black or white such that for each row and column,

- the size of the clusters matches the given cluster-size sequence exactly, where "earlier" clusters are to the left of and above "later" clusters
- each cluster is separated by at least one white square

An example of a solved nonogram is given in figure 1.

2.2 Formulating the Problem as an IP

In order to solve the nonogram using techniques from optimization, we must be able to formulate the problem as an IP. Our formulation follows the suggestion given by Bob Bosch [1] in the recreational mathematics column *Optima mind sharpener*. This section is a brief summary of his article, which appeared in issue 65 of *Optima*, the newsletter of the Mathematical Programming Society.

Bosch's motivating idea for this IP formulation is to view nonograms as two interlocking tiling

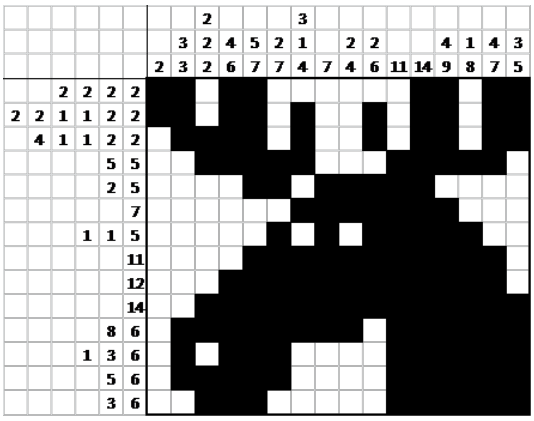


Figure 1: Example of Solved Nonogram

problems. More specifically, we first view the rows and columns as two separate tiling problems, where we are given a sequence of cluster sizes, and we must determine where to place the clusters to satisfy the sequence. We then impose the additional constraint that the rows and columns must agree in terms of which squares are colored black on the common grid, effectively "combining" the solutions of the two tiling problems into a single solution for the nonogram.

2.3 Data and Variables

To simplify the general definitions of variables and constraints, we first define the following data variables:

$$\begin{aligned} m &= \text{the number of rows} \\ n &= \text{the number of columns,} \\ a_i &= \text{the number of clusters in row } i \\ b_j &= \text{the number of clusters in column } j \\ p_{i,1}, p_{i,2}, \ldots, p_{i,a_i} &= \text{the cluster-size sequence for row } i \\ q_{j,1}, q_{j,2}, \ldots, q_{j,b_i} &= \text{the cluster-size sequence for column } j \end{aligned}$$

In addition, define the *position* of a row/column cluster to be the number of the column containing the leftmost square of the row cluster, or the number of the row containing the topmost square of the column cluster. Then, let

$$\begin{aligned} l_{i,t} &= \text{the leftmost possible position of row } i\text{'s } t^{\text{th}} \text{ cluster} \\ r_{i,t} &= \text{the rightmost possible position of row } i\text{'s } t^{\text{th}} \text{ cluster} \\ u_{j,t} &= \text{the highest possible position of column } j\text{'s } t^{\text{th}} \text{ cluster} \\ d_{j,t} &= \text{the lowest possible position of column } j\text{'s } t^{\text{th}} \text{ cluster} \end{aligned}$$

where $1 \leq t \leq a_i$ for row i and $1 \leq t \leq b_j$ for column j. These values tell us the range of possible positions of every row and column cluster in the problem. E.g., if $l_{2,3} = 5$ and $r_{2,3} = 8$, then row 2's 3rd cluster must be placed at 5, 6, 7, or 8.

We can now define the necessary variables to solve the problem. First, let

$$z_{i,j} = \begin{cases} 1 & \text{if square } (i,j) \text{ (row } i\text{, column } j\text{) is colored black,} \\ 0 & \text{if it is colored white} \end{cases}$$

for all $1 \leq i \leq m$ and $1 \leq j \leq n$, i.e., for all squares in the grid. Next, let

$$y_{i,t,j} = \begin{cases} 1 & \text{if } t^{\text{th}} \text{ cluster in } i^{\text{th}} \text{ row is placed with leftmost square on column } j, \\ 0 & \text{if not} \end{cases}$$

for all $1 \leq i \leq m$, $1 \leq t \leq a_i$, and $l_{i,t} \leq j \leq r_{i,t}$, i.e., for all possible row cluster positions. Finally, let

$$x_{j,t,i} = \begin{cases} 1 & \text{if } t^{\text{th}} \text{ cluster in } j^{\text{th}} \text{ column is placed with topmost square on row } i, \\ 0 & \text{if not} \end{cases}$$

for all $1 \leq j \leq n$, $1 \leq t \leq b_j$, and $u_{j,t} \leq i \leq d_{j,t}$, i.e., for all possible column cluster positions.

2.4 Constraints

In addition, we have two classes of constraints: one for solving the row and column tiling problems separately, and one for ensuring the solutions to the two problems are compatible with each other. The first class of cluster constraints:

- Ensure that each row cluster appears in exactly one position, i.e.,

$$\sum_{j=l_{i,t}}^{r_{i,t}} y_{i,t,j} = 1,$$

  for all $1 \leq i \leq m$ and $1 \leq t \leq a_i$.

- Ensure that later row clusters appear to the right of earlier row clusters. To do this, note that if row i's t^{th} cluster is placed at column j, then the leftmost possible position of the next cluster is $j + p_{i,t} + 1$ (the t^{th} cluster starts at column j, has size $p_{i,t}$, and we need at least one space between clusters). Hence,

$$y_{i,t,j} \leq \sum_{j'=j+p_{i,t}+1}^{r_{i,t+1}} y_{i,t+1,j'},$$

  for all $1 \leq i \leq m$, $1 \leq t \leq a_i$, and $l_{i,t} + 1 \leq j \leq r_{i,t}$.

- Ensure the same thing is true for column constraints.

The second class of double coverage constraints:

- Ensure that if square (i,j) is black, then there is at least one row cluster in row i and one column cluster in column j that is positioned so it covers that square, i.e.,

$$z_{i,j} \leq \sum_{\substack{\text{all row cluster positions } j' \\ \text{that can cover square (i,j)}}} y_{i,t,j'},$$

$$z_{i,j} \leq \sum_{\substack{\text{all column cluster positions } i' \\ \text{that can cover square (i,j)}}} x_{j,t,i'},$$

for all $1 \leq m$ and $1 \leq j \leq n$. To express this more mathematically, note that if a row cluster at position $y_{i,t,j'}$ covers square (i, j), then j' starts at most $p_{i,t} - 1$ squares before j and never after j. In addition, in order for j' to be a feasible position, $l_{i,t} \leq j' \leq r_{i,t}$ must be true. Hence, the range of j' we should consider is

$$\min\{l_{i,t}, j - p_{i,t} + 1\} \leq j' \leq \max\{r_{i,t}, j\}.$$

Since there are a_i total row clusters in row i, this gives us the row coverage constraint

$$z_{i,j} \leq \sum_{t=1}^{a_i} \sum_{j'=\min\{l_{i,t}, j-p_{i,t}+1\}}^{\max\{r_{i,t},j\}} y_{i,t,j'}$$

for all $1 \leq i \leq m$ and $1 \leq j \leq n$. A similar argument shows that the column coverage constraint can be expressed as

$$z_{i,j} \leq \sum_{t=1}^{b_j} \sum_{i'=\min\{u_{j,t}, i-q_{j,t}+1\}}^{\max\{d_{j,t},i\}} y_{j,t,i'}$$

for all $1 \leq i \leq m$ and $1 \leq j \leq n$.

- Ensure white squares are never covered by any row or column clusters. This can be restated as ensuring that every squared covered as a consequence of placing a row cluster at position $y_{i,t,j'}$ must be colored black, i.e.,

  $$z_{i,j} \geq y_{i,t,j'} z_{i,j} \geq x_{j,t,i'}$$

  for all row cluster positions j' and column cluster positions i' that can cover square (i, j), for all $1 \leq i \leq m$ and $1 \leq j \leq n$. Note that the range of j' and i' are the same as those used above. Hence, we create these constraints

  - for all j' such that $l_{i,t} \leq j' \leq r_{i,t}$ and $j - p_{i,t} + 1 \leq j' \leq j$, and
  - for all i' such that $u_{j,t} \leq i' \leq d_{j,t}$ and $i - q_{j,t} + 1 \leq j' \leq j$,

  for all $1 \leq i \leq m$ and $1 \leq j \leq n$.

2.5 Necessity of Double Coverage Constraints

Some readers may be skeptical that double coverage constraints are necessary. After all, there is a single variable $z_{i,j}$ which controls whether a given cell is black or white; is that not enough to ensure the consistency of the row cluster and column cluster solutions?

The answer is a resounding yes. First, it is clear that simply eliminating the double coverage constraints from our current formulation would result in a trivial solution of $z_{i,j} = 0$ for all i, j, since the cluster constraints by themselves do not use the $z_{i,j}$ variable.

Furthermore, we cannot use the $z_{i,j}$ variables directly to write the cluster constraints, since a particular square (i, j) may or may not be part of a given cluster depending on how other squares in the entire grid are colored. Since IP constraints must be specified *prior* to solving the formulation, we need double coverage constraints to ensure that the squares in the grid stay consistent relative to the row cluster and column cluster solutions.

2.6 Implementation

We used GLPK (GNU Linear Programming Kit) and its associated GNU MathProg modeling language in our implementation. Our model takes in the size of the grid and the row and column cluster-sizes of the nonogram, then automatically generates the IP, which is then solved using GLPK's standalone solver glpsol. To increase the speed of the solver (and the scale of nonograms solvable in a reasonable amount of time), we then convert the IP into a CNF-Satisfiability problem which is then solved using the MINISAT solver. The model was written by Andrew Makhorin [3] and was included with the GLPK software as an example.

Using this model we are able to solve general nonograms of size 50x50 in roughly 20 seconds. Nonograms with unique solutions can be solved even more quickly.

3 Puzzle Generation and Nonogram Art

Inspired by past research on Domino Art by Bob Bosch, we next focused on using a solved nonogram board to represent a recognizable image. Each cell in the nonogram board has a variable $x_{i,j}$ to represent it, where:

$$x_{i,j} = \begin{cases} 1 & \text{if square } (i,j) \text{ is colored black,} \\ 0 & \text{if it is colored white} \end{cases}$$

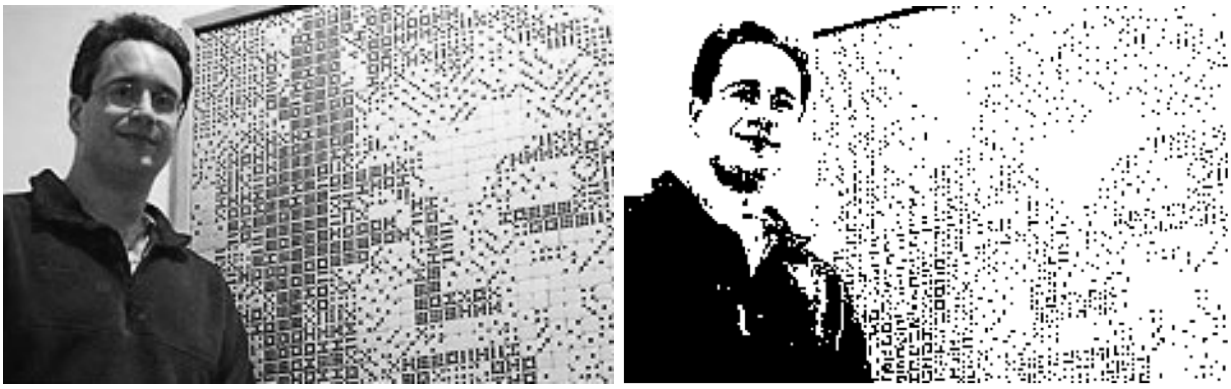
The first step of this process was to convert an image into a solved nonogram board. Intuitively, we create the desired nonogram board by dividing an image into its pixels, where each pixel is a cell on the board. We then convert a color image to greyscale. From greyscale, we consider two different methods in order to determine our nonogram board: the threshold method and the translucent squares method.

3.1 The Threshold Method

One method in which to determine whether to set each cell in our resulting nonogram board to 0 or 1 is by asking whether each pixel in an image is black or white. Given a greyscale image, however, pixels are not simply black or white—each pixel has a value from 0 to 255 which corresponds to the specific shade of grey of that pixel. The closer to 0 the value is, the darker the shade of grey of the pixel.

We use a threshold method, in which we arbitrarily select a value from 0 to 255. If a pixel's grey-value is above our selected threshold value (i.e., closer to white), then we set the corresponding nonogram cell to white by setting $x_{i,j}$ for that pixel $= 0$. If a pixel's grey-value is below the

threshold value, then we set the corresponding $x_{i,j} = 1$. In our examples, we chose 60 to be the threshold value.



The original greyscale image is on the left. The threshold method was applied to this image, resulting in a picture comprised of black and white pixels on the left.

3.2 Translucent Squares

While the threshold method produces a sufficient white and black board, we looked for a way to shade an image with more subtlety, taking into account lighter and darker regions. However, we wanted to maintain binary decision variables.

In a 2004 paper by Bob Bosch and Adrianne Herman on *Pointillism via Linear Programming* [2], the authors address a method of using translucent discs to create images. Their approach seemed well suited to help accomplish our goal. The paper discusses setting a binary variable for each pixel in an image. If a cell (i, j) has $x_{i,j} = 1$, then a translucent disc of a certain constant opacity is centered at cell (i, j). The discs, however, are large enough that they overlap onto several neighboring cells. The darkness of a cell is thus determined not simply by its own $x_{i,j}$ value, but by the $x_{i',j'}$ values of its neighbors as well.

We modify this idea slightly to better apply to our nonogram example. If a cell (i, j) has $x_{i,j} = 1$, then we set a 3×3 translucent square centered at (i, j) that darkens the cell (i, j) along with its eight neighbors. We define this set of cells for any (i, j) to be the neighbor set of (i, j), notated as $N(i, j)$. For edges and corners, neighbor sets are modified to appropriately contain 5 and 3 neighbors respectively.

We then use a slightly modified version of the integer program introduced in *Pointillism via Linear Programming* [2] to minimize the difference between ideal shading of a cell and that cell's actual shading as determined by our IP.

For each cell in the nonogram board—each pixel in the image—we determine $g_{i,j}$, a variable that represents the ideal number of translucent squares that will shade cell (i, j). By how we defined the size of our translucent squares, it is evident that a cell can have from 0 to 9 squares shading it. We use each pixel's grey-value of 0 to 255 to determine $g_{i,j}$. Call $v_{i,j}$ the grey-value

for a cell (i, j). Then we have that, for each cell (i, j):

$$g_{i,j} = \frac{9 \times v_{i,j}}{255}$$

We then define $d_{i,j}$ to be the actual number of translucent squares shading cell (i, j) — the number of cells in the neighbor set of cell (i, j) that have translucent squares centered at them such that they also shade cell (i, j). That is:

$$d_{i,j} = \sum_{(i',j') \in N(i',j')} x_{i',j'}$$

Then our objective function is simply to minimize the difference between $g_{i,j}$ and $d_{i,j}$ for each cell. In order to avoid non-linear absolute value constraints, we add a new variable $z_{i,j}$ to be the positive difference between $g_{i,j}$ and $d_{i,j}$ for each cell. This gives us our integer program in full.

$$\begin{aligned} \min \quad & \sum_i \sum_j z_{i,j} \\ \textbf{s.t.} \quad & z_{i,j} \geq d_{i,j} - g_{i,j} \forall i, j \\ & z_{i,j} \geq -d_{i,j} + g_{i,j} \forall i, j \\ & d_{i,j} = \sum_{(i',j') \in N(i',j')} x_{i',j'} \end{aligned}$$

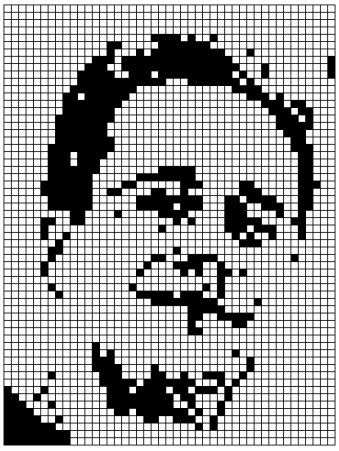
The result of solving this integer program is a solved nonogram board that can be used to project a shaded image of our original image.

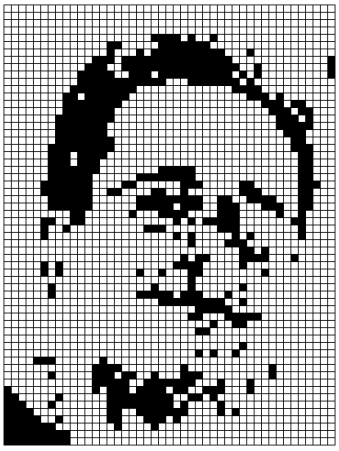
The original greyscale image is on the left. The solution to translucent squares IP of this image is on the right.

3.3 Between Images and Nonograms

Using the threshold method or the translucent squares method, we can find a solved nonogram board that represents our original image. We wrote a simple program that takes in our solved board and generates the row and column cluster constraints that are given in a nonogram puzzle. We feed these constraints into our IP solver for nonograms and compare the solution of the puzzle to the image that inspired it.

Here are the results that we obtained using the threshold method on an image to produce nonogram puzzle constraints. The desired result is on the left, while a solved puzzle corresponding to the constraints is on the right.





Clearly our nonogram puzzle does not have a unique solution, as the solved board does not match our initial puzzle which generated the constraints. However, if we were set on finding the exact same solution, we could add a small number of additional constraints to require specific cells to be black. Further work on the topic is necessary to determine the number of additional constraints required.

The images below include the original greyscale image on the left, the solution to our translucent squares integer program to within 0.05% of optimality in the center, and the solution to the nonogram puzzle generated by the center image then projected as translucent squares on the right.

4 An IP KenKen Solver

A KenKen puzzle is an n x n grid of cells. A solved KenKen puzzle contains exactly one integer from 1 to n in each cell such that there are no repeated entries in any row or column. An additional feature of KenKen puzzles is that the puzzle is partitioned into "blocks" of cells whose entries must satisfy certain arithmetical constraints. These constraints are indicated by a number and an operation. Using the given operation, cell entries within the block must combine to produce the given number. A prototypical 4 x 4 KenKen is shown below:

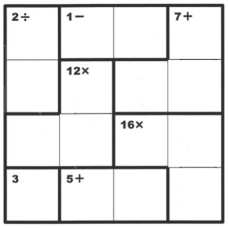


Figure 2. A 4 x 4 KenKen

To formulate our integer program KenKen solver, we begin with the decision variables $x_{i,j,k}$, where

$$x_{i,j,k} = \begin{cases} 1 & \text{if the integer } k \text{ is placed in row } i, \text{ column } j \\ 0 & \text{if not} \end{cases}$$

We can use these variables to write constraints that ensure a completed KenKen is a Latin square. First, we place exactly one entry in each cell. For a given cell (i, j), the sum of the $x_{i,j,k}$'s must be 1. That is:

$$\sum_{k=1}^{n} x_{i,j,k} = 1 \; \forall \; i, j$$

We also must have exactly one of each integer from 1 to n in every row and column. We accomplish this using the constraints

$$\sum_{j} x_{i,j,k} = 1 \; \forall \; i, k$$

for rows and the constraints

$$\sum_{i} x_{i,j,k} = 1 \; \forall \; j, k$$

for the columns.

Before proceeding with the other constraints, it is convenient to define a new set of variables. Let

$$p_{i,j} = \sum_{k=1}^{n} k \times x_{i,j,k} \; \forall \; i, j$$

Notice that, by expressing cell entries as $p_{i,j}$'s, we convert from the binary variable $x_{i,j,k}$ to the actual entry from 1 to n that appears in cell (i, j) of the puzzle. To satisfy a block that contains a single cell, we use a constraint that says that the entry in that cell must be the given value. If one of the puzzle's clues says that the entry in cell (i, j) must be k, then we use the constraint $p_{i,j} = k$. If the puzzle contains cells that must be combined via addition, then we set the sum of the $p_{i,j}$'s within that block equal to the specified value. For example, if the puzzle contains a block B whose entries must sum to l, then we use the constraint

$$\sum_{i} \sum_{j} p_{i,j} = l \; \forall \; (i, j) \in B$$

Constraints for the other three operations—multiplication, division, and subtraction—are more difficult to implement. The approach we use for addition block constraints will not work for the

other operations. Attempts to multiply or divide the $p_{i,j}$'s result in nonlinear constraints. Also, division and subtraction are not commutative operations. KenKen puzzles do not specify the order in which we subtract or divide cells, as long as it is possible to subtract or divide cell entries in some way such that the difference or quotient gives us the desired value. So although subtraction is a linear operation, we would need to employ constraints like $|p_{i,j} - p_{i+1,j}| = t$ for two adjacent cells whose difference must be t. Introducing absolute value terms is consistent with the rules of KenKen, but we wish to avoid the resulting nonlinearity.

The alternative approach we devised involves a consideration of all sets of integers from 1 to n that, when combined in some order using a block's designated operation, satisfy the given arithmetical constraint. Each set must contain the same number of integers as cells in the block, since each of these sets represents entries that could be placed in the block. We will illustrate this approach using multiplication as an example, but the approach can just as easily be used to accommodate subtraction and division constraints. Given a block of c cells and product value p, we consider all possible factorizations of p such that the factorization is comprised of c integers between 1 and n, with n being the largest possible entry for a cell in an n x n KenKen. Let F be the set of all such factorizations. Note that $F = \{f_1, f_2, ..., f_m\}$, where each f_y is a possible set of c entries that could fill the cells that comprise a multiplication block. We can think of the f_y's as a new decision variable. Let

$$f_y = \begin{cases} 1 & \text{if factor combination } f_y \text{ is used} \\ 0 & \text{if not} \end{cases}$$

If f_y=1, then the corresponding cells in the multiplication block must each contain one of the factors represented in f_y.

To accomplish this, we use constraints of the following form:

$$f_y \leq \sum_i \sum_j x_{i,j,k} \; \forall \; k \in f_y \text{ and } \forall \; (i,j) \in M,$$

where M is the multiplication block of interest. It is not difficult to see that this constraint gives us the desired result. If f_y=1 (if we use factor combination f_y), then each of the factors in f_y appears exactly once in the block. However, this approach assumes that each number in a particular factor combination is unique. Suppose that a certain factor combination has h copies of the same factor. Then, we must modify our constraint to be

$$h \times f_y \leq \sum_i \sum_j x_{i,j,k} \; \forall \; n \in f_y \text{ and } \forall \; (i,j) \in M$$

This ensures that h of the cells in M contain the factor that appears h times in f_y. The benefit of using this method is that all constraints are linear. The downside is that the process of determining F is not automated.

We can handle subtraction and division in a similar manner. First, generate the set of all possible integer combinations that could be placed in the cells within a block to satisfy a given arithmetical clue. Next, write constraints that ensure the correct entries upon choosing a particular combination. The exact location of each entry will be determined by other puzzle constraints.

We used Microsoft Excel to solve the KenKen shown in Figure 1. Due to the nature of that specific puzzle, not every arithmetical constraint was needed to produce the solution. We did not need to include the subtraction constraint for cells (1,2) and (1,3); nor did we need the multiplication constraint for cells (3,3), (3,4), and (4,4). We suggest that readers interested in solving KenKen puzzles input only as many constraints as are needed for Excel to find the solution. After including the Latin square constraints, single-cell block constraints, and addition constraints, we suggest handling the remaining constraints one at a time. Running the solver after each new arithmetical constraint has been added to the integer program is the most efficient strategy. Inputting all constraints at once is not necessary.

Future work for this project includes implementing our IP in software that automatically generates the possible sets of integers we need to accommodate multiplication, division, and subtraction operation blocks. In addition, if we could program the software to generate the requisite constraints, then that would eliminate the need to input constraints manually.

References

[1] Bosch, Robert A: "Painting By Numbers." *Optima* **65**, 16-17, (2001). <http://www.oberlin.edu/math/faculty/bosch/pbn-page.html>.

[2] Bosch, B. and Herman, A. 2005. "Pointillism via Linear Programming." *The UMAP Journal* **26** (4): 405-411.

[3] Makhorin, Andrew: "Solving Paint-By-Numbers Puzzles with GLPK." *GLPK Documentation*, (2011).

We affirm that we have adhered to the Honor Code in this assignment.