



INFORMS Transactions on Education

Publication details, including instructions for authors and subscription information:
<http://pubsonline.informs.org>

Puzzle—Solving Smartphone Puzzle Apps by Mathematical Programming

Sönke Hartmann

To cite this article:

Sönke Hartmann (2018) Puzzle—Solving Smartphone Puzzle Apps by Mathematical Programming. INFORMS Transactions on Education 18(2):127-141. <https://doi.org/10.1287/ited.2017.0182>

Full terms and conditions of use: <https://pubsonline.informs.org/Publications/Librarians-Portal/PubsOnLine-Terms-and-Conditions>

This article may be used only for the purposes of research, teaching, and/or private study. Commercial use or systematic downloading (by robots or other automatic processes) is prohibited without explicit Publisher approval, unless otherwise noted. For more information, contact permissions@informs.org.

The Publisher does not warrant or guarantee the article's accuracy, completeness, merchantability, fitness for a particular purpose, or non-infringement. Descriptions of, or references to, products or publications, or inclusion of an advertisement in this article, neither constitutes nor implies a guarantee, endorsement, or support of claims made of that product, publication, or service.

Copyright © 2017, The Author(s)

Please scroll down for article—it is on subsequent pages



With 12,500 members from nearly 90 countries, INFORMS is the largest international association of operations research (O.R.) and analytics professionals and students. INFORMS provides unique networking and learning opportunities for individual professionals, and organizations of all types and sizes, to better understand and use O.R. and analytics tools and methods to transform strategic visions and achieve better outcomes.

For more information on INFORMS, its publications, membership, or meetings visit <http://www.informs.org>

Puzzle

Solving Smartphone Puzzle Apps by Mathematical Programming

Sönke Hartmann^a

Hamburg School of Business Administration (HSBA), 20457, Hamburg, Germany

Contact: soenke.hartmann@hsba.de (SH),  <http://orcid.org/0000-0001-6687-1480>

Received: March 16, 2017

Revised: May 12, 2017; June 6, 2017


Accepted: June 9, 2017

Published Online in Articles in Advance:
October 23, 2017

<https://doi.org/10.1287/ited.2017.0182>

Copyright: © 2017 The Author(s)

Abstract. This paper considers six logic puzzles (i.e., single player games) that are available as smartphone apps. The six logic puzzles are: *Thermometer Puzzles*, *Kakuro (Cross Sums)*, *Match 22: Color Puzzle Game*, ∞ *Infinity Loop*, *Slider*, and *Flow Free*. We provide mathematical models that can be applied to obtain solutions for these puzzles. In OR/MS lectures, the apps and models can be used as examples or exercises when teaching mathematical programming and for discussion of familiar model types such as shortest path and network flow models. Given the popularity of logic puzzle apps on smartphones, such exercises might be motivating for students. The level of difficulty of the models presented here varies from easy to more difficult.

 **Open Access Statement:** This work is licensed under a Creative Commons Attribution 4.0 International License. You are free to copy, distribute, transmit and adapt this work, but you must attribute this work as "INFORMS Transactions on Education. Copyright © 2017 The Author(s). <https://doi.org/10.1287/ited.2017.0182>, used under a Creative Commons Attribution License: <https://creativecommons.org/licenses/by/4.0/>."

Supplemental Material: The MathProg files are available at <https://www.informs.org/Publications/Subscribe/Access-Restricted-Materials>.

Keywords: puzzle • teaching modeling

1. Introduction

Many logic puzzles can be modeled as integer programs, and solvers can be applied to find solutions. This has motivated OR/MS lecturers (including the author of this paper) to use such puzzles as interesting exercises in class. Dozens of logic puzzles along with IP models have been described in the literature, including the following: Chlond (2005) applies mathematical modeling techniques to Sudoku. The classical n queens problem is studied by Letavec and Ruggiero (2002), while Chlond and Toase (2002) look at chessboard placement puzzles in a more general context. DePuy and Taylor (2007) address board puzzles such as the Cracker Barrel peg game and the rush hour problem. Pearce and Forbes (2017) discuss the Fillomino Puzzle. Yeomans (2003) examine what is usually called "Einstein's riddle;" Chlond (2014) consider other logic grid puzzles.

Most puzzles were originally paper-and-pencil or board games, but many are also available for smartphones, including the puzzles mentioned above. There are also some smartphone puzzles that have not been available as paper-and-pencil or board games. Logic puzzle apps are very popular: According to Google Play store, the ∞ *Infinity Loop* and *Flow Free* apps (which will be discussed in this paper) were installed more than 10,000,000 and 100,000,000 times, respectively. It might be especially interesting for students to work on OR exercises that are based on popular apps they can download on their phones or tablets.

Here, we present six smartphone apps that can be used as entertaining and motivating examples and exercises in mathematical programming. The apps were selected based on several criteria. First, we wanted to achieve different levels of difficulty of the resulting models, varying between easy and more difficult. Second, we wished to cover a broad range of problem types and modeling techniques. Thus, we included different types of variables, mostly binary (as would probably be expected for logic games) but also integer and continuous. We also included specific problem types such as a shortest path and a network flow problem to allow for discussions of such familiar concepts in a rather unusual context. Third, we wanted to encourage students to work with the apps. Therefore, we preferred puzzles with simple, easy to learn, rules, and we chose apps that are available free (all apps discussed here are available free for Android; some are available for iOS as well).

We proceed as follows: Section 2 presents the *Thermometer Puzzles* app, which leads to a rather straightforward mathematical model. Section 3 describes a model for the *Kakuro (Cross Sums)* app; it could be used as an easy to moderate exercise in an introductory lecture. In Section 4, we discuss the app *Match 22: Color Puzzle Game* for which the model is somewhat more advanced. This section also examines the ∞ *Infinity Loop* app to which a similar model can be applied. The last two apps are closely related to network models. Section 5 introduces the app *Slider*, which can be

modeled as a shortest path problem. This would be an easy exercise. Finally, Section 6 tackles the *Flow Free* app that leads to an extended network flow model that can be viewed as more difficult. It also allows students to revisit the subtour elimination constraints from the traveling salesman problem (TSP) and to discuss the computational impact of different modeling approaches. Section 7 presents concluding remarks.

2. Thermometer Puzzles

In the *Thermometer Puzzles* app, a grid with $n \times n$ cells is filled with thermometers. Each thermometer occupies two or more consecutive cells, in a row or a column (see Figure 1). The player must determine which cells are filled with mercury. For each row and each column, the number of filled cells is given. Moreover, one has to take into account that thermometers are always filled from bottom to top (the bottom is indicated by a circle). This implies that if a cell is filled with mercury, then all cells towards the bottom of the same thermometer must be filled as well. Likewise, if a cell is not filled, then all cells towards the top must not be filled.

To develop a mathematical model for this puzzle, we introduce some notation. Let R_i denote the filled cells in row i , and let C_j denote the filled cells in column j . Moreover, we let D be the set that includes all cell dependencies related to the thermometers. Each such dependency is given by a quadruple $(i, j, k, l) \in D$, reflecting that cell (k, l) can only be filled if cell (i, j) is filled as well.

The binary decision variables are defined by

$$x_{ij} = \begin{cases} 1, & \text{if cell } (i, j) \text{ is filled,} \\ 0, & \text{otherwise.} \end{cases}$$

As is often the case with puzzles, we are just looking for a feasible solution, so there is no optimization and

thus no objective (that is, the objective can simply be 0). The constraints are as follows:

$$\sum_{j=1}^n x_{ij} = R_i, \quad i = 1, \dots, n, \quad (1)$$

$$\sum_{i=1}^n x_{ij} = C_j, \quad j = 1, \dots, n, \quad (2)$$

$$x_{ij} \geq x_{kl}, \quad (i, j, k, l) \in D, \quad (3)$$

$$x_{ij} \in \{0, 1\}, \quad i, j = 1, \dots, n. \quad (4)$$

Constraints (1) and (2) take care of the required number of filled cells in each row and column, respectively. Then constraints (3) ensure that a cell (k, l) cannot be filled if cell (i, j) below (k, l) in the same thermometer is not filled. Finally, constraints (4) define the decision variables to be binary.

3. Kakuro (Cross Sums)

Kakuro, also called Cross Sums, is a classical paper-and-pencil puzzle that has been around for decades. Several apps are available as well. We consider the app *Kakuro* (*Cross Sums*) displayed in Figure 2. The rules are as follows: We have an $n \times n$ grid in which the empty cells must be filled with digits (between 1 and 9). For each set of consecutive empty cells, horizontal or vertical, a value is given. The sum of the digits in these cells must equal the given value. Moreover, the digits involved in such a sum must be different.

We use the following notation. Each empty cell is called (i, j) , where i is the row and j is the column in the grid. The set of all empty cells is denoted C . Moreover, K is the number of sums in the grid (we need not distinguish between horizontal and vertical sums). For sum $k \in \{1, \dots, K\}$, we have the related set of cells R_k and the resulting value of the sum r_k . Consider the first

Figure 1. Smartphone App *Thermometer Puzzles*

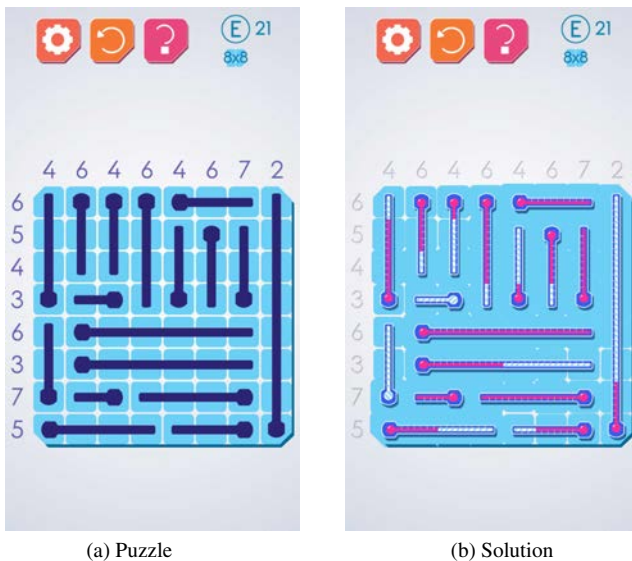
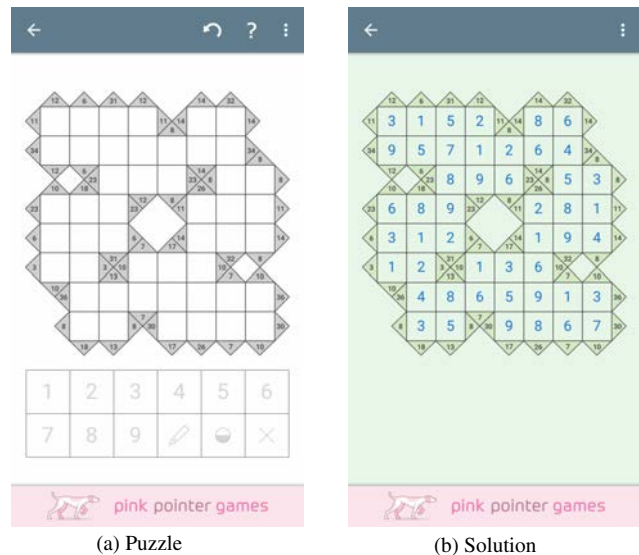


Figure 2. Smartphone App *Kakuro* (*Cross Sums*)



horizontal sum in the upper left corner of Figure 2 as an example. We have $k = 1$, $R_1 = \{(1, 1), (1, 2), (1, 3), (1, 4)\}$ and $r_1 = 11$. For notational convenience, we refer to the set of digits as D , thus $D = \{1, \dots, 9\}$.

Next, we define the decision variables. We have a binary decision variable for each cell $(i, j) \in C$ and each digit $d \in D$:

$$x_{ijd} = \begin{cases} 1, & \text{if digit } d \text{ is assigned to cell } (i, j), \\ 0, & \text{otherwise.} \end{cases}$$

We obtain the following mathematical formulation (again, there is no objective function):

$$\sum_{d \in D} x_{ijd} = 1, \quad (i, j) \in C, \quad (5)$$

$$\sum_{(i,j) \in R_k} x_{ijd} \leq 1, \quad k = 1, \dots, K; d \in D, \quad (6)$$

$$\sum_{(i,j) \in R_k} \sum_{d \in D} dx_{ijd} = r_k, \quad k = 1, \dots, K, \quad (7)$$

$$x_{ijd} \in \{0, 1\}, \quad (i, j) \in C; d \in D. \quad (8)$$

Constraints (5) ensure that each empty cell is assigned exactly one digit. Then constraints (6) take care of the rule that all digits belonging to a sum must be different, that is, each digit may occur at most once in a sum. According to constraints (7), the digits must add up to the required sum. Constraints (8) limit the decision variables to binary values.

4. Match 22 and ∞ Infinity Loop

In the app *Match 22: Color Puzzle Game*, several disks are given. Each disk is divided into four segments, and each segment is colored in yellow, orange, red or blue. The goal is to arrange the disks in such a way that the adjacent segments of any two neighboring disks have the same color. To achieve this, the player can turn the

disks. Every time he or she tips a disk, it turns left (i.e., counterclockwise) by 90 degrees. The moves (tips) are counted, so one would want to match all colors with the smallest possible number of moves. Figure 3 shows an example level along with the solution.

To develop a mathematical model, we label the disks using $i = 1, \dots, n$. For any two horizontally neighboring disks i and k , we introduce an ordered pair (i, k) which means that disk k is the immediate neighbor to the right of disk i . The set of all such pairs is called H . Analogously, V is the set of all pairs of vertical neighbors.

Next, we transform the four colors into numbers; we choose 1 for yellow, 2 for orange, 3 for red, and 4 for blue. The four segments of a disk are labeled 0, 1, 2, and 3 according to Figure 3 (c). We can see that for horizontal neighbors $(i, k) \in H$, the color (i.e., number) of segment 3 of disk i must match the color (number) of segment 1 of disk k . Similarly, for vertical neighbors $(i, k) \in V$, the color (number) of segment 2 of disk i must match the color (number) of segment 0 of disk k .

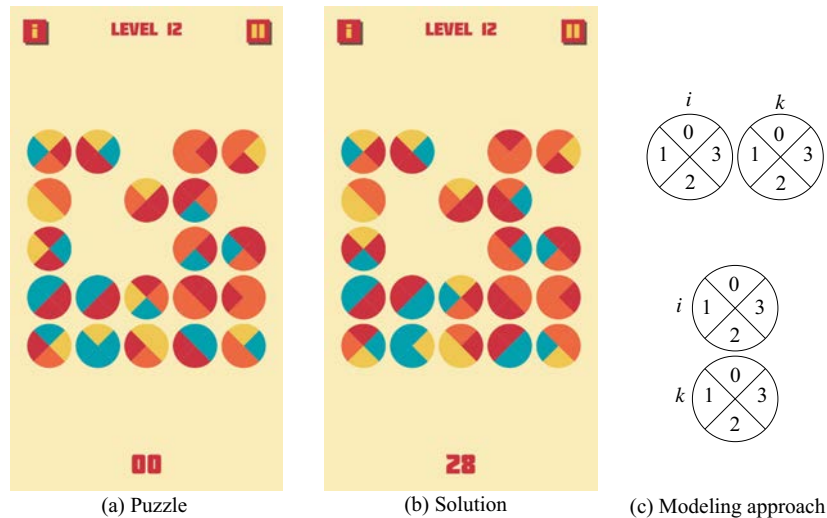
The colors of the disks in the initial setting are captured by parameters d_{ih} , where i is the number of the disk and h is the segment index according to Figure 3(c). Consider the upper left disk in Figure 3(a), referred to as disk 1. Segment 0 is yellow and segment 3 is red, so we have $d_{10} = 1$ and $d_{13} = 3$. In the model below, turning a disk essentially corresponds to transforming segment index h .

The binary decision variables specify for each disk $i = 1, \dots, n$ how many times it must be turned left by 90 degrees.

$$x_{ij} = \begin{cases} 1, & \text{if } j \text{ left-turns should be applied to disk } i, \\ 0, & \text{otherwise.} \end{cases}$$

Of course, between 0 and 3 such left turns can be applied to each disk; thus we have $j \in \{0, \dots, 3\}$. We can

Figure 3. Smartphone App *Match 22: Color Puzzle Game*



now provide a mathematical programming formulation. In contrast to the previous puzzle, this model actually includes an objective. Note also that the model contains a constant for the maximum number of turns. This is because the number of segments of a disk is fixed by the design of the game.

$$\text{Minimize } \sum_{i=1}^n \sum_{j=0}^3 jx_{ij} \quad (9)$$

$$\text{subject to } \sum_{j=0}^3 x_{ij} = 1, \quad i = 1, \dots, n, \quad (10)$$

$$\sum_{j=0}^3 d_{i,(3-j) \bmod 4} \cdot x_{ij} = \sum_{j=0}^3 d_{k,(1-j) \bmod 4} \cdot x_{kj}, \quad (i, k) \in H, \quad (11)$$

$$\sum_{j=0}^3 d_{i,(2-j) \bmod 4} \cdot x_{ij} = \sum_{j=0}^3 d_{k,(0-j) \bmod 4} \cdot x_{kj}, \quad (i, k) \in V, \quad (12)$$

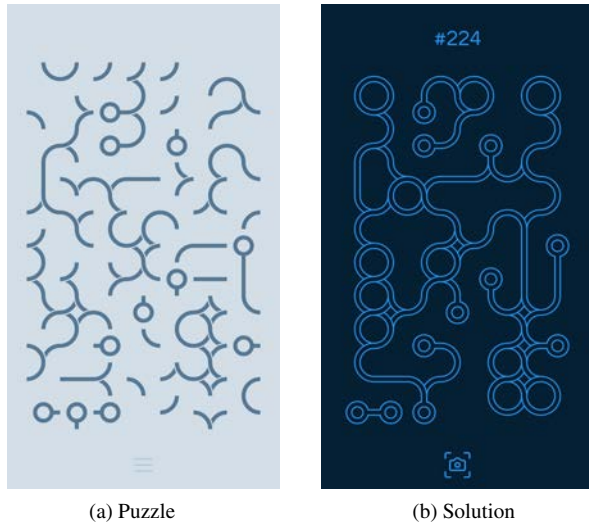
$$x_{ij} \in \{0, 1\} \quad i = 1, \dots, n; j = 0, \dots, 3. \quad (13)$$

Objective (9) minimizes the total number of moves. Constraints (10) ensure that every disk is assigned exactly one number of turns. Subsequently, constraints (11) and (12) observe that horizontally and vertically adjacent segments have the same color, respectively. Finally, constraints (13) restrict the decision variables to be binary.

The app ∞ *Infinity Loop* is based on a similar idea. As in *Match 22*, tiles must be turned to solve the puzzle. Here, however, each tile contains up to four connections (to the left, to the right, up, and down), and the tiles must be turned such that an overall pattern without any open connections is achieved. Figure 4 shows an example level as well as the related solution.

We can apply a rather straightforward extension of the mathematical model described above for *Match 22*

Figure 4. Smartphone App ∞ *Infinity Loop*



to solve ∞ *Infinity Loop*. Of course, each disk corresponds to a tile, but instead of colors, we just have to capture in the parameters whether there is a connection in segment h of tile i ($d_{ih} = 1$) or not ($d_{ih} = 0$). We use the same binary decision variables x_{ij} as above to describe the number of turns of each tile. To find a valid solution, we take into account an additional requirement that does not appear in *Match 22*. In ∞ *Infinity Loop* there is a restriction that the outer tiles must be turned in a way that no open connection points to the outside of the grid. To capture this, we define sets B_{top} , B_{left} , B_{bottom} , and B_{right} containing the border tiles at the top, on the left, at the bottom, and on the right, respectively. Note that, for example, the top left cell will be in B_{top} and B_{left} . Now we add the following constraints to take this restriction into account, and we obtain (9)–(17) as a mathematical model for ∞ *Infinity Loop*:

$$\sum_{j=0}^3 d_{i,(0-j) \bmod 4} \cdot x_{ij} = 0, \quad i \in B_{\text{top}}, \quad (14)$$

$$\sum_{j=0}^3 d_{i,(1-j) \bmod 4} \cdot x_{ij} = 0, \quad i \in B_{\text{left}}, \quad (15)$$

$$\sum_{j=0}^3 d_{i,(2-j) \bmod 4} \cdot x_{ij} = 0, \quad i \in B_{\text{bottom}}, \quad (16)$$

$$\sum_{j=0}^3 d_{i,(3-j) \bmod 4} \cdot x_{ij} = 0, \quad i \in B_{\text{right}}. \quad (17)$$

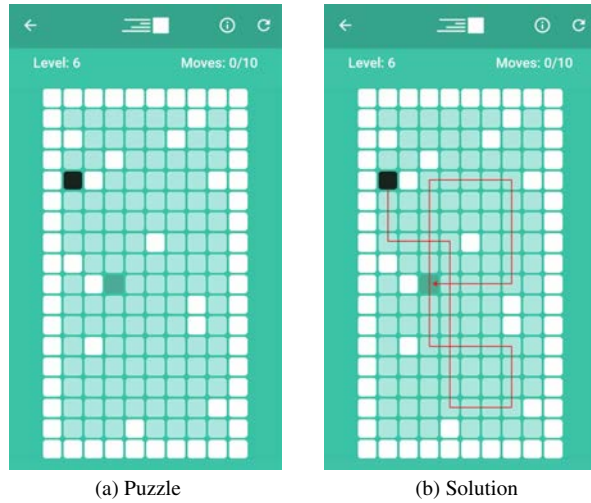
5. Slider

In the *Slider* app, a rectangle consisting of rows and columns is given. One of the cells is the starting point. It contains a black block that must be moved to another cell, the destination marked by a gray cell (see Figure 5); the white cells are obstacles.

The player has to wipe across the smartphone display to move the block. It can be moved left or right within the row or up or down within the column. It is not possible to move the block by just a few cells. When moved in a direction, the block will always stop at the last cell of the rectangle in the respective row or column, or at the cell immediately before an obstacle. The player solves the puzzle if he or she can move the block to the target cell. Beyond that, the goal is to accomplish this with the minimum number of moves.

Let m be the number of rows of the rectangle, and let n be the number of columns (we do not include the white cells on the border, see Figure 5). Then we denote the starting cell by (i_s, j_s) and the destination cell by (i_t, j_t) . We can now model the slider puzzle as a standard shortest path problem by defining an appropriate directed graph $G = (V, E)$. The set of vertices or nodes is defined as $V = \{(i, j) \mid i = 1, \dots, m; j = 1, \dots, n\}$, that is, each cell is a node in the graph. The set E of edges is defined as follows: For each cell (i, j) , there is an arc to each cell (k, l) that can be reached from (i, j) in a single

Figure 5. Smartphone App *Slider*



move. The edges are then written as (i, j, k, l) . Let $S_{ij} = \{(k, l) \mid (i, j, k, l) \in E\}$ represent the set of successors of node (i, j) .

If there is no obstacle, an inner cell (i, j) with $1 < i < m$ and $1 < j < n$ has successors $(i, 1)$, (i, n) , $(1, j)$ and (m, j) . Of course, cells on the border are associated with fewer than four moves. In case there is an obstacle in cell, e.g., $(i, j + 3)$, then a move to the right would have to stop before that obstacle, hence the arc representing the move to the right would be $(i, j + 2)$ instead of (i, n) . If the obstacle is in $(i, j + 1)$, then there is no possible move to the right and thus no such arc. Finally, there are no edges leading out of the target cell (i_t, j_t) , and cells representing obstacles are not associated with any edges.

We need binary decision variables that determine which move should be made and thus which edges in the graph should be selected:

$$x_{ijkl} = \begin{cases} 1, & \text{if the move from cell } (i, j) \text{ to cell } (k, l) \\ & \text{should be made,} \\ 0, & \text{otherwise.} \end{cases}$$

Now we are ready to give the mathematical programming formulation:

$$\text{Minimize } \sum_{i=1}^m \sum_{j=1}^n \sum_{(k,l) \in S_{ij}} x_{ijkl} \quad (18)$$

$$\text{subject to } \sum_{(k,l) \in S_{i_s j_s}} x_{i_s j_s k l} = 1, \quad (19)$$

$$\sum_{\substack{(k,l) \in V \\ (i,j) \in S_{kl}}} x_{k l i j} = \sum_{(k,l) \in S_{ij}} x_{ijkl}, \quad (20)$$

$$(i, j) \in V \setminus \{(i_s, j_s), (i_t, j_t)\}, \quad (20)$$

$$x_{ijkl} \in \{0, 1\}, \quad i = 1, \dots, m; \quad j = 1, \dots, n; (k, l) \in S_{ij}. \quad (21)$$

Objective (18) minimizes the number of moves. Constraints (19) force the black block to leave the starting

cell, and constraints (20) are the usual flow constraints for all nodes except for the source and the destination. The variables are defined to be binary by constraints (21).

6. Flow Free

In the app *Flow Free*, a square grid is given. Some cells contain colored dots; there are always two dots of the same color. The player has to connect any two dots of the same color by a continuous line; see Figure 6. Several rules must be observed: A line always moves horizontally or vertically from cell to cell of the grid; diagonal lines are not permitted. All lines must start or end in a cell with a dot, and is not possible that a line enters a cell with a dot and leaves it again. Moreover, lines are not allowed to intersect, that is, each cell of the grid may be associated only with a single line. Each cell must be connected by a line (thus the solution of Figure 6(c) is invalid). Finally, each line must be connected to two dots, that is, additional lines are not possible. There are also other apps with similar puzzles.

We will model this puzzle as a network flow problem (which matches the name of the app). It can be viewed as a multi-commodity flow problem where each color is represented by a commodity and each pair of dots of the same color corresponds to the source and destination of a commodity. First, we define a directed graph $G = (V, E)$. Each cell of the $n \times n$ grid corresponds to a node which is referred to by (i, j) . The set of neighbor nodes of a node (i, j) is given by $N_{ij} = \{(i-1, j), (i+1, j), (i, j-1), (i, j+1)\} \cap V$. The set E of edges is then obtained by defining an edge (i, j, k, l) between any two neighbor nodes $(i, j) \in V$ and $(k, l) \in N_{ij}$.

Next, one dot of each color is defined as source node and the other as destination node. The sets of sources and destination nodes are denoted as S and T , respectively. We also have to take care of the colors that are transformed into values. The color of the first source and destination is 1, that of the second is 2, and so on. These color values are captured by parameters C_{ij} for every source and every destination node $(i, j) \in S \cup T$. We have Q different colors, $Q = |S|$.

Let us now define the decision variables. We consider binary variables that take into account the logical flow (i.e., to avoid intersecting lines and to ensure that every cell is connected) and the colors of the lines:

$$x_{qijkl} = \begin{cases} 1, & \text{if a line of color } q \text{ should be drawn} \\ & \text{from cell } (i, j) \text{ to cell } (k, l), \\ 0, & \text{otherwise.} \end{cases}$$

Moreover, we have to avoid cycles in the solution. Consider the four empty cells on the right of Figure 6(c). The flow constraints could be fulfilled by creating a cycle through these four cells, but this would not be allowed by the rules. To solve this issue, we introduce variables z_{ij} for all nodes $(i, j) \in V$.

Figure 6. Smartphone app *Flow Free*

The related constraints are explained below. Now we are ready to present the model:

$$\sum_{q=1}^Q \sum_{(k,l) \in N_{ij}} x_{qkl ij} = 0, \quad (i,j) \in S, \quad (22)$$

$$\sum_{q=1}^Q \sum_{(k,l) \in N_{ij}} x_{qij kl} = 0, \quad (i,j) \in T, \quad (23)$$

$$\sum_{q=1}^Q \sum_{(k,l) \in N_{ij}} x_{qkl ij} = 1, \quad (i,j) \in V \setminus S, \quad (24)$$

$$\sum_{q=1}^Q \sum_{(k,l) \in N_{ij}} x_{qij kl} = 1, \quad (i,j) \in V \setminus T, \quad (25)$$

$$\sum_{q=1}^Q \sum_{(k,l) \in N_{ij}} q x_{qij kl} = C_{ij}, \quad (i,j) \in S, \quad (26)$$

$$\sum_{q=1}^Q \sum_{(k,l) \in N_{ij}} q x_{qkl ij} = C_{ij}, \quad (i,j) \in T, \quad (27)$$

$$\sum_{q=1}^Q \sum_{(k,l) \in N_{ij}} q x_{qkl ij} = \sum_{q=1}^Q \sum_{(k,l) \in N_{ij}} q x_{qij kl}, \quad d(i,j) \in V \setminus (S \cup T), \quad (28)$$

$$z_{kl} - z_{ij} + M \left(1 - \sum_{q=1}^Q x_{qij kl} \right) \geq 1, \quad (i,j) \in V, (k,l) \in N_{ij}, \quad (29)$$

$$x_{qij kl} \in \{0,1\}, \quad q=1, \dots, Q, (i,j) \in V, (k,l) \in N_{ij}, \quad (30)$$

$$z_{ij} \geq 0, \quad (i,j) \in V. \quad (31)$$

Constraints (22) and (23) ensure that there is no flow into a source and out of a destination node, respectively. Then constraints (24) imply that there is a flow of 1 into each node that is not a source. Analogously, constraints (25) enforce a flow of 1 out of each node that is not a destination node. Note that these two constraints also imply that there is a flow through each node and that flows do not intersect. Next, constraints (26) and

(27) indicate that there is a flow out of each source and into each destination, respectively, that corresponds to the related color value. The usual flow constraints are given by (28). They ensure that the color that goes into a node will also go out of it.

Moreover, constraints (29) link the x and the z variables. They state that each node must be associated with a z value that is larger than that of its predecessor. This way, cycles are made impossible, and every flow must be connected to a source and a destination node. Of course, these are the subtour elimination constraints proposed by Miller et al. (1960) for the TSP. M is a large number; note that we can set $M = n^2$. The last two constraints (30) and (31) define the decision variables.

It might be interesting to study another possible modeling approach for the *Flow Free* app. In the model above, we have captured both the logical flow in the color flow by binary decision variables. The idea is to split this: Now the logical flow (i.e., the decision whether there is a line from one cell to another) is taken care of by binary decision variables related to the edges:

$$x_{ijkl} = \begin{cases} 1, & \text{if a line should be drawn from cell } (i,j) \\ & \text{to cell } (k,l), \\ 0, & \text{otherwise.} \end{cases}$$

The color of the flow (i.e., the color of the line) from a cell (i,j) to a cell (k,l) is now captured by variables y_{ijkl} which do not have to be binary or integer. That is, while variables y_{ijkl} take the color of the flow into account, variables x_{ijkl} are needed to ensure that each cell is connected and that flows (lines) do not intersect. Finally, we again need decision variables z_{ij} for all nodes $(i,j) \in V$ that help to avoid cycles in the solution. The resulting model can be stated as follows:

$$\sum_{(k,l) \in N_{ij}} x_{kl ij} = 0, \quad (i,j) \in S, \quad (32)$$

$$\sum_{(k,l) \in N_{ij}} x_{ij kl} = 0, \quad (i,j) \in T, \quad (33)$$

$$\sum_{(k,l) \in N_{ij}} x_{klij} = 1, \quad (i, j) \in V \setminus S, \quad (34)$$

$$\sum_{(k,l) \in N_{ij}} x_{ijkl} = 1, \quad (i, j) \in V \setminus T, \quad (35)$$

$$\sum_{(k,l) \in N_{ij}} y_{ijkl} = C_{ij}, \quad (i, j) \in S, \quad (36)$$

$$\sum_{(k,l) \in N_{ij}} y_{klij} = C_{ij}, \quad (i, j) \in T, \quad (37)$$

$$\sum_{(k,l) \in N_{ij}} y_{klij} = \sum_{(k,l) \in N_{ij}} y_{ijkl}, \quad (i, j) \in V \setminus (S \cup T), \quad (38)$$

$$y_{ijkl} \geq x_{ijkl}, \quad (i, j) \in V, (k, l) \in N_{ij}, \quad (39)$$

$$y_{ijkl} \leq M_1 x_{ijkl}, \quad (i, j) \in V, (k, l) \in N_{ij}, \quad (40)$$

$$z_{kl} - z_{ij} + M_2(1 - x_{ijkl}) \geq 1, \quad (i, j) \in V, (k, l) \in N_{ij}, \quad (41)$$

$$x_{ijkl} \in \{0, 1\}, \quad (i, j) \in V, (k, l) \in N_{ij}, \quad (42)$$

$$y_{ijkl} \geq 0, \quad (i, j) \in V, (k, l) \in N_{ij}, \quad (43)$$

$$z_{ij} \geq 0, \quad (i, j) \in V. \quad (44)$$

The first four constraints (32)–(35) represent the logical flow. They are analogous to constraints (22)–(25). The colors are now observed by the y variables. Constraints (36), (37), and (38) take care of the colors leaving the sources, entering the destinations, and flowing through the remaining nodes, respectively.

Next, constraints (39) and (40) link the x and the y variables. They say that there is a flow on an edge according to the associated y variable if and only if there is a flow according to the associated x variable. Parameter M_1 is a large number; $M_1 = Q$ is a natural choice if the colors of the flows are transformed into values $1, 2, \dots, Q$. Then constraints (41) link the x and the z variables. Similar to constraints (29) of the first model, they ensure that no cycles occur (analogously to (29), M_2 is a large number that can be set to $M_2 = n^2$). Finally, the remaining three constraints (42)–(44) define the decision variables.

The two models presented for the *Flow Free* app can be used to discuss the impact of modeling techniques. In the first approach, the logical flow and the colors are captured by binary variables. By contrast, the second approach uses binary variables only for the logical flow while continuous variables are used for the colors. Clearly, the first model contains $Q \cdot n^4$ binary variables whereas the second contains only n^4 .

Often models with fewer binary variables lead to shorter solution times when solvers are applied. This is also the case here: We solved the example of Figure 6(a) using the solver of the GNU Linear Programming Kit (GLPK), see Makhorin (2010), on a computer with an Intel i5 CPU running at 3.0 GHz. The solution time of the first model (22)–(31) was 324.7 seconds while the solution time of the second model (32)–(44) was only 0.4 seconds.

7. Conclusions

In this paper, we have presented six smartphone apps that provide different logic puzzles. We developed integer (and mixed integer) programming models that can be applied to determine solutions for these puzzles.

The puzzle apps and models are used in the introductory OR/MS lectures in BSc management programs. Considering this, our particular focus was apps that allow for models with a difficulty level between easy and moderate. Some puzzles (usually *Thermometer Puzzles* and *Kakuro*) are given to the students as additional modeling exercises that can be done at home. In evaluations of the lectures, the exercises generally get positive feedback. While this applies to the exercises in general and not only to the puzzle exercises in particular, our impression is that most students actually do the puzzle exercises, and that many find them interesting and motivating. The other puzzles are used for classroom discussions whenever there is enough time. Questions such as the following are discussed: What would the variables/the underlying graph/the constraints look like? What are the similarities to other models; what ideas from other models can be used here again? We find these kinds of discussions helpful to teach IP modeling because they support a deeper understanding that is often not achieved if students try to learn given models by heart. In the discussions, the puzzles are also used to point out that mathematical models and solvers are actually very general tools that can solve a broad variety of problems. Thus, puzzle-based exercises and discussions have become a regular part of our introductory OR/MS lectures.

Models and example data implemented in MathProg (Makhorin (2010)) can be found in the appendices. Note also that all instances provided in the appendices could be solved in less than 0.5 seconds. Exceptions were the *Kakuro* instance of Figure 2, which required 4.5 seconds, and the first model for the *Flow Free* instance of Figure 6, which required Figure 324.7 seconds (all tests were carried out on a computer with an Intel i5 CPU running at 3.0 GHz).

While we currently use the apps only for teaching mathematical modeling, it should be noted that some of them might also be used as examples when teaching other OR techniques such as branch-and-bound or constraint programming.

Acknowledgments

The author thanks the developers of the apps for their kind permissions to reproduce the screenshots. *Thermometer Puzzles*: Francois Guibert (Frozax Games, France); *Kakuro* (*Cross Sums*): Pink Pointer Games (Brazil); *Match 22*: Abhinav Bhadoria (Zero Logic Games, India); ∞ *Infinity Loop*: Muhammad Satar (Webavenue, Portugal) and Jonas Lekevicius and Balys Valentukevicius (Lithuania); *Slider*: Alex Gwyn (USA); *Flow Free*: Sharon Newman (Big Duck Games LLC, USA).

Appendix A. MathProg File for Thermometer Puzzles

The code given below corresponds to the model introduced in Section 2. The data part captures the example displayed in Figure 1.

```

param n integer;
param R{1..n} integer;
param C{1..n} integer;
set D within {1..n} cross {1..n} cross {1..n} cross {1..n};

var x{1..n,1..n} binary;

maximize F: 0;

s.t. RowCount {i in 1..n}: sum{j in 1..n} x[i,j] = R[i];
s.t. ColCount {j in 1..n}: sum{i in 1..n} x[i,j] = C[j];
s.t. Dependencies {(i,j,k,l) in D}: x[i,j] >= x[k,l];

solve;

printf"\nSolution:\n\n";
for {i in 1..n}
{
    printf{j in 1..n} x[i,j];
    printf"\n";
}
printf"\n";

data;

param n := 8;

param : R C :=
1      6  4
2      5  6
3      4  4
4      3  6
5      6  4
6      3  6
7      7  7
8      5  2 ;

set D := (1,5,1,6) (1,6,1,7)
(4,3,4,2)
(5,2,5,3) (5,3,5,4) (5,4,5,5) (5,5,5,6) (5,6,5,7)
(6,2,6,3) (6,3,6,4) (6,4,6,5) (6,5,6,6) (6,6,6,7)
(7,3,7,2) (7,7,7,6) (7,6,7,5) (7,5,7,4)
(8,1,8,2) (8,2,8,3) (8,3,8,4) (8,7,8,6) (8,6,8,5)
(4,1,3,1) (3,1,2,1) (2,1,1,1) (7,1,6,1) (6,1,5,1)
(1,2,2,2) (2,2,3,2)
(1,3,2,3) (2,3,3,3)
(1,4,2,4) (2,4,3,4) (3,4,4,4)
(4,5,3,5) (3,5,2,5)
(2,6,3,6) (3,6,4,6)
(4,7,3,7) (3,7,2,7)
(8,8,7,8) (7,8,6,8) (6,8,5,8) (5,8,4,8) (4,8,3,8) (3,8,2,8) (2,8,1,8) ;

end;
```

Appendix B. MathProg File for Kakuro (Cross Sums)

Below, we give the MathProg code resulting from the model presented in Section 3. The data corresponds to the example given in Figure 2.

```

param n integer;
param K integer;
param r{1..K} integer;
set R{1..K} within {1..n} cross {1..n};
set C within {1..n} cross {1..n};

var x{(i,j) in C, 1..9} binary;
```

```
maximize F: 0;

s.t. Digit      {(i,j) in C}:      sum{d in 1..9}      x[i,j,d]   = 1;
s.t. Different  {k in 1..K, d in 1..9}: sum{(i,j) in R[k]} x[i,j,d] <= 1;
s.t. Sum        {k in 1..K}:      sum{(i,j) in R[k], d in 1..9} d * x[i,j,d] = r[k];

solve;

printf"\nSolution:\n\n";
for {i in 1..n}
{
    for {j in 1..n} printf if (i,j) in C then sum{d in 1..9} d * x[i,j,d] else " ";
    printf"\n";
}
printf"\n";

data;

param n := 8;
param K := 30;

param r :=
1 11
2 14
3 34
4 23
5 8
6 23
7 11
8 6
9 14
10 3
11 10
12 36
13 8
14 30
15 12
16 10
17 6
18 18
19 31
20 13
21 12
22 7
23 8
24 17
25 14
26 26
27 32
28 7
29 8
30 10 ;

set R[ 1] := (1,1) (1,2) (1,3) (1,4);
set R[ 2] := (1,6) (1,7);
set R[ 3] := (2,1) (2,2) (2,3) (2,4) (2,5) (2,6) (2,7);
set R[ 4] := (3,3) (3,4) (3,5);
set R[ 5] := (3,7) (3,8);
set R[ 6] := (4,1) (4,2) (4,3);
set R[ 7] := (4,6) (4,7) (4,8);
set R[ 8] := (5,1) (5,2) (5,3);
set R[ 9] := (5,6) (5,7) (5,8);
set R[10] := (6,1) (6,2);
set R[11] := (6,4) (6,5) (6,6);
set R[12] := (7,2) (7,3) (7,4) (7,5) (7,6) (7,7) (7,8);
set R[13] := (8,2) (8,3);
set R[14] := (8,5) (8,6) (8,7) (8,8);
set R[15] := (1,1) (2,1);
set R[16] := (4,1) (5,1) (6,1);
set R[17] := (1,2) (2,2);
```

```

set R[18] := (4,2) (5,2) (6,2) (7,2) (8,2);
set R[19] := (1,3) (2,3) (3,3) (4,3) (5,3);
set R[20] := (7,3) (8,3);
set R[21] := (1,4) (2,4) (3,4);
set R[22] := (6,4) (7,4);
set R[23] := (2,5) (3,5);
set R[24] := (6,5) (7,5) (8,5);
set R[25] := (1,6) (2,6);
set R[26] := (4,6) (5,6) (6,6) (7,6) (8,6);
set R[27] := (1,7) (2,7) (3,7) (4,7) (5,7);
set R[28] := (7,7) (8,7);
set R[29] := (3,8) (4,8) (5,8);
set R[30] := (7,8) (8,8);

```

```

set C : 1 2 3 4 5 6 7 8 :=
1      + + + + - + + -
2      + + + + + + -
3      - - + + - + +
4      + + + - - + +
5      + + + - - + +
6      + + - + + + -
7      - + + + + + +
8      - + + - + + + ;

```

```
end;
```

This MathProg Code will print the following solution (compare to Figure 2(b)):

Solution:

```

3152 86
9571264
 896 53
689 281
312 194
12 136
4865913
35 9867

```

Appendix C. MathProg File for Match 22

The following code corresponds to the model of Section 4. The data corresponds to the example given in Figure 3.

```

param n integer;
set H within {1..n} cross {1..n};
set V within {1..n} cross {1..n};
param d{1..n, 0..3};

var x{1..n,0..3} binary;

minimize F: sum{i in 1..n, j in 0..3} j * x[i,j];

s.t. OneOrientation{i in 1..n}:
    sum{j in 0..3} x[i,j] = 1;
s.t. Horizontal {(i,k) in H}:
    sum{j in 0..3} d[i,(3-j) mod 4] * x[i,j] = sum{j in 0..3} d[k,(1-j) mod 4] * x[k,j];
s.t. Vertical {(i,k) in V}:
    sum{j in 0..3} d[i,(2-j) mod 4] * x[i,j] = sum{j in 0..3} d[k,(0-j) mod 4] * x[k,j];

solve;

printf"\Solution:\n\n";
for {i in 1..n}
    printf"Disk %2d: %d left turn(s)\n", i, sum{j in 0..3} j*x[i,j];
printf"\n%d left turns in total\n\n", sum{i in 1..n, j in 0..3} j*x[i,j];

data;

param n := 20;

set H := (1,2) (3,4) (6,7) (9,10) (11,12) (12,13) (13,14)
        (14,15) (16,17) (17,18) (18,19) (19,20) ;

```

```
set V := (1,5) (3,7) (5,8) (7,9) (8,11) (9,14) (10,15)
        (11,16) (12,17) (13,18) (14,19) (15,20) ;
```

```
param d : 0 1 2 3 :=
1      1 4 2 3
2      1 3 3 4
3      2 2 2 3
4      2 2 3 1
5      2 1 1 2
6      1 2 3 3
7      3 3 4 2
8      3 1 3 4
9      2 2 4 3
10     3 4 2 3
11     4 4 3 3
12     4 4 3 3
13     3 1 4 2
14     3 2 2 3
15     2 3 2 2
16     4 3 2 1
17     1 4 4 4
18     1 3 2 1
19     4 3 3 4
20     1 2 2 4 ;
```

end;

This MathProg file leads to the output below, which corresponds to the solution shown in Figure 3(b):

Solution:

```
Disk 1: 0 left turn(s)
Disk 2: 0 left turn(s)
Disk 3: 1 left turn(s)
Disk 4: 0 left turn(s)
Disk 5: 0 left turn(s)
Disk 6: 0 left turn(s)
Disk 7: 1 left turn(s)
Disk 8: 3 left turn(s)
Disk 9: 1 left turn(s)
Disk 10: 0 left turn(s)
Disk 11: 0 left turn(s)
Disk 12: 2 left turn(s)
Disk 13: 3 left turn(s)
Disk 14: 2 left turn(s)
Disk 15: 2 left turn(s)
Disk 16: 3 left turn(s)
Disk 17: 3 left turn(s)
Disk 18: 2 left turn(s)
Disk 19: 3 left turn(s)
Disk 20: 2 left turn(s)
```

28 left turns in total

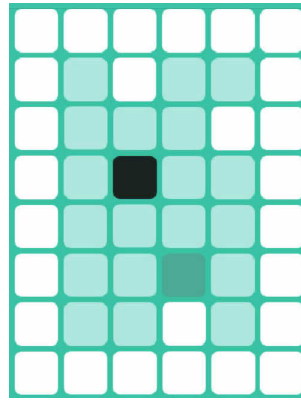
Appendix D. MathProg File for Slider

The following MathProg code contains the model of Section 5. The data corresponds to a very small example shown in Figure D.1. Note that this is not taken from the Slider app: The original levels of the app would have led to much larger data in the MathProg file. Note also that the border cells are not part of the data.

```
param m integer;
param n integer;
param i_s integer;
param j_s integer;
param i_t integer;
param j_t integer;
set ST within {1..m} cross {1..n} := {(i_s,j_s), (i_t,j_t)};
set S{i in 1..m, j in 1..n} within {1..m} cross {1..n};

var x{i in 1..m, j in 1..n, k in 1..m, l in 1..n : (k,l) in S[i,j]} binary;

minimize F: sum{i in 1..m, j in 1..n} sum{k in 1..m, l in 1..n : (k,l) in S[i,j]} x[i,j,k,l];
```


Figure D.1. Small example based on *Slider*

```

s.t. Flow{i in 1..m, j in 1..n : (i,j) not in ST}:
    sum{k in 1..m, l in 1..n : (i,j) in S[k,l]} x[k,l,i,j]
    = sum{k in 1..m, l in 1..n : (k,l) in S[i,j]} x[i,j,k,l];
s.t. Source:
    sum{k in 1..m, l in 1..n : (k,l) in S[i_s,j_s]} x[i_s,j_s,k,l] = 1;

solve;

printf"\nSolution (moves may be given in arbitrary order):\n\n";
for {i in 1..m}
    for {j in 1..n}
        printf {(k,l) in S[i,j] : x[i,j,k,l] = 1} : "(\%d,\%d) --> (\%d,\%d)\n", i, j, k, l;
printf"\n";

data;

param m := 6;
param n := 4;

param i_s := 3;
param j_s := 2;
param i_t := 5;
param j_t := 3;

set S[1,1] := (6,1) ;
set S[2,1] := (1,1) (6,1) (2,3) ;
set S[3,1] := (1,1) (6,1) (3,4) ;
set S[4,1] := (1,1) (6,1) (4,4) ;
set S[5,1] := (1,1) (6,1) (5,4) ;
set S[6,1] := (1,1) (6,1) (6,2) ;

set S[1,2] := ;
set S[2,2] := (6,2) (2,1) (2,3) ;
set S[3,2] := (2,2) (6,2) (3,1) (3,4) ;
set S[4,2] := (2,2) (6,2) (4,1) (4,4) ;
set S[5,2] := (2,2) (6,2) (5,1) (5,4) ;
set S[6,2] := (2,2) (6,1) ;

set S[1,3] := (5,3) (1,4) ;
set S[2,3] := (1,3) (5,3) (2,1) ;
set S[3,3] := (1,3) (5,3) (3,1) (3,4) ;
set S[4,3] := (1,3) (5,3) (4,1) (4,4) ;
set S[5,3] := ;
set S[6,3] := ;

set S[1,4] := (1,3) ;
set S[2,4] := ;
set S[3,4] := (6,4) (3,1) ;
set S[4,4] := (3,4) (6,4) (4,1) ;
set S[5,4] := (3,4) (6,4) (5,1) ;

```

```
set S[6,4] := (3,4) ;
```

```
end;
```

This MathProg file produces the following output:

Solution (moves may be given in arbitrary order):

```
(2,2) --> (2,3)
(2,3) --> (5,3)
(3,2) --> (2,2)
```

Appendix E. MathProg File for Flow Free

The MathProg code given below contains the first model of Section 6. The data section reflects the example of Figure 6 (a).

```
param n integer;
param Q integer;
set V := {1..n} cross {1..n};
set N{(i,j) in V} within V := {(i-1,j), (i+1,j), (i,j-1), (i,j+1)} inter {1..n} cross {1..n};
set S within {1..n} cross {1..n};
set T within {1..n} cross {1..n};
param C{1..n, 1..n} integer;

var x{q in 1..Q, (i,j) in V, (k,l) in N[i,j]} binary;
var z{(i,j) in V} >= 0;

minimize F: 0;

s.t. FlowInSources {(i,j) in S}: sum{q in 1..Q} sum{(k,l) in N[i,j]} x[q,k,l,i,j] = 0;
s.t. FlowOutSinks {(i,j) in T}: sum{q in 1..Q} sum{(k,l) in N[i,j]} x[q,i,j,k,l] = 0;
s.t. FlowInOther {(i,j) in V diff S}: sum{q in 1..Q} sum{(k,l) in N[i,j]} x[q,k,l,i,j] = 1;
s.t. FlowOutOther {(i,j) in V diff T}: sum{q in 1..Q} sum{(k,l) in N[i,j]} x[q,i,j,k,l] = 1;
s.t. FlowOutSources{(i,j) in S}: sum{q in 1..Q} sum{(k,l) in N[i,j]} q * x[q,i,j,k,l] = C[i,j];
s.t. FlowInSinks {(i,j) in T}: sum{q in 1..Q} sum{(k,l) in N[i,j]} q * x[q,k,l,i,j] = C[i,j];
s.t. Flow {(i,j) in V diff (S union T)}: sum{q in 1..Q} sum{(k,l) in N[i,j]} q * x[q,k,l,i,j]
= sum{q in 1..Q} sum{(k,l) in N[i,j]} q * x[q,i,j,k,l];
s.t. xz_Link {(i,j) in V, (k,l) in N[i,j]}: z[k,l] - z[i,j] + n * n * (1 - (sum{q in 1..Q} x[q,i,j,k,l]))
>= 1;

solve;

printf"\nSolution:\n\n";
for {i in 1..n}
{
    for {j in 1..n}
        printf "%d", if (i,j) in S then sum{q in 1..Q} sum{(k,l) in N[i,j]}
        q*x[q,i,j,k,l] else sum{q in 1..Q} sum{(k,l) in N[i,j]} q*x[q,k,l,i,j];
        printf"\n";
    }
printf"\n";

data;

param n := 8;
param Q := 7;
set S := (2,1) (2,2) (2,5) (2,6) (6,2) (8,1) (8,5);
set T := (2,3) (2,7) (5,3) (5,5) (6,8) (7,8) (8,8);

param C : 1 2 3 4 5 6 7 8 :=
1      0 0 0 0 0 0 0 0
2      1 2 1 0 3 4 2 0
3      0 0 0 0 0 0 0 0
4      0 0 0 0 0 0 0 0
5      0 0 3 0 4 0 0 0
6      0 5 0 0 0 0 0 5
7      0 0 0 0 0 0 0 6
8      6 0 0 0 7 0 0 7 ;

end;
```

The MathProg code for the second model for the *Flow Free* app is as follows:

```

param n integer;
set V := {1..n} cross {1..n};
set N{(i,j) in V} within V := {(i-1,j), (i+1,j), (i,j-1), (i,j+1)} inter {1..n} cross {1..n};
set S within {1..n} cross {1..n};
set T within {1..n} cross {1..n};
param C{1..n, 1..n} integer;

var x{(i,j) in V, (k,l) in N[i,j]} binary;
var y{(i,j) in V, (k,l) in N[i,j]} >= 0;
var z{(i,j) in V} >= 0;

minimize F: 0;

s.t. x_FlowInSources {(i,j) in S}: sum{(k,l) in N[i,j]} x[k,l,i,j] = 0;
s.t. x_FlowOutSinks {(i,j) in T}: sum{(k,l) in N[i,j]} x[i,j,k,l] = 0;
s.t. x_FlowInOther {(i,j) in V diff S}: sum{(k,l) in N[i,j]} x[k,l,i,j] = 1;
s.t. x_FlowOutOther {(i,j) in V diff T}: sum{(k,l) in N[i,j]} x[i,j,k,l] = 1;
s.t. y_FlowOutSources{(i,j) in S}: sum{(k,l) in N[i,j]} y[i,j,k,l] = C[i,j];
s.t. y_FlowInSinks {(i,j) in T}: sum{(k,l) in N[i,j]} y[k,l,i,j] = C[i,j];
s.t. y_Flow {(i,j) in V diff (S union T)}: sum{(k,l) in N[i,j]} y[k,l,i,j] = sum{(k,l) in N[i,j]} y[i,j,k,l];
s.t. xy_Link1 {(i,j) in V, (k,l) in N[i,j]}: y[i,j,k,l] >= x[i,j,k,l];
s.t. xy_Link2 {(i,j) in V, (k,l) in N[i,j]}: y[i,j,k,l] <= card(S) * x[i,j,k,l];
s.t. xz_Link {(i,j) in V, (k,l) in N[i,j]}: z[k,l] - z[i,j] + n*(1-x[i,j,k,l]) >= 1;

solve;

printf"\nSolution:\n\n";
for {i in 1..n}
{
  for {j in 1..n}
    printf "%d", if (i,j) in S then sum{(k,l) in N[i,j]} y[i,j,k,l] else sum{(k,l) in N[i,j]} y[k,l,i,j];
  printf"\n";
}
printf"\n";

data;

param n := 8;
set S := (2,1) (2,2) (2,5) (2,6) (6,2) (8,1) (8,5);
set T := (2,3) (2,7) (5,3) (5,5) (6,8) (7,8) (8,8);

param C : 1 2 3 4 5 6 7 8 :=
1      0 0 0 0 0 0 0 0
2      1 2 1 0 3 4 2 0
3      0 0 0 0 0 0 0 0
4      0 0 0 0 0 0 0 0
5      0 0 3 0 4 0 0 0
6      0 5 0 0 0 0 0 5
7      0 0 0 0 0 0 0 6
8      6 0 0 0 7 0 0 7 ;

end;

```

This will produce the following output, which corresponds to the solution of Figure 6 (b):

Solution:

```

11122222
12123422
22223444
22233444
22334455
55555555
55566666
66667777

```

References

- Chlond MJ (2005) Classroom exercises in IP modeling: Su doku and the log pile. *INFORMS Trans. Ed.* 5(2):77–79.
- Chlond MJ (2014) Logic grid puzzles. *INFORMS Trans. Ed.* 15(1):166–168.
- Chlond MJ, Toase CM (2002) IP modeling of chessboard placements and related puzzles. *INFORMS Trans. Ed.* 2(2):1–11.
- DePuy GW, Taylor GD (2007) Using board puzzles to teach operations research. *INFORMS Trans. Ed.* 7(2):160–171.
- Letavec C, Ruggiero J (2002) The n -queens problem. *INFORMS Trans. Ed.* 2(3):101–103.
- Makhorin A (2010) Modeling language GNU MathProg. Language reference for GLPK version 4.45.
- Miller CE, Tucker AW, Zemlin RA (1960) Integer programming formulation of traveling salesman problems. *J. ACM* 7(4):326–329.
- Pearce RH, Forbes MA (2017) The Fillomino puzzle. *INFORMS Trans. Ed.* 17(2):85–89.
- Yeomans JS (2003) Solving “Einstein’s riddle” using spreadsheet optimization. *INFORMS Trans. Ed.* 3(2):55–63.