American Journal of Operations Research, 2016, 6, 213-225
Published Online May 2016 in SciRes. http://www.scirp.org/journal/ajor
http://dx.doi.org/10.4236/ajor.2016.63022

An Integer Programming Model for the KenKen Problem

Vardges Melkonian

Department of Mathematics, Ohio University, Athens, Ohio, USA
Email: melkonia@ohio.edu

Received 21 March 2016; accepted 7 May 2016; published 10 May 2016

Abstract

In this paper we consider modeling techniques for the mathematical puzzle KenKen. It is an interesting puzzle from modeling point of view since it has different kinds of mathematical restrictions that are not trivial to express as linear constraints. We give an integer program for solving KenKen and its implementation on modeling language AMPL. Our integer program uses prime number factorizations for converting product restrictions into linear constraints. It can be also used for teaching various integer programming techniques in an Operations Research course.

Keywords

Discrete Optimization, Integer Programming, Education Operations Research, Mathematical Games

1. Introduction

A KenKen puzzle is a grid of *n* by *n* cells (see **Figure 1**). The goal is to fill the whole grid with numbers 1 to n, making sure no number is repeated in any row and column. An additional feature of KenKen (compared to a similar puzzle Sudoku) is that the grid is partitioned into "cages". Each cage consists of several adjacent cells. The top left corner of each cage has a target number and an arithmetic operation (sum, difference, product, ratio). The numbers entered into a cage must combine (in any order) to produce the target number using the arithmetic operation. An 8 by 8 example of KenKen is given in **Figure 1**. This is a real example from KenKen website [1].

Among similar puzzles, KenKen is particularly interesting for mathematicians. Thanks to its mathematical constraints, it creates a different level of interest and challenge for the solver. It is also a more challenging task to create a mathematical model that can solve the puzzle.

While Sudoku has been studied extensively, not much research has been done on KenKen. [2] shows how ideas from number theory can be used to solve KenKen. [3] discusses how KenKen can be used to develop rea-

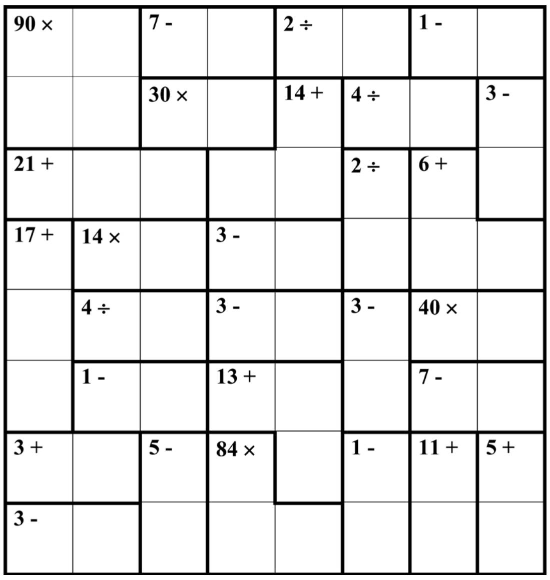

Figure 1. An example of 8 × 8 KenKen.

soning skills for different levels of students. In this paper we give an integer programming model for solving KenKen. The Latin square constraints were given before for Sudoku [4] [5]. Our contribution is giving linear constraints for sum, difference, product, and ratio restrictions. The product restrictions are the hardest ones for expressing by linear constraints. We give a non-standard way of using prime number factorizations for product constraints. The other parts of the model use different integer programming techniques, such as Either-Or constraints, converting absolute values to linear constraints, using auxiliary binary variables. Thus, the model could be an example for teaching different integer programming techniques.

We implemented the model on optimization modeling language AMPL [6] and tested on examples. The implementation requires different data structures needed for the model, such as prime number factorization of a number. The AMPL techniques also can be a good teaching tool on how to implement an integer programming model with an optimization software.

The paper is organized as follows. Section 2 gives the set of constraints that make the solution a Latin square. Sections 3, 4, 5, 6 cover sum, difference, ratio, product constraints correspondingly. Section 7 gives the AMPL model and an analysis of the solution. An Appendix section gives more details on a secondary method for converting product restrictions to linear constraints.

2. Latin Square Constraints

First we need to give constraints to provide that the numbers filling the grid form a Latin square, that is, each number occurs exactly once in each row and exactly once in each column. This type of constraints was given before for Sudoku puzzles (reference). The following binary variables are essential for giving those constraints.

$$x_{ijk} = \begin{cases} 1 & \text{if cell } (i, j) \text{ is assigned value } k \\ 0 & \text{otherwise} \end{cases}$$

The following set of constraints provide that each cell (i, j) gets exactly one value k from $1,\cdots,n$.

$$\sum_{k=1}^{n} x_{ijk} = 1 \quad \text{for each cell } (i,\ j) \tag{2.1}$$

The following set of constraints provides that there is exactly one number k in each row i:

$$\sum_{j=1}^{n} x_{ijk} = 1 \quad \text{for each row } i \text{ and each number } k \text{ from } 1,\cdots,n \tag{2.2}$$

The following set of constraints provides that there is exactly one number k in each column j:

$$\sum_{i=1}^{n} x_{ijk} = 1 \text{ for each column } j \text{ and each number } k \text{ from } 1,\cdots,n \tag{2.3}$$

3. Addition Constraints

While the binary variables defined are useful for giving the Latin square constraints, there is not a good way of using them to express the arithmetic restrictions by linear constraints. As we will see below, it is more helpful to define the following general integer variables for those constraints.

Let y_{ij} be the numerical value assigned to cell (*i*, *j*). The possible values to be assigned to y_{ij} are numbers 1 to *n*. This new set of variables can be easily used to express the summation restrictions by linear constraints. For a cage *C* having a target number *t* and arithmetic operation "+", the constraint is

$$\sum_{(i,j)\in C} y_{ij} = t \tag{3.1}$$

For example, the cage consisting of cells (3, 1), (3, 2) and (3, 3) in **Figure 1** needs the following constraint: $y_{31} + y_{32} + y_{33} = 21$.

The new integer variables should be connected to the original binary variables. For each cell (*i*, *j*), the connection is given by the following constraint.

$$\sum_{k=1}^{n} k \cdot x_{ijk} = y_{ij} \tag{3.2}$$

The constraints (3.1) and (3.2) together provide that y_{ij} takes the value *k* for which the corresponding variable x_{ijk} is equal to 1.

Note that constraint (3.2) is an example of an integer programming technique to provide that a variable takes one of the given values, 1 to *n* in this case.

Constraint (3.2) provides that y_{ij} is a sum of products of integers, and thus it can take only integer values. Therefore, there is no need to require variables y_{ij} to be integers since (3.2) will guarantee it. This observation reduces the number of integer variables in the model, thus making the solution process more efficient.

4. Difference Constraints

A difference restriction is given for two adjacent cells with a target number d. For example, if the cells are (*i*, *j*) and (*i*, *j* + 1) then the restriction is that *either* $y_{ij} - y_{i,j+1} = d$ *or* $y_{i,j+1} - y_{ij} = d$. This is an example of Either-Or constraints (reference). While each of the two equalities is a linear constraint, their Either-Or combination is not. It should be replaced by an equivalent set of constraints where all of them must hold. It is normally done by introducing a new binary variable and using the big *M* method [7]. However, that method works when each of the Either-Or constraints is an inequality. In the case of equalities, first each of them should be switched to a pair of equivalent inequalities and only then applying the standard technique. But in this case there is a simpler way of switching to linear constraints as explained below.

Note that requiring to have one of $y_{ij} - y_{i,j+1} = d$ *or* $y_{i,j+1} - y_{ij} = d$ hold is equivalent of requiring that

$$\left| y_{ij} - y_{i,j+1} \right| = d .$$

The absolute value makes the constraint nonlinear. But it has the following equivalent linear constraint

$$y_{ij} - y_{i,j+1} = d - 2 \cdot u \cdot d , \tag{4.1}$$

where *u* is an auxiliary binary variable.

When *u* = 0, (4.1) becomes $y_{ij} - y_{i,j+1} = d$; while when *u* = 1, (4.1) becomes $y_{ij} - y_{i,j+1} = -d$.

For example, the cage consisting of cells (1, 3) and (1, 4) in **Figure 1** needs the following constraint: $y_{13} - y_{14} = 7 - 2 \cdot 7 \cdot u$.

5. Division Constraints

A division restriction is given for two adjacent cells with a target number r. For example, if the cells are (*i*, *j*) and

(i, j + 1) then the restriction is that *either* $y_{ij}/y_{i,j+1}=r$ *or* $y_{i,j+1}/y_{ij}=r$. This is another example of Either-Or constraints. But unlike the difference constraints, there is no simple way of switching it to equivalent linear constraints. Below is given the sequence of steps for achieving linearity.

1) $y_{ij}/y_{i,j+1}=r$ *or* $y_{i,j+1}/y_{ij}=r$.
2) $y_{ij}-r\cdot y_{i,j+1}=0$ *or* $y_{i,j+1}-r\cdot y_{ij}=0$ (by simple algebra).
3) ($y_{ij}-r\cdot y_{i,j+1}\ge 0$ and $y_{ij}-r\cdot y_{i,j+1}\le 0$) *or* ($y_{i,j+1}-r\cdot y_{ij}\ge 0$ and $y_{i,j+1}-r\cdot y_{ij}\le 0$)
(by switching to equivalent pairs of inequalities).

4)
$$y_{ij}-r\cdot y_{i,j+1}\ge 0-M\cdot u \quad (5.1)$$
$$y_{ij}-r\cdot y_{i,j+1}\le 0+M\cdot u \quad (5.2)$$
$$y_{i,j+1}-r\cdot y_{ij}\ge 0-M\cdot(1-u) \quad (5.3)$$
$$y_{i,j+1}-r\cdot y_{ij}\le 0+M\times(1-u), \quad (5.4)$$

where u is an auxiliary binary variable and M is a large positive number (its choice is explained below).

When u = 0, (5.1) and (5.2) together imply $y_{ij}-r\cdot y_{i,j+1}=0$ and hence $y_{ij}/y_{i,j+1}=r$; while (5.3) and (5.4) do not force any restrictions on the variables.

When u = 1, (5.3) and (5.4) together imply $y_{i,j+1}-r\cdot y_{ij}=0$ and hence $y_{i,j+1}/y_{ij}=r$; while (5.1) and (5.2) do not force any restrictions on the variables.

Thus, constraints (5.1)-(5.4) should be included in the model for every ratio cage.

Proper choice of big M.

To solve the integer program efficiently, M should be assigned the smallest possible value. For a puzzle of size $n \times n$, the smallest value for M to make constraints (5.1)-(5.4) work is $r\cdot\left(n-(n \bmod r)\right)-(n \operatorname{div} r)$. Consider the following example to illustrate that choice. Suppose the target number of a ratio cage is 3 in a puzzle of size 8 × 8. Then the highest cell values satisfying the ratio restriction are 2 and 6. Here 6 is the highest multiple of 3 under 8, and it can be written as 8 – (8 mod 3); while 2 is 6 div 3 = 8 div 3. Suppose u = 1, $y_{i,j+1}$ = 6, and y_{ij} = 2, thus making $y_{i,j+1}-r\cdot y_{ij}=0$. Then the left-hand side of (5.1) is
$y_{ij}-r\cdot y_{i,j+1}=2-3\times 6=(8 \operatorname{div} 3)-3\times\left(8-(8 \bmod 3)\right)=-16$. Thus, 16 should be the smallest value assigned to M to make constraint (5.1) satisfied.

In the AMPL model of Section 7, the (5.1)-(5.4) are given in the segment named "ratio constraints", while big M is defined in the segment named "sets and parameters used in ratio constraints".

Example.

The cage consisting of cells (5, 2) and (5, 3) in **Figure 1** needs the following set of constraints:

$$y_{52}-4\cdot y_{53}\ge 0-30\cdot u$$
$$y_{52}-4\cdot y_{53}\le 0+30\cdot u$$
$$y_{53}-4\cdot y_{52}\ge 0-30\cdot(1-u)$$
$$y_{53}-4\cdot y_{52}\le 0+30\cdot(1-u)$$

6. Product Constraints

Product restrictions are the hardest ones for converting to linear constraints. Simply requiring that the product of y-variables is equal to the target number makes it a nonlinear constraint. We suggest two different ways of giving linear constraints for product restrictions.

Our main method gives a non-standard way of covering all product restrictions by using prime number factorizations of target numbers. That method is covered in Subsection 6.1 and is the basis of the AMPL model given in Section 7.

The second method is more intuitive and can be used to practice standard integer programming techniques. The disadvantages of this method are: (i) auxiliary binary variables are needed to write the constraints which makes the solution of the integer program less efficient; (ii) more importantly, while the method covers the most common situations it is not clear how to extend it to the general case. But this method is even more intuitive and simple than method 1 for the most common product restrictions when a cage consists of two cells (3 out of 5 product cages in the example of **Figure 1** consist of two cells). Thus, one can use a combination of methods 1 and 2 when building the model. An idea how method 2 works is given in Subsection 6.2 for a special case; the rest of the discussion is in **Appendix A**.

6.1. Method 1 for Product Restrictions

The method first finds the prime number factorization of the target number. The puzzles in [1] are of size at most 9 × 9. For that kind of puzzles, the prime numbers in the factorizations are 2, 3, 5, 7. In this section we will give constraints for each of those prime numbers for a puzzle of size 9 × 9. But the constraints can be easily generalized to any problem size and any prime number.

- *Product constraint for* 5: Suppose the power of 5 in the prime number factorization is d (could also be 0). Then the following constraint should be added.

$$\sum_{(i,j)\in C} x_{ij5} = d \tag{6.1.1}$$

The constraint provides that the number of cells in cage C getting 5 is d.

- *Product constraint for* 7: The constraint for 5 can be easily extended to 7. Suppose the power of 7 in the prime number factorization is d (could also be 0). Then the following constraint should be added.

$$\sum_{(i,j)\in C} x_{ij7} = d \tag{6.1.2}$$

The constraint provides that the number of cells in cage C getting 7 is d.

- *Product constraint for* 3: Suppose the power of 3 in the prime number factorization is d (could also be 0). Then the following constraint should be added.

$$\sum_{(i,j)\in C} \left(x_{ij3} + x_{ij6} + 2x_{ij9}\right) = d \tag{6.1.3}$$

It is similar to the previous case except that each entry 9 in the cage contributes 2 to the total power of 3, thus the coefficient of x_{ij9} is 2.

- *Product constraint for* 2: Suppose the power of 2 in the prime number factorization is d (could also be 0). Then the following constraint should be added.

$$\sum_{(i,j)\in C} \left(x_{ij2} + 2x_{ij4} + x_{ij6} + 3x_{ij8}\right) = d \tag{6.1.4}$$

Here each entry 4 contributes 2 and each entry 8 contributes 3 to the total power of 2. Thus the coefficient of x_{ij4} is 2, and the coefficient of x_{ij8} is 3.

- *A complete example*: Suppose the target number is 2520. Its prime number factorization is $2^3 \times 3^2 \times 5^1 \times 7^1$. Then the following set of constraints is needed.

$$\sum_{(i,j)\in C} \left(x_{ij2} + 2x_{ij4} + x_{ij6} + 3x_{ij8}\right) = 3$$

$$\sum_{(i,j)\in C} \left(x_{ij3} + x_{ij6} + 2x_{ij9}\right) = 2$$

$$\sum_{(i,j)\in C} x_{ij5} = 1$$

$$\sum_{(i,j)\in C} x_{ij7} = 1$$

Note that there is no easy way to extend Method 2 (described in Subsection 6.2 and **Appendix A**) to this example.

General product constraint: Let P be the set of prime numbers used in a puzzle of size $n \times n$. For a prime number $p \in P$ we define the following three sets.

$M[p]$ represents the integers from 1 to n that are multiples of p but are not multiples of its square:

$$M\left[p\right] = \left\{i \in 1,\cdots,n : i \bmod p = 0 \text{ and } i \bmod p^2 != 0\right\};$$

S[*p*] represents the integers from 1 to *n* that are multiples of p^2 but are not multiples of p^3:

$$S[p] = \{ i \in 1, \cdots, n : i \bmod p^2 = 0 \text{ and } i \bmod p^3 \,!= 0 \};$$

C[*p*] represents the integers from 1 to *n* that are multiples of p^3:

$$C[p] = \{ i \in 1, \cdots, n : i \bmod p^3 = 0 \};$$

Note that there are no higher powers of *p* in actual Kenken puzzles; but the technique can be easily generalized to higher powers too.

Let power[*t*, *p*] be the power of $p \in P$ in prime number factorization of target number *t*. The parameter power[*t*, *p*] is recursively computed in the parameters section of the AMPL code. Then we have the following general product constraint for *p*.

$$\sum_{(i,j)\in C} \left(\sum_{k\in M[p]} x_{ijk} + \sum_{k\in S[p]} 2x_{ijk} + \sum_{k\in C[p]} 3x_{ijk} \right) = \text{power}[t, p] \tag{6.1.5}$$

This general constraint represents the product constraints in our AMPL code.

6.2. Method 2 for the Case When the Cage Consists of Two Cells and There Is a Single Factorization for the Target Number

Most product restrictions in KenKen are given for two adjacent cells, and there is a single factorization for the target number. It happens when

(i) the target number is a prime number, namely, 2, 3, 5, 7;

(ii) the target number is composite but the size of the puzzle implies a single factorization; for example, when the target number is 4, 9, 10, 14, 15, 16, 20, 21 in puzzles of size at most 9 × 9.

Case (*i*). Suppose the cage consists of two adjacent cells (*i*, *j*) and (*i*, *j* + 1), and the target number is a prime number *p*. Then the restriction is the following:

$$\left(x_{ijp} = 1 \text{ and } x_{i,j+1,1} = 1 \right) \text{ or } \left(x_{ij1} = 1 \text{ and } x_{i,j+1,p} = 1 \right) \tag{6.2.1}$$

There are two ways to convert the restriction to linear constraints.

Technique 1: Restriction (6.2.1) is equivalent to requiring that

$$\left(x_{ijp} + x_{i,j+1,1} = 2 \right) \text{ or } \left(x_{ij1} + x_{i,j+1,p} = 2 \right)$$

The Either-Or constraint is equivalent to the following pair of linear constraints:

$$x_{ijp} + x_{i,j+1,1} = 2 \cdot u, \tag{6.2.2}$$

$$x_{ij1} + x_{i,j+1,p} = 2 \cdot (1 - u), \tag{6.2.3}$$

where *u* is an auxiliary binary variable.

When *u* = 1, (6.2.2) and (6.2.3) together imply $x_{ijp} = 1$, $x_{i,j+1,1} = 1$, $x_{ij1} = 0$, $x_{i,j+1,p} = 0$; thus cell (*i*, *j*) gets value *p*, and cell (*i*, *j* + 1) gets value 1.

When *u* = 0, (6.2.2) and (6.2.3) together imply $x_{ijp} = 0$, $x_{i,j+1,1} = 0$, $x_{ij1} = 1$, $x_{i,j+1,p} = 1$; thus cell (*i*, *j*) gets value 1, and cell (*i*, *j* + 1) gets value *p*.

Technique 2: The second way is less intuitive but simpler since it is given by just one constraint without using any auxiliary variables.

$$x_{ijp} + x_{i,j+1,1} + x_{ij1} + x_{i,j+1,p} = 2 \tag{6.2.4}$$

Another advantage of this second way is that it does not need any new auxiliary binary variables.

Constraint (6.2.4) does not work by itself but rather with the combination of other constraints we introduced before. Recall that constraint (2.1) provides that each cell gets exactly one value *k* from $1, \cdots, n$; that constraint for cells (*i*, *j*) and (*i*, *j* + 1) are given below:

$$\sum_{k=1}^{n} x_{ijk} = 1, \quad \sum_{k=1}^{n} x_{i,j+1,k} = 1 \tag{2.1a}$$

Also, constraint (2.2) provides that there is exactly one number *k* assigned to each row *i*; that constraint for row *i* and numbers 1 and *p* are given below:

$$\sum_{j=1}^{n} x_{ij1} = 1, \quad \sum_{j=1}^{n} x_{ijp} = 1 \tag{2.2a}$$

There are only two combinations of variable values that can satisfy constraints (2.1), (2.1a), (2.2a) at the same time:

- $x_{ijp} = 1$, $x_{i,j+1,1} = 1$, $x_{ij1} = 0$, $x_{i,j+1,p} = 0$; thus cell (*i*, *j*) gets value *p*, and cell (*i*, *j* + 1) gets value 1.
- $x_{ijp} = 0$, $x_{i,j+1,1} = 0$, $x_{ij1} = 1$, $x_{i,j+1,p} = 1$; thus cell (*i*, *j*) gets value 1, and cell (*i*, *j* + 1) gets value *p*.

Case (*ii*). Suppose the target number is composite but the size of the puzzle implies a single factorization. For example, if the target number is 30 for a puzzle of size 9 × 9 then the only factorization is 30 = 5 × 6. The solution for this case is identical to case (i) by taking 5 and 6 instead of 1 and *p*.

Example.

The cage consisting of cells (2, 3) and (2, 4) in **Figure 1** needs the following constraint:

$$x_{235} + x_{246} + x_{236} + x_{245} = 2$$

7. The AMPL Model and Its Solution

In this section we give the full AMPL model for the integer program developed in previous sections. The model has comments for most of the parameters, sets, variables, constraints; more detailed explanations about how they work are given in Sections 2-6. We also give a data set for the example of **Figure 1** and its solution.

The model is given in Section 7.1. The data set is in Section 7.2. A brief analysis of the solution process and efficiency follows in Section 7.3.

7.1. The AMPL Model

```
######### INPUT DATA: PARAMETERS AND SETS ########
param n;
# size of the problem
set rows := 1..n;
set columns := 1..n;
set cells := {rows, columns};
### Cage information ###
param t; # number of cages
set cage{1..t} within cells;
# each cage is a collection of cells
check: forall {(i,j) in cells} card({c in 1..t: (i,j) in cage[c]}) = 1;
# checking that each cell is in exactly one cage
param target_number{1..t};
param operation{1..t} symbolic;
# target number and operation for each cage
### The following parameters are used for giving difference and ratio constraints ###
### by distinguishing the two cells in difference and ratio cages ###
param min_cell_number{c in 1..t}:=min{(i,j) in cage[c]}(i+j);
param max_cell_number{c in 1..t}:=max{(i,j) in cage[c]}(i+j);
### Sets and parameters used in ratio constraints ###
set possible_ratio_numbers:=2..n;
# possible ratio numbers in a puzzle of size n
param bigM{r in possible_ratio_numbers: r<=n}:= r * (n - (n mod r)) - (n div r);
### Sets and parameters used in product constraints ###
set prime_numbers;
# prime numbers for puzzles of size up to 9
set prime_number_multiples {p in prime_numbers}:= {i in 1..n: i mod p = 0 and i mod (p^2) != 0};
set prime_number_squares {p in prime_numbers}:= {i in 1..n: i mod (p^2) = 0 and i mod (p^3) != 0};
set prime_number_cubes {p in prime_numbers}:= {i in 1..n: i mod (p^3) = 0};
param max_multipl_target_numbers := max{c in 1..t: operation[c]='product'} target_number[c];
# maximum product target number in the puzzle
param prime_number_power {m in 1..max_multipl_target_numbers, p in prime_numbers}
:= if (m mod p != 0) then 0 else prime_number_power[m div p, p] + 1;
# this parameter gives the power of prime number p in prime number factorization of m
################################ VARIABLES ##################################
var x{i in rows, j in columns, k in 1..n} binary;
# is equal 1 if value k is assigned to cell (i,j)
var y{i in rows, j in columns} >=1, <=n;
# the value assigned to cell (i,j)
var u{c in 1..t: operation[c]='difference' or operation[c]='ratio'} binary;
# auxiliary binary variable for a difference or ratio cage c
############################ OBJECTIVE FUNCTION ############################
maximize something: 1;
##### no need for an objective function #####
################################ CONSTRAINTS ################################
##### Latin Square Constraints #####
# Constraint (2.1): each cell is assigned exactly one value
subject to one_value_for_each_cell {i in rows, j in columns}:
sum{k in 1..n} x[i,j,k]=1;
# Constraint (2.2): each row has exactly one number k
subject to one_of_each_number_for_each_row{i in rows, k in 1..n}:
sum{j in columns} x[i,j,k]=1;
# Constraint (2.3): each column has exactly one number k
subject to one_of_each_number_for_each_column{j in columns, k in 1..n}:
sum{i in rows} x[i,j,k]=1;
# Constraint (3.2): connection between x and y variables
subject to value_of_each_square {i in rows, j in columns}:
y[i,j]=sum{k in 1..n} k*x[i,j,k];
##### Constraint (3.1): Sum Constraints #####
s.t. Cage_sum_restriction{c in 1..t: operation[c] == 'sum'}:
  sum{(i,j) in cage[c]} y[i,j] = target_number[c];
##### Constraint (4.1): Difference Constraints #####
s.t. Cage_difference_restriction{c in 1..t: operation[c] == 'difference'}:
  sum{(i,j) in cage[c]: i+j == max_cell_number[c]} y[i,j] -
  sum{(i,j) in cage[c]: i+j == min_cell_number[c]} y[i,j] =
         target_number[c] - 2 * target_number[c] * u[c];
##### Ratio Constraints (5.1)-(5.4) #####
s.t. Cage_ratio_restriction1{c in 1..t: operation[c] == 'ratio'}:
  sum{(i,j) in cage[c]: i+j = max_cell_number[c]} y[i,j] -
  target_number[c] * sum{(i,j) in cage[c]: i+j = min_cell_number[c]} y[i,j] >=
          - bigM[target_number[c]] * u[c];
s.t. Cage_ratio_restriction2{c in 1..t: operation[c] == 'ratio'}:
  sum{(i,j) in cage[c]: i+j = max_cell_number[c]} y[i,j] -
  target_number[c] * sum{(i,j) in cage[c]: i+j = min_cell_number[c]} y[i,j] <=
          bigM[target_number[c]] * u[c];
s.t. Cage_ratio_restriction3{c in 1..t: operation[c] == 'ratio'}:
  sum{(i,j) in cage[c]: i+j = min_cell_number[c]} y[i,j] -
  target_number[c] * sum{(i,j) in cage[c]: i+j = max_cell_number[c]} y[i,j] >=
          - bigM[target_number[c]] * (1 - u[c]);
s.t. Cage_ratio_restriction4{c in 1..t: operation[c] == 'ratio'}:
    sum{(i,j) in cage[c]: i+j = min_cell_number[c]} y[i,j] -
    target_number[c] * sum{(i,j) in cage[c]: i+j = max_cell_number[c]} y[i,j] <=
            bigM[target_number[c]] * (1 - u[c]);
##### Product Constraints (6.1.5) #####
s.t. Cage_product_restriction{c in 1..t, p in prime_numbers: operation[c] == 'product' and p<=n}:
  sum{(i,j) in cage[c], k in prime_number_multiples[p]} x[i,j,k] +
  sum{(i,j) in cage[c], k in prime_number_squares[p]} 2*x[i,j,k] +
  sum{(i,j) in cage[c], k in prime_number_cubes[p]} 3*x[i,j,k] = prime_number_power[target_number[c],p];
```

7.2. The Data Set for Figure 1

```
param n:=5;
param t:=10;
set cage[1]: = (1, 1) (1, 2) (2,1);
set cage[2]: = (1, 3) (1, 4);
set cage[3]: = (1, 5) (2, 5);
set cage[4]: = (2, 2) (2, 3) (2,4) (3,2) (4,2);
set cage[5]: = (3, 1) (4, 1);
set cage[6]: = (3, 3) (3, 4);
set cage[7]: = (3, 5) (4, 5) (5,5);
set cage[8]: = (4, 3) (5, 3) (5,4);
set cage[9]: = (4, 4);
set cage[10]: = (5, 1) (5,2);
param: target_number operation:=
1         9     "product"
2         1     "difference"
3         2     "ratio"
4         13    "sum"
5         1     "difference"
6         2     "ratio"
7         15    "product"
8         24    "product"
9         3     "sum"
10        3     "difference";
set prime_numbers:= 2 3 5 7;
```

7.3. The Solution, Efficiency of the Model and Future Directions

We ran the AMPL model on the NEOS server using the solver Gurobi [8] and received the following correct solution for the puzzle.

```
y [*,*]
:    1    2    3    4    5    6    7    8     :=
1    3    5    8    1    2    4    7    6
2    2    3    5    6    8    1    4    7
3    8    6    7    5    1    3    2    4
4    4    7    2    8    5    6    3    1
5    7    1    4    3    6    2    8    5
6    6    4    3    2    7    5    1    8
7    1    2    6    7    4    8    5    3
8    5    8    1    4    3    7    6    2
;
```

It took under 1 second to return the solution. Thus, the model is very efficient for puzzles of size 9 or less. But it is still an interesting question how to make the model more efficient for any size. Below we describe related open questions.

It would be more efficient to solve the LP-relaxation of our integer program. But our computations show that the LP-relaxation does not always return an integer solution. It is an interesting open question if there are any techniques to achieve integrality. A possible technique could be the following.

The integer program does not need an objective function. Thus, one has a flexibility of adding an appropriate objective function. It is an interesting open question if there is an objective function that would make the optimal solution of the LP-relaxation integral.

The existence of appropriate cutting planes that would make the solution process more efficient and perhaps make the optimal solution of the LP-relaxation integral is another open question.

References

[1] The KenKen Website. http://www.kenkenpuzzle.com/

[2] Watkins, J. (2012) Triangular Numbers, Gaussian Integers, and KenKen. *The College Mathematics Journal*, **43**, 37-42. http://dx.doi.org/10.4169/college.math.j.43.1.037

[3] Reiter, H., Thornton, J. and Vennebush, G.P. (2013) Using KenKen to Build Reasoning Skills. *Mathematics Teacher*, **107**, 341-347.

[4] Chlond, M. (2005) Classroom Exercises in IP Modeling: Sudoku and the Log Pile. *INFORMS Transactions on Education*, **5**, 77-79. http://dx.doi.org/10.1287/ited.5.2.77

[5] Bartlett, A.C., Chartier, T.P., Langville, A.N. and Rankin, T.D. (2008) An Integer Programming Model for the Sudoku Problem. *The Journal of Online Mathematics and Its Applications*, **8**, Article ID: 1798. http://www.maa.org/external_archive/joma/Volume8/Bartlett/index.html

[6] The AMPL Website. http://www.ampl.com/

[7] Hillier, F. and Lieberman, G. (2014) Introduction to Operations Research. 10th Edition, McGraw-Hill, New York.

[8] The Gurobi Website. https://neos-server.org/neos/solvers/lp:Gurobi/AMPL.html

Appendix. Method 2 for Product Restrictions

In this appendix, we give a further discussion on Method 2 for product restrictions.

A1. The Subcase When the Cage Consists of Two Cells and There Are More Than One Factorization for the Target Number

In puzzles of size at most 9 × 9, if a product cage consists of two cells and the target number is composite then at most 2 different factorizations are possible. It is easy to verify it for all possible composite target numbers. Namely, numbers 4, 9, 10, 14, 15, 16, 20, 21, 27, 28, 30, 32, 35, 36, 40, 42, 45, 48, 54, 56, 63, 72 can have only one factorization; while numbers 6, 8, 12, 18, 24 can have two different factorizations. In this subsection we will show how to write linear constraints in the case of two different factorizations. For the sake of simplicity and better demonstration, we will show it on an example of a specific number; the technique is identical for other numbers.

Suppose the cage consists of two adjacent cells (*i*, *j*) and (*i*, *j* + 1), and the target number is 12. Then the restriction is the following:

$$\left(x_{ij3}=1 \text{ and } x_{i,j+1,4}=1\right)\text{or}\left(x_{ij4}=1 \text{ and } x_{i,j+1,3}=1\right)\text{or}\left(x_{ij2}=1 \text{ and } x_{i,j+1,6}=1\right)\text{or}\left(x_{ij6}=1 \text{ and } x_{i,j+1,2}=1\right) \quad \text{(A.1)}$$

As in Subsection 6.1, there are two ways to convert the restriction to linear constraints. And again one of the methods is less intuitive but more efficient since it requires fewer constraints and auxiliary binary variables. But for the sake of comparison and completeness, we will give both methods, starting from the more intuitive one.

Method 1: The restriction (A.1) is equivalent to requiring that

$$\left(x_{ij3}+x_{i,j+1,4}=2\right)\text{or}\left(x_{ij4}+x_{i,j+1,3}=2\right)\text{or}\left(x_{ij2}+x_{i,j+1,6}=2\right)\text{or}\left(x_{ij6}+x_{i,j+1,2}=2\right)$$

The "1-out-of-4 must hold" constraint is equivalent to the following set of linear constraints:

$$x_{ij3}+x_{i,j+1,4}=2\cdot u_{34}, \quad \text{(A.2)}$$

$$x_{ij4}+x_{i,j+1,3}=2\cdot u_{43}, \quad \text{(A.3)}$$

$$x_{ij2}+x_{i,j+1,6}=2\cdot u_{26}, \quad \text{(A.4)}$$

$$x_{ij6}+x_{i,j+1,2}=2\cdot u_{62}, \quad \text{(A.5)}$$

$$u_{34}+u_{43}+u_{26}+u_{62}=1 \quad \text{(A.6)}$$

where u_{34}, u_{43}, u_{26}, u_{62} are auxiliary binary variables.

Constraint (A.6) implies that exactly one of the u_{pq} variables takes value 1 while others are zero. When $u_{pq}=1$, the corresponding x_{ijp} and $x_{i,j+1,q}$ variables also take value 1 while other x-variables in (A.2)-(A.5) are forced to be zero; thus cell (*i*, *j*) gets value *p*, and cell (*i*, *j* + 1) gets value *q*.

Method 2: The first step of this method is the same as in method 1. The restriction (A.1) is equivalent to requiring that

$$\left(x_{ij3}+x_{i,j+1,4}=2\right)\text{or}\left(x_{ij4}+x_{i,j+1,3}=2\right)\text{or}\left(x_{ij2}+x_{i,j+1,6}=2\right)\text{or}\left(x_{ij6}+x_{i,j+1,2}=2\right)$$

But from here we proceed as it was done in method 2 of Subsection 6.1. The Either-Or restriction ($x_{ij3}+x_{i,j+1,4}=2$) or ($x_{ij4}+x_{i,j+1,3}=2$) is equivalent to constraint $x_{ij3}+x_{i,j+1,4}+x_{ij4}+x_{i,j+1,3}=2$ since x-variables satisfy (2.1) and (2.2). Based on the same reason, the Either-Or restriction ($x_{ij2}+x_{i,j+1,6}=2$) or ($x_{ij6}+x_{i,j+1,2}=2$) is equivalent to $x_{ij2}+x_{i,j+1,6}+x_{ij6}+x_{i,j+1,2}=2$.

Thus, the restriction (A.1) is equivalent to

$$\left(x_{ij3}+x_{i,j+1,4}+x_{ij4}+x_{i,j+1,3}=2\right)\text{or}\left(x_{ij2}+x_{i,j+1,6}+x_{ij6}+x_{i,j+1,2}=2\right)$$

This Either-Or restriction is converted to an equivalent pair of linear constraints in a standard way.

$$x_{ij3}+x_{i,j+1,4}+x_{ij4}+x_{i,j+1,3}=2\cdot u, \quad \text{(A.7)}$$

$$x_{ij2}+x_{i,j+1,6}+x_{ij6}+x_{i,j+1,2}=2\times(1-u), \quad \text{(A.8)}$$

where u is an auxiliary binary variable.

A2. The Subcase When the Cage Consists of K Cells in the Same Row (Column) and There Is a Single Factorization for the Target Number

The technique discussed in this section is the extension of the technique for 2-cell cages discussed in subsection 6.1.

Suppose the cells in the cage are $(i,j),(i,j+1),\cdots,(i,j+k-1)$, the target value is m, and there is a single factorization of $m=m_1\cdot m_2\cdots m_k$. Then there are k! permutations of assigning numbers $m_1,m_2,\cdots,m_k$ to the cells $(i,j),(i,j+1),\cdots,(i,j+k-1)$. The following linear constraint together with Latin square constraints (2.1) and (2.2) forces that exactly one of those permutations is chosen.

$$\sum_{r=0}^{k-1}\sum_{p=1}^{k}x_{i,j+r,m_p}=k \tag{A.9}$$

The reason that this combination of constraints works is the same that we had for two-cell cages with a single factorization as discussed in Subsection 6.1.

Here is a specific example to illustrate how constraint (A.9) works. Suppose the cells in the cage are (1, 1), (1, 2), (1, 3); the target number is 18, and thus the single factorization is 18 = 1 × 3 × 6. The constraint (A.9) in this case is

$$x_{111}+x_{113}+x_{116}+x_{121}+x_{123}+x_{126}+x_{131}+x_{133}+x_{136}=3$$

This constraint together with Latin square constraints (2.1) and (2.2) provides that exactly one of the following 3! = 6 combinations of assignments:

- cell (1,1) gets 1; cell (1,2) gets 3; cell (1,3) gets 6;
- cell (1,1) gets 1; cell (1,2) gets 6; cell (1,3) gets 3;
- cell (1,1) gets 3; cell (1,2) gets 1; cell (1,3) gets 6;
- cell (1,1) gets 3; cell (1,2) gets 6; cell (1,3) gets 1;
- cell (1,1) gets 6; cell (1,2) gets 1; cell (1,3) gets 3;
- cell (1,1) gets 6; cell (1,2) gets 3; cell (1,3) gets 1.

A3. The Subcase When the Cage Consists of K Cells in the Same Row (Column) and There Are More Than One Factorization for the Target Number

The technique discussed in this section is the extension of the technique for 2-cell cages discussed in subsection A1.

Suppose the cells in the cage are $(i,j),(i,j+1),\cdots,(i,j+k-1)$, the target value is m, and there are more than one factorizations. Denote the set of all factorizations $\Phi=\{F_1,\cdots,F_s\}$. Let the factorization for F_t be $m=m_{t1}\cdot m_{t2}\cdots m_{tk}$. As in subsection A1, the constraint for factorization F_t would be

$$\sum_{r=0}^{k-1}\sum_{p=1}^{k}x_{i,j+r,m_{tp}}=k \tag{A.10}$$

But we want constraint (A.10) to be satisfied by only one of the factorizations. Note that for the other factorizations the left-hand side of (A.10) is not necessarily 0 since the same factor m could be in more than one factorization. The following set of constraints takes the above considerations into account.

$$\sum_{r=0}^{k-1}\sum_{p=1}^{k}x_{i,j+r,m_{tp}}\geq k\cdot u_t\,,\ \text{ for each }\ F_t\in\Phi \tag{A.11}$$

$$\sum_{t=1}^{s}u_t=1 \tag{A.12}$$

Note that the left-hand side of (A.11) cannot be more than k because of constraints (2.1) and (2.2). Thus, (A.12) will force that the left-hand side of (A.11) is equal to k for exactly one factorization; for other factorizations, the left-hand side of (A.11) is ≥0, hence not forcing anything on its x-variables.

Here is a specific example to illustrate how constraints (A.11)-(A.12) work. Suppose the cells in the cage are

(1, 1), (1, 2), (1, 3); the target number is 48, and thus the following three factorizations are possible:

$$48 = 1 \times 6 \times 8;$$

$$48 = 2 \times 3 \times 8;$$

$$48 = 2 \times 4 \times 6.$$

The constraints (A.11)-(A.12) in this case are

$$x_{111} + x_{116} + x_{118} + x_{121} + x_{126} + x_{128} + x_{131} + x_{136} + x_{138} \geq 3 \cdot u_1 \tag{A.13}$$

$$x_{112} + x_{113} + x_{118} + x_{122} + x_{123} + x_{128} + x_{132} + x_{133} + x_{138} \geq 3 \cdot u_2 \tag{A.14}$$

$$x_{112} + x_{114} + x_{116} + x_{122} + x_{124} + x_{126} + x_{132} + x_{134} + x_{136} \geq 3 \cdot u_3 \tag{A.15}$$

$$u_1 + u_2 + u_3 = 1 \tag{A.16}$$

A4. General Case (Cage Occupies Multiple Rows and Columns)

A problem in this case could be the following. Even in the case of a single factorization, e.g., the cage is in cells (1, 1), (1, 2), (2, 1), and the target number is 35 = 1 × 5 × 7, constraint of type (A.9) would not work:

$$x_{111} + x_{115} + x_{117} + x_{121} + x_{125} + x_{127} + x_{211} + x_{215} + x_{217} = 3$$

A solution satisfying this constraint could be $x_{111} = 1$, $x_{125} = 1$, $x_{215} = 1$, that is, cells (1, 2) and (2, 1) getting value 5 while cell (1, 1) getting value 1, hence the cell restriction is not satisfied. Note that constraints (2.1) and (2.2) are not violated by this solution.

A solution for the general case is given by the method described in Subsection 6.1.