# KenKen Modelling Project

## December 7th, 2020

## CMPE204

## Team 107 :

Nicole Ooi – 16NMXO - 20060957

Lukas Bauer – 16LJB6 - 20052327

Michael Assheton-Smith – 16MBAS1 – 20053112

Alexander Ingham – 16ADWI – 20054078

Statement of originality:

Following professional practice, we bear the burden of proof for original work. We have read the Policy on Academic Integrity posted on the Faculty of Engineering and Applied Science web site (http://engineering.queensu.ca/policy/Honesty.html) and confirm that this work is in accordance with the Policy. All work in this report is by the listed author, with the exception of properly cited material.

# Project Summary

KenKen is a popular arithmetic and logic puzzle that appears in 150 publications worldwide and is the favorite logic game of members of this project. Like sudoku, a successful solution adheres to a defined set of conditions. This project aims to determine whether a given configuration is a valid KenKen board, to estimate its level of difficulty, and to provide the solution to the board. A valid KenKen board has only 1 possible solution, and this will dictate whether the given configuration is valid or not. An example KenKen board is provided in Figure 1.

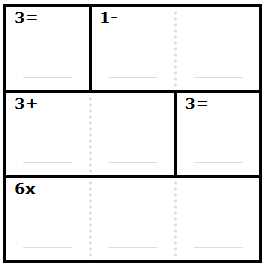


Figure 1 - 3x3 example of a KenKen configuration (board configuration #2 in run.py)

For concision, a few terms will be defined. N will refer to the length of one side of the board, measured in squares – as all KenKen boards are square-shaped, this can be considered the length or width. A region is a bolded area denoted by a number and operator, or only a number, in the top corner. The region consists of 1 or more squares within the KenKen board, where a square may only hold one number. Lastly, the numbering for rows and columns will begin at 1 in the report.

A KenKen configuration is valid if the following rules are met:

1. Each column contains the natural numbers between 1 and N, inclusive.
2. Each row contains the natural numbers between 1 and N, inclusive.
3. The operator in the top left of each bolded region can be applied to the numbers filled in the squares within in that region to result in the number in the top left of that region. The operator may be applied in any direction (ie. Left to right, or right to left) as long as one of the directions results in the number in the top left of that region.

# Propositions

This is true when the row i has all the required numbers, from 1 to N.

This is true when the column j has all the required numbers, from 1 to N.

This is true when the region x evaluates to n when using the mathematical operator t.

: This is true if the square at row i and column j, or (i, j) contains the number k.

This is true if the square at row i and column j, or (i,j) contains a number from 1 to N.

# Constraints

### Constraint 1

Each square may only contain the numbers 1-N.

For example, is a proposition that can be created for a 3x3 KenKen board and can either be True or False. However, would not be created for a KenKen board within the scope of this project and would never be True.

This can also be shown with:

### Constraint 2

The board’s smallest size is 3x3 and the largest size it can go up to is 4x4.

For example, is a proposition that can be created within the scope of the project. However, would not be created within the scope of this project, as it references a 5th row and column.

### Constraint 3

The board’s solution must not defy any of the game’s rules. These rules are the following:

* 1. Each row must contain all the required numbers, from 1 to N, in any order.

For a 3x3 board:

* 1. Each column must contain all the required numbers, from 1 to N, in any order.

For a 3x3 board:

* 1. Each region’s arithmetic constraints are met using the numbers within the region, the mathematical operator and the assigned result.

For a region in a 3x3 board as shown below:

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

The region has two valid configurations for its arithmetic constraints to be met with a 3 in the top and a 1 in the bottom, or vice versa. Other regions were omitted for clarity.

# Implementation

## Python Implementation

### Propositions and Constraints

Propositions and constraints were added to the Encoding object within the test\_kenken(N) function. The propositions and constraints that were used, along with their implementation in the code, can be found in Table 1.

Table 1 - Python implementation of corresponding model propositions and constraints.

|  |  |
| --- | --- |
| Propositions/Constraints | Python Implementation |
| This is true when the row i has all the required numbers, from 1 to N.  This is true when the column j has all the required numbers, from 1 to N. | These propositions were implemented by iterating over all ri nnf variables and adding a constraint. The constraint created three disjuncts of conjuncts, representing whether each number from 1 to N existed in the row. For example, if the value 1 existed in a row of 3 squares, the three possible placements for the 1 were ORed. To get each possible placement, the square’s values that were not 1 were negated and ANDed together with the square’s variable indicating that it is 1. The constraint added to the encoding object was that ri could only be true if and only if every single disjunction of conjuncts were true (indicating that each value existed in one and only one square in the row). Column propositions were implemented similarly. |
| This is true when the region x evaluates to n when using the mathematical operator t. | Each region in the board was stored in an object of class *Region*. This class contained instance variables: *members, rslt,* and *operator*. *Members* contained a list of the squares within that region. *Operator* and *rslt* variables stored the operation corresponding to that region, as well as the region’s desired arithmetic result, respectively. |
| : This is true if the square at row i and column j, or (i, j) contains the number k. | Each square in the board was stored in an object of class *Square*. These objects had instance variables: *is\_valid* and *value*. *Value* was a list of nnf variables, of length N. The values were represented with one-hot encoding: the 0th element in the list represented the leftmost bit and the list was a sequence of bits. Each bit was represented by an nnf variable. |
| This is true if the square at row i and column j, or (i,j) contains a number from 1 to N. | These propositions were implemented by iterating over all squares in the board and adding a constraint for its vi,j nnf variable, *is\_valid* contained in *Square*. The constraint ensured that the square’s value was any number from 1 to N by ORing their one-hot encodings, also known as their values. |

### Functions

To implement the model in python, two classes were defined: one class to represent squares on the board and another to represent regions. These classes were discussed above. Several helper functions were also defined, including: a function to display the board configuration, a function to display the solution to the board, an “If and only if” function, etc. The largest and most notable function is the test\_kenken(N) method. It is called by the main function and takes in the size of the board as an input. These functions are outlined in Table 2.

Table 2 - Functions used in Python Implementation

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Input(s) | Output(s) | Purpose |
| getSquareVal() | Atom | Integer | Helper function used to retrieve the value of the number represented from the nnf Var passed in as a parameter. |
| Iff() | left, right | True/False | Helper function to evaluate “if and only if” logical expressions |
| getTrueAtoms() | A solved encoding object “solution” | A sorted list of only the true atoms of the solution “true\_in\_sol” | Helper function to return list of only the atoms labeled "True" and in alphabetical order (useful for examining proposed solution for board) |
| displayBoard() | A solved encoding object “solution” and the boards dimension “N” | Prints the solution to the board | Helper function to visually display the solution for the board |
| printConfig() | A list of Region objects | Prints the initial board configuration | Helper function to visually display the initial configuration for the board |
| test\_kenken() | Size of the board “N” | Creates the encoding object and adds constraints to it. | Function to create and verify constraints for the board and add them to the encoding object to be solved. |

### Arithmetic

There were arithmetic operations that needed to be verified within the constraints of the project. This was required to ensure that the numbers of a certain region could assert the result using the given operator. To accomplish this, all possible arrangements of the respective operator’s application were verified using the built-in python “eval” function. Next, valid arrangements were made into constraints and exclusively ORed before being added to the encoding object to ensure that only one constraint could be added. This was done for addition, subtraction, multiplication, and division and can be found on lines 288-361 in run.py.

## JAPE Proofs

1. R1→(P1,1,1∨P1,2,1∨P1,3,1)∧(P1,1,2∨P1,2,2∨P1,3,2)∧(P1,1,3∨P1,2,3∨P1,3,3) ⊢ R1→¬(¬P1,1,1∧¬P1,2,1∧¬P1,3,1)
   * R1 being satisfied implies that at least one value in the first row is equal to 1, at least one value is equal to 2, and at least one value is equal to 3. This means that all the values in the first row cannot simultaneously not equal 1 if R1 is satisfied.
2. R1→(P1,1,1∨P1,2,1∨P1,3,1)∧(P1,1,2∨P1,2,2∨P1,3,2)∧(P1,1,3∨P1,2,3∨P1,3,3), P1,1,1→(¬P1,1,2∧¬P1,1,3), P1,2,2→(¬P1,2,1∧¬P1,2,3), P1,1,1∧P1,2,2 ⊢ R1→P1,3,3
   * R1 being satisfied implies that at least one value in the first row is equal to 1, at least one value is equal to 2, and at least one value is equal to 3. If the value in the first row and first column is equal to 1, then that square cannot hold the values 2 or 3. If the value in the first row and second column is equal to 2, then that square cannot hold the values 1 or 3. Supposing both of those squares hold the assumed values, then the remaining square in the first row must hold the value 3.
3. (¬P1,2,1∧¬P1,3,1)→P1,1,1, P1,2,2→(¬P1,2,1∧¬P1,2,3), P1,3,3→(¬P1,3,1∧¬P1,3,2) ⊢ (P1,2,2∧P1,3,3)→P1,1,1
   * If the value in the first row and second/third columns are not equal to 1, then 1 must be put in the remaining square in the row. Comparatively, if the value in the first row and second column is equal to 2, then 2 cannot be in the first/third columns. If the value in the third column of the first row is 3, then 1 or 2 cannot be the value in the third column. This means that if, of the first row, 2 is the value in the second column and 3 is the value in the third column, then 1 must be the value in the first column.

# Model Exploration

Explorations, interpretations, and analyses were required for both the submitted version of the model as well as during intermediate versions to understand and better communicate its available insight. The output from the model was extended to answer the original guiding questions for the project, to explore and test constraints, and to determine the computational limitations of the model.

As mentioned in the Project Summary section, the goal of the project was to model a solution of a KenKen board given its configuration, to determine if it is a valid board configuration with only one solution, and to approximate its difficulty to solve. The output of the model, given the constraints and variable initializations, is provided in a dictionary. The first way the model was explored was to extract this information and present it in a clear manner which satisfies the original goals of the project. Ideally, this would be a visual representation of the solved board, and a message to the user displaying its difficulty and whether there is only one solution or not. To achieve this, the helper function *getTrueAtoms(sol)* was created and called to take the solution and output the names of the atoms which evaluate to true in alphabetical order. Then, the function *displayBoard(solution, dim)* used this sorted solution along with the *getSquareVal(atom)* helper function to extract the numbered values for each of the squares and display them in a formatted manner. For example, if the model returned that the atom, a0\_1, was true, then *getTrueAtoms(sol)* would collect and organize it with the other atoms in alphabetical order, *getSquareVal(atom)* would strip “a0\_” to just leave the value “1”, and *displayBoard(solution, dim)* organized and visually printed this information in a replica board. A screenshot of a solved board can be found below in Figure 2. To output the difficulty, the time taken to for the SAT solver to solve the solutions was recorded and the puzzle was classified as easy, medium, hard, or very hard depending on its time taken. Finally, the number of solutions determined by *count\_solutions()* was used to communicate to the user whether it was a valid configuration with one solution or not. An example of this output can be seen in Figure 3.

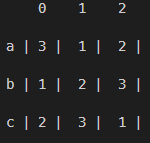


Figure 2 - Example board solution output for board configuration #2 (Figure 1).

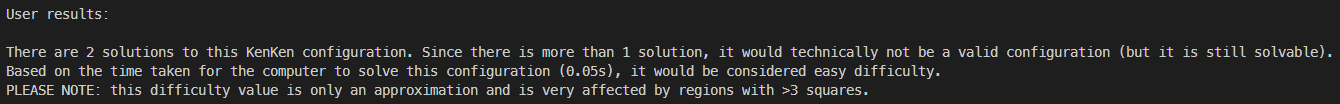


Figure 3 - Example of report for the user for board configuration #2 (Figure 1).

While extensions on the final model were necessary to present and interpret conclusions, the model during development was also examined to gain insights on the comprehensiveness of the constraints. Testing the row, column, and arithmetic constraints separately helped eliminate errors and produce better quality code once they were implemented together. An example of an error fixed through this process was that the model outputted zero valid solutions at one point while there was known to be at least one. To test the cause of this, the constraints were separated into three groups: the column constraints, the row constraints, and the arithmetic constraints. First, the arithmetic constraints were commented out. The expected result for a 3x3 board after these constraints were removed was that there would be 12 solutions, with no repeat numbers in any row or column. This number was originally determined by hand-counting the permutations. The output of the model still provided zero solutions, so the error was isolated to be in either the row or column constraints. Next, when the column constraints were commented out, the output jumped to 216 solutions (63) which was expected based on the calculated number of possible permutations. This indicated that there was a semantic error in the lines of code that generate the column constraints, and it was able to be found and fixed. A process like this was used multiple times during the development to ensure reliable output and execution of the model’s solver.

In the first version of the project, the Python model supported solving a 5x5 KenKen board. Unfortunately, the project’s goals had to be modified to limit configurations to 3x3 and 4x4 as execution time required to solve a 5x5 configuration was too large, and solutions were never found. This introduces the third area of exploration of the model: to determine the computational limitations of the system and improve optimization. The first area of optimization was to isolate redundant variables and constraints to speed up solving operation. To eliminate redundant generated atoms, the variables in question were added as a constraint and the number of solutions were recorded. Then, the negation of the variable was added as a constraint instead to see if the number of solutions were affected. If they were not affected over a range of tests, then the atom was redundant and removed. This elimination occurred for atoms generated by the Region class to determine if the region was satisfied or not. Since the constraints for the regions were already covered in the constraint section for the arithmetic, they could be removed. After this process was repeated for any questionable variables, configurations causing longer time to solve were inspected. By testing various initial configurations of boards, it was found that boards with multiple regions of size greater than 3 took significantly longer to solve (Figure 4 vs Figure 5). This is due to the structure of the arithmetic constraints, which requires calculations for all the permutations of the values in the regions rather than just the combinations. The exponential increase in generated statements and constraints caused increased computational time required in the model’s solving step. It was assumed that the difficulty ranking would also be influenced by regions containing more squares, so the increased time taken to solve these configurations was accounted for and contributed to developing the time thresholds which as a result determines the difficulty class.

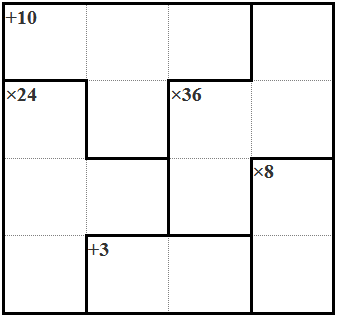


Figure 4 - Example of board configuration with multiple regions of 4+ squares which takes longer to solve.

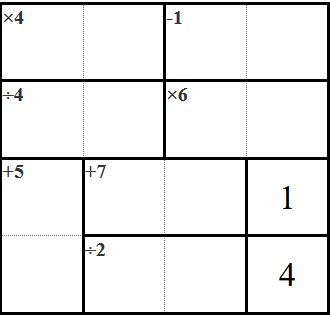


Figure 5 - Example of board configuration with regions of 2 squares which takes much less time to solve.

# First-Order Extension

The model using predicate logic could be extended to incorporate first-order logic and assist in the verification of KenKen boards. The use of universal quantifiers would allow us to create logical statements with rows and columns represented as predicates. Predicates would enable us to map points on the board (i,j) to the integer in that square.

## Propositions

One(i,j): True if the integer at (i, j) is 1. A predicate like this will be created for each number from 1 – N.

R(i): True if the row i contains all integers from 1 – N.

C(j): True if the column j contains all integers from 1 – N.

O(x, n, t): True if region x evaluates to n when using the mathematical operator t.

A(x, n, t): True if operator t is addition, and the sum of all numbers in region x results in n. Predicates M, S and D would each be defined the same but for respective operators multiplication, subtraction and division.

## Constraints

### Constraint 1

Each square may only contain the numbers 1-N.

### Constraint 2

The board’s smallest size is 3x3 and the largest size it can go up to is 4x4. This constraint is restricted by the domain for i and j.

### Constraint 3

The board’s solution must not defy any of the game’s rules.

1. Each row must contain all the required numbers, from 1 to N, in any order.

For a 3x3 board:

1. Each column must contain all the required numbers, from 1 to N, in any order.

For a 3x3 board:

1. Each region’s arithmetic constraints are met using the numbers within the region, the mathematical operator and the assigned result.

∀ *x O(x, n, t)* → *A(x, n, t)* ∨ *S(x, n, t)* ∨ *M(x, n, t)* ∨*D(x, n, t)*