# SAT-based 3D KenKen Solver

Zach Goldberg, Peter Kim, Maitreyee Joshi, Johannes Thorarensen

November 2020

## 1 Introduction

Everyone's heard of the game Sudoku, but have you heard of Sudoku's lesser-known little brother named KenKen? The game of KenKen has many similarities to the more popular game Sudoku. Just like in Sudoku, every row and column of the board may not contain duplicates of a number. However, unlike in Sudoku, in which the grid is always 9 x 9, each KenKen board is an $n$ x $n$ grid, where $3 \leq n \leq 9$. The grid cells in the KenKen board are also partitioned into 'cages', with each cage consisting of a target number and a mathematical operation (addition, subtraction, multiplication, or division). In addition, every cage only consists of adjacent cells. The goal of KenKen is to fill the grid with digits ranging from 1 to $n$ such that the grid is a Latin Square and when the mathematical operation is applied to all the entries in the cage, the output equals the target number. For example, take a look at an example board below:

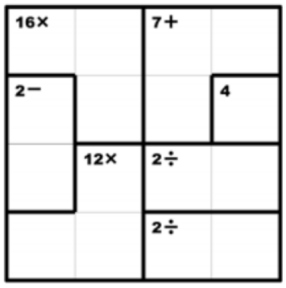


Figure 1: An Example KenKen Board

This 4x4 KenKen board is split into seven "cages". The top-rightmost cage consists of three cells with the target number of 7 and the addition operator. This means that the three numbers located inside those cages must add up to 7. Take a look at the center-leftmost cage consisting of two cells. We notice that this cage has a target number of 2 but with the subtraction operator. This

means that performing subtraction on those two variables must equate to the number 2. The two numbers must also be distinct, as the cage contains cells in the same column. An example set of numbers that would satisfy this cage's property would be $\{1, 3\}$, as $3 - 1 = 2$. The example KenKen board has been solved below to demonstrate its properties.

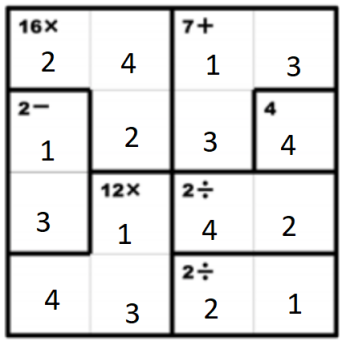


Figure 2: Solved Example KenKen Board

However, as SAT-solvers for traditional KenKen boards already exist, we decided to implement a little dimensional twist to the board itself. In place of 2-dimensional boards, we will be solving 3-dimensional KenKen "cubes" with the dimensions $n$ x $n$ x $n$. A 3-dimensional KenKen board must be a "latin cube", by which we mean that every 2-dimensional cross-section of the grid must be a Latin Square. The requirements for cages remain largely unchanged with two exceptions: the cells of a cage may span across all three dimensions as long as they are adjacent, and the size of cages, denoted by $c$, will be limited to $1 \leq c \leq 3$. We will convert a KenKen board into a DIMACS format that a SAT-solver will accept to generate a valid solution. We will then verify the solution output by the SAT-solver.

# 2 Approach

To model 3-dimensional KenKen boards using propositional logic, we largely adapted the constraints for what constitutes a valid solution for traditional 2-dimensional boards. Traditional KenKen solvers solve the game by representing boards as a Constraint Satisfaction Problem (CSP). These solvers often use backtracking algorithms or forward checking algorithms with minimum remaining values. The constraints are as follows for a traditional 2-dimensional KenKen Board:

1. Each cell in a $n$ x $n$ size puzzle must be a number between 1 and $n$.
2. A number cannot be repeated within the same row.

3. A number cannot be repeated within the same column.

4. In a one-cell cage, the cell's number will be the target number.

5. Each 'cage', or each region bounded by a heavy border, contains a target number and an arithmetic operation. Performing the arithmetic operation on each of the cells in the cage must produce the target number. Numbers may be repeated within a cage if necessary, as long as they do not repeat within a single row or column.

We will use boolean variables and expressions to replicate the digits and their constraints. For simplicity, it is common to require that the boolean expression be written in conjunctive normal form. As such, we represent the 3-d position and the number that occupies that position with an 4-digit number where the first three digits represent the 3-dimensional position with the last digit representing that cell's target number. So, in a 3x3 cube, if 1 is positioned at (0, 0, 0) we represent that as 0001. Similarly, if 3 is positioned at (0, 1, 2) we represent that as 0123. Therefore, if $n$ is the dimension, then there are $n^4$ SAT variables. This notation is converted into DIMACS format SAT variables using the table in Figure 3.

1=0001, 2=0002, 3=0003, 4=0011, 5=0012, 6=0013, 7=0021, 8=0022, 9=0023, 10=0101,
11=0102, 12=0103, 13=0111, 14=0112, 15=0113, 16=0121, 17=0122, 18=0123, 19=0201,
20=0202, 21=0203, 22=0211, 23=0212, 24=0213, 25=0221, 26=0222, 27=0223, 28=1001,
29=1002, 30=1003, 31=1011, 32=1012, 33=1013, 34=1021, 35=1022, 36=1023, 37=1101,
38=1102, 39=1103, 40=1111, 41=1112, 42=1113, 43=1121, 44=1122, 45=1123, 46=1201,
47=1202, 48=1203, 49=1211, 50=1212, 51=1213, 52=1221, 53=1222, 54=1223, 55=2001,
56=2002, 57=2003, 58=2011, 59=2012, 60=2013, 61=2021, 62=2022, 63=2023, 64=2101,
65=2102, 66=2103, 67=2111, 68=2112, 69=2113, 70=2121, 71=2122, 72=2123, 73=2201,
74=2202, 75=2203, 76=2211, 77=2212, 78=2213, 79=2221, 80=2222, 81=2223

Figure 3: DIMACS Conversion

Overall, there are five preliminary constraints on the digits.

1. For an $n$ x $n$ x $n$ cube, every coordinate value $x$ must be within range $1 <= x <= n$. Let's say a cube is 3 x 3 x 3. That means that at every coordinate is the number 1 or 2 or 3. See: `KenKen#everyBlockHasANumberClauses`

2. Every coordinate has no more than one value. So, 0001 implies not 0002 in CNF is $\neg 0001 \vee \neg 0002$, which when translated to actual DIMACS yields -1 -2 0. See: `KenKen#implicationsForAssignment`

3. All values on the $X$, $Y$, and $Z$ axes are distinct. So, in a 3 x 3 x 3 KenKen cube, $0001 \Rightarrow 1002 \vee 1003$ would be one implication that must hold. Converting it to CNF, we get $\neg 0001 \vee 1002 \vee 1003$. We represent this CNF clause in DIMACS as **-1 29 30 0**.

See: `KenKen#implicationsForAssignment`

4. Each cage must be no more than 3 cells.

5. In each cage, the numbers must combine through the cage's mathematical operation to equal the cage's target number.

We can represent whether or not a KenKen cube is solved through propositional logic by defining the following propositional variable: $p_{ijkm}$ = true if and only if the cell at x = $i$ and y = $j$ and z = $k$ contains the number $m$.
This definition allows us to represent **constraint 1** in the following manner:

$$(p_{0001} \vee p_{0002} \vee ... \vee p_{000n}) \wedge (p_{0011} \vee p_{0012} \vee ... \vee p_{001n}) \wedge (p_{0111} \vee p_{1112} \vee ... \vee p_{111n}) \wedge ... \wedge$$

$$\left(p_{(n-1)(n-1)(n-1)1} \vee p_{(n-1)(n-1)(n-1)2} \vee ... \vee p_{(n-1)(n-1)(n-1)n}\right)$$

for all $i, j, k$ in $[0, n-1]$ and all $m$ in $[1, n]$.

To model **Constraint 2** in our application, we iterate over every coordinate position in the 3-dimensional board and every number that can be at each of those coordinates. We mandate that if at position $(x, y, z)$ is the number $i$, then at $(x, y, z)$ the number cannot also be $j$, where $1 \leq j \leq n$ and $j \neq i$. For example, if $n = 3$ and $x = 0, y = 1, z = 2, i = 3$, we add the following clauses to the overall DIMACS expression:

1. $0123 \Rightarrow \neg 0121$

2. $0123 \Rightarrow \neg 0122$

Similarly, for **Constraint 3**, we mandate that if $(x, y, z)$ is the number $i$, then all cells $(x', y, z)$, $(x, y', z)$, $(x, y, z')$ cannot be equal to $i$, with $x', y', z'$ in range $[0, n-1]$ and $x' \neq x$, $y' \neq y$ and $z' \neq z$. For example, if $n = 3$ and $x = 0, y = 1, z = 2, i = 3$, then we add the following clauses to the overall DIMACS expression:

1. $0123 \Rightarrow 0101 \vee 0102$

2. $0123 \Rightarrow 0111 \vee 0112$

3. $0123 \Rightarrow 0021 \vee 0022$

4. $0123 \Rightarrow 0221 \vee 0222$

5. $0123 \Rightarrow 1121 \vee 1122$

6. $0123 \Rightarrow 2121 \vee 2122$

Regarding **Constraint 4**, we added this restriction on the size of cages to simplify the cube as we found it too difficult to work on arbitrarily large cages. In other words, this is self-imposed to simplify the problem.

For **Constraint 5**, we can define any cage to be true only when the arithmetic operation applied to all the cells in that cage equal the target number. For example, say that we have a cage of size 1 at position (0,1,2) with the addition operation and target number 2. Since this cage has a size of 1, we know that (0,1,2) must be equal to 2, or 0122, which can be written in DIMACS as **17 0** using our conversion included in Figure 3. For a cage of size 2, consider the positions (2,0,2) and (2,1,2), where the target number is 1 and the operation is subtraction. We first find pairs of numbers in the range [1,n] with $n = 3$ that subtract to 1: (1,2), (2,1), (2,3), (3,2). From here, we map the pairs into a dictionary. The pair of (1,2),

$$1 \mapsto 2$$

and (3,2),

$$3 \mapsto 2$$

and finally the pairs (2,1) and (2,3),

$$2 \mapsto 1, 3$$

We then translated the pairs into implications, so the pair (1, 2) was written as

$$2021 \Rightarrow 2122$$

which could further be rewritten as

$$\neg 2021 \vee 2122$$

finally in DIMACS form

**-61 71 0**

Similarly, the pair (3, 2) was written as

$$2023 \Rightarrow 2122$$

which was rewritten as

$$\neg 2023 \vee 2122$$

finally in DIMACS form

**-63 71 0**

Lastly, for the pairs (2,1) and (2,3), we get the following implication,

$$2022 \Rightarrow 2121 \vee 2123$$

which can be rewritten as

$$\neg 2022 \vee 2121 \vee 2123$$

which in DIMACS is

**-62 70 72 0**

For cages of size 3, the same logic was used to arrive at their implications. These implications were then evaluated together to see if they are satisfiable through the SAT solver using the Sat4J library. Our metrics for success was measured by passing JUnits tests in a polynomial run time.

# 3 Methodology

In implementing our KenKen cube to DIMACs program, we broke the KenKen board into three subproblems. Initially, we converted the board into a set of DIMACS clauses that require each cell to have one value.

Next, we created the DIMACS clauses for each cage of the board. In each of these cages, repetition was allowed, but we maintained that the axes x, y, and z containing a cell in the cage are distinct. To do this, we find all permutations of the possible values for each cell with the given operation. Then, we create a Map<Integer,List<Integer>> where the key is the number and the value is the list of possible values that the key implies exist in the cage.

For example, if we have a cage of size 2 with positions (2,0,2) and (2,1,2) with target number 1 and the subtraction operator, the list of permutations is as follows: (1,2), (2,1), (2,3), and (3,2). Next, we determine the possible unique numbers in the cage: (1,2,3). We then create the mapping of possible values which represent the implications of values for what the remaining cell can be filled with

$$1 \mapsto (2),\ 2 \mapsto (1, 3),\ 3 \mapsto (2)$$

This list of implications is then converted into DIMACS using our `KenKen#implicationsForAssignment` method. Likewise, the implications are converted into DIMACS strings, all of which is done in the `Cage#oneCellClauses`,

`Cage#twoCellClauses`, and `Cage#threeCellsClauses` methods. Finally, we iterate through the positions of the board and create implications that maintain the Latin Cube property of unique values in each of the axes. For example, if position (0,0,0) contains value 1, then in each axes x, y, and z, starting from that cell, there cannot be another 1. After the DIMACS clauses are formed for each cell, each cage, and the Latin Cube implications, the combined DIMACS String is then run through the SAT4J solver and the output string is displayed.

In order to quantify the success of our program, we established two metrics with our implementation. The first and most important was the accuracy of the output (i.e. was the output satisfiable and vice-versa). We are absolutely confident in the accuracy of the program, as we unit tested individual methods and tested the program with multiple complex example boards. Our second metric for success was efficiency, which we measured through Big-O analysis of our conversion from a KenKen board to a DIMACS string. Our program was able to solve the DIMACS string with SAT4J, an existing Java SAT solver.

# 4 Results

As mentioned above, we utilize two metrics for success. For our first metric, demonstrating that the DIMACS representation of a KenKen puzzle is correct, our application supports converting the output of the SAT-solver into a 3-dimensional array that represents the solution, and verifying that solution. In our tests (KenKenTests.java), we give three example KenKen boards of dimensions 2x2x2, 3x3x3 and 4x4x4 respectively. These example boards contain cages of all sizes (1 to 3) and utilize all mathematical operations. Our tests demonstrate that the the SAT-solver's solutions for these three examples are in fact correct, giving us confidence that our application correctly converts any 3-dimensional KenKen puzzle into a CNF expression in DIMACS format.

A secondary metric we used throughout this project is efficiency. When converting the board, there exist 3 stages that create DIMACS clauses independent of each other:

1. Generating the clauses that state a cell must contain a value: $O(n^4)$.
2. Generating the entire boards cage clauses: $O(m * n^3)$ where $m$ is the number of cages.
3. Generating the Latin Cube clauses that maintain distinct axes: $O(n^4)$.

Our implementation is successful if it creates the DIMACS string in polynomial time. When analyzing the runtime of the conversion, the most expensive step is generating the DIMACS clauses for the cages. The number of cages, m, will always satisfy $m > n$. For each cage, we must iterate $O(m * n^3)$, where $m$ is the number of cages and n is the number of cells, to find all the possible implications of the cage. Seeing as how this is still polynomial time, our solution is satisfactory. The other steps to generate the DIMACS string are each $O(n^4)$.

This means that as n approaches infinity, solving the cages takes the longest time, which makes sense as that is the most complex step. We have not taken steps to implement common optimizations for DIMACS conversion which leaves open a possible route for further research.

# 5 Summary

As shown, 3-dimensional KenKen boards were able to be contextualized into DIMACS format using propositional logic and boolean satisfiability. Through this contextualization into a DIMACS format, we were able to demonstrate that unconventional boards with multiple numerical rules are able to be manipulated in a way for a SAT-solver to generate a valid solution to the board's specifications. As a side-effect of our project, we were also able to create a latin-cube generator for any cube with a size $n$. Moving forward, a possible improvement to our project would be implementing a greater maximum size of cages, rather than limiting each cage with a size of three. In addition, as Sudoku boards share a large similarity with KenKen boards, a similar approach may allow for solving 3-dimensional Sudoku boards.

# References


- "ALL-NEW TECHNOLOGY." KenKen Puzzle Official Site - Free Math Puzzles That Make You Smarter!, www.kenkenpuzzle.com/.
- Erdem, Ozan. "Encoding Problems in Boolean Satisfiability." Constraint Satisfaction and Optimization, 17 Nov. 2019, ozanerdem.github.io/jekyll/update/2019/11/17/representation-in-sat.html.