KENKEN GENERATOR AND SOLVER

A PROJECT IN LISP

Emma Brunell

Artificial Intelligence II: Topics in Symbolic AI

SUNY Oswego

May 13, 2019

Contents

1 Background

1.1 A Brief History

KenKen is a mathematics game developed in 2004 by a Japanese math teacher named Tetsuya Miyamoto. The name of the game literally translates to "the game that makes you smarter", or "cleverness squared". Miyamoto developed KenKen to help his students build their arithmetic and problem-solving skills. His intent was that his students would gain knowledge in a fun way, so that they would not dread practicing their math.

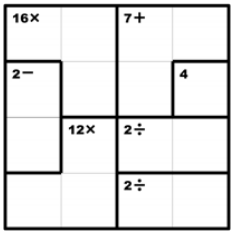

Figure 1: A blank 4x4 game of KenKen

1.2 How to Play

KenKen is similar to the popular game Sudoku in many respects. The game is laid out in a grid, typically ranging in sizes 3-by-3 to 6-by-6. Within the grid, there are traditionally heavily outlined cages, which are simply groups of squares that neighbor each other. Each cage has a number, called the target, and a mathematical operator in one of the corners. The operators that are used in the game are addition, subtraction, multiplication, and division.

The object of the game is to fill in the grid with numbers 1 to the size of the board. No number can be repeated within its row or column, similar to Sudoku. The extra piece to KenKen is that the numbers within each of the cages must achieve their respective target number. Numbers can be repeated within the cages, as long as the previous row/column rule is still being followed. Cages with repeated numbers in them happen relatively often, as the cages tend to span multiple rows or columns.

Addition and multiplication cages have no size limit on them. However, division cages can only be size two. Subtraction cages are a point of disagreement; some say that subtraction cages should also only be size two, while others believe that there does not

need to be a limit as long as the result of the cage stays within the realm of positive integers.

2 Generating a Game of KenKen

2.1 Latin Squares

A Latin Square is an n-by-n grid filled with n symbols, each of which are distinct in a given row or column. The Latin Square was originally developed by Leonard Euler. It is the basis for many combinatonics problems, and thus the basis for a game of KenKen. I implemented the below algorithm (Figure 2) to make a balanced Latin Square for which to start generating a game. I represented this Latin square as a list of lists, where the outer list position corresponded to the x-axis, and the position of the number within the sub-list corresponded with the y-axis. Thus, the bottom left corner of the square was position (0,0) and the top right corner was (n,n).

1	2	n	3	n-1	4	...
2	3	1	4	n	5	...
3	4	2	5	1	6	...
⋮	⋮	⋮	...	⋮	⋮	⋮
n-1	n	n-2	1	n-3	2	...
n	1	n-1	2	n-2	3	...

Balanced square algorithm (UF).

Figure 2: A visual description of a balanced Latin Square

2.2 Polyominoes

Polyominoes describe a set of shapes, usually squares, which are touching on at least one side. A common form of polyominoes is in the game of dominoes, where there are two squares neighboring each other. Additionally, the object of the computer game of Tetris is to align differently shaped polyominoes together to form a solid line. I decided to think of the idea of polyominoes to form the cages within the KenKen game. To do this, I generated a list of available coordinates of the game board. I iterated through the list of coordinates, and found the north, south, east, and west neighboring coordinates for each. I added one of the neighboring coordinates to the initial coordinate, if it had not already

been placed in a cage. I used this method to generate cages up to size 3. I decided to only make cages up to 3 squares long, because looking online at sample games, I saw that there were very few with larger cages.

2.3 Target Numbers and Operators

After placing the polyominoes "on top" of the generated Latin Square, the next step was to create the target number and operator to serve as a clue for each cage. Monomino (single) cages recieve no operator, just the target number in the corner of the cage; the target number in their case is just the solution to that particular square. One of the available operators was randomly applied to each of the other cages, as long as it would result in a whole number above zero. This operator and the result were what would be placed in each cage.

2.4 Graphics

I also chose to represent the game graphically, for ease of use, as it is difficult to look at an internal represenation of a game of KenKen. Much of the problem-solving process is visual and relies on pattern recognition. I used Professor Graci's SimpleViewerAgent Java and Lisp programs, with a few modifications to make it fit the requirements of a KenKen Game. This program works by writing out coordinates and the information within the coordinate to a text file using Lisp. The Java program then reads the text file as it is edited and displays the data using Swing. My implementation of KenKen is unique in the graphical aspect because I display the cages as groups of squares filled in with the same color. I believe there could be some connection between color and solving math problems more easily, also, it makes the game a little more fun, especially in the context of elementary students learning arithmetic.

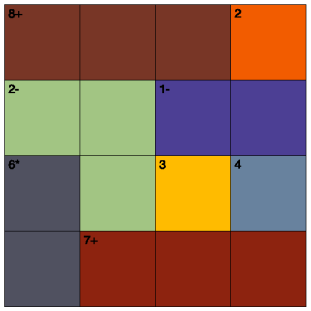

Figure 3: A generated 4x4 game of KenKen from my program

3 Writing a Solver

3.1 Heuristics

I broke my solver down into two main parts, the first part of which relied on some human techniques for solving a KenKen game. I relied on my own experience of playing the game before to decide on some reliable methods for solving KenKen or narrowing down the search field:

1. One Left In Row/Col: Check to see if any of the columns or rows only have one square not yet filled in: if so, the solution to this square is just the number missing from 1 to n in that row or column.

2. Get Factors: Fill in each incomplete square with the factors available to it that would assist in completing the cage. This narrows down the search space as the numbers 1 through n are not applicable for each square.

3. Eliminate Options: If a possible factor has already been used within that row or column, eliminate it from a square's list of potential solution options.

4. Find Only Option: If, within a list of factors in a square, this location for a factor is the only possible place it could go within a row or column, place it there and eliminate all other factors from the square.

I iterate through these heuristics several times each, as often if one heuristic leads to a square or cage being solved, it allows the other heuristics to also make progress towards the solution. However, if a solution has not been reached by the end of four iterations, it is likely to not be able to succeed using just these techniques. Then, the state of the board is solved using a depth-first search.

3.2 Depth-First Search

I use a modified depth-first-search with Heuristics 1 and 2 to try to arrive at a goal state. When coupled with the initial search on the heuristics, it is much more successful at solving than just used on its own; however, it still does not always correctly arrive at the goal state. I believe this is because it is difficult to check whether a number will actually be the solution to completing a cage, if the cage is not already one step away from being solved yet.

4 Project Results

4.1 Running the Program

Before running the program, the files *kenken.l*, *simpleviewerагent.l*, and *SimpleViewerAgent.jar* must be located within the same directory. To run the graphics part of the program, execute in a terminal:

```
$ java -jar SimpleViewerAgent.java
```

Within CLISP or another Lisp development environment, execute:

```
[ ]> (load "kenken.l")
[ ]> (demo *your-size-grid-here*)
```

This will begin sending commands to the Java program, and you will be able to watch it generate a game and then solve it. It can be interesting to watch, because due to the heuristics implementations, the program often solves the puzzle in a logical way, similar to the steps a human would take to solve it.

4.2 Reflection

I saw a lot of improvement with this program's results after I decided to integrate a heuristic solver along with the depth-first-search. Now, the program almost always is able to solve up to a 5-by-5 game, and has a much higher rate of success with the 6-by-6 grids. I think that the search would be improved by adding even more heuristics to the program, which is something I would like to experiment with in the future. Some heuristics that I have thought of include favoring filling in multiplication/division and size 2 cages first (due to having less possible factors), and adding a heuristic that finds cages with only one square left and solving them. There are also more "expert" techniques online, such as summing up the rows and applying a formula to narrow down the search space. It would be interesting to see if I could implement a heuristic-only solver. I really enjoyed learning about the process of solving this puzzle, and was pleasantly surprised at how much I liked thinking about heuristics for my program to use. I also liked thinking about the more complex math principles that are behind a game of KenKen, aside from the basic arithmetic on the surface. I feel as though I have definitely improved my KenKen skills in general, and maybe I won't end up needing to use my solver to help me out after all!

5 References

1. Hueston, D. (n.d.). 'Teaching without teaching': Creator of KenKen puzzles cultivates young minds through math and fun. Retrieved from https://www.japantimes.co.jp/news/2019/01/18/national/teaching-without-teaching-creator-kenken-puzzles-cultivates-young-minds-math-fun/

2. Latin Square. (n.d.). Retrieved from http://mathworld.wolfram.com/LatinSquare.html

3. Latin Square Design: Definition and Balanced Latin Square Algorithm. (2017, October 12). Retrieved from https://www.statisticshowto.datasciencecentral.com/latin-square-design/

4. Polyominoes (n.d.). Retrieved from https://home.adelphi.edu/ stemkoski/mathematrix/polys.html