

VISUALLY PARSING A KENKEN BOARD

Brandon Packard
CU-GAME, Pennsylvania Western University
bpackard@pennwest.edu

ABSTRACT

Creating bots that play games or performing similar tasks has become an increasingly popular area of research. Traditionally, these bots need to have access to the code or back-end of their target task in some way. In this paper, we present a parser which is capable of breaking down a KenKen board from an image on a screen to a representation that an Artificial Intelligence (AI) could use to play the game - somewhat emulating the way that a human breaks down an image of a KenKen puzzle into a representation that allows their brain to solve it. We believe that this is an important step towards being able to create bots that can play games or perform similar tasks for situations in which it is impractical or impossible to have direct access to the game/task via traditional methods.

1 Introduction

The game of *KenKen* is a grid-based puzzle game. In a way, it is of similar spirit to Sudoku, but a step up in complexity. As such, the game of KenKen has features that make it harder to algorithmically parse than other grid-based games like Minesweeper or Sudoku. We will now describe the game of KenKen in detail, for any readers who may be unfamiliar with it or need a refresher on how the game works.

1.1 The Game of KenKen

In the super popular game of Sudoku, a user must place the numbers 1 through 9 onto a grid, only using each number once in each row, column, and 3x3 "region". KenKen, shown in Figure 1, is a similar sort of game but adds an extra layer of game mechanics. Each number must be used once per row and once per column, as with Sudoku. However, there are no fixed-size "regions" of any sort. This allows KenKen to have board sizes, such as 5x5, that don't really make sense for Sudoku. Furthermore, in Sudoku, some of the numbers are filled in as clues. In KenKen however, the clues take the form of a series of "cages" that enclose some of the grid cells. All cages have a number associated with them. Some of them only enclose a single cell and do not have an operation - essentially just giving you a number on the board. Most cages, however, have an associated math operation. These operations are as follows:

2÷		20×	2-	
2-			5+	
6×	3-	2-		60×
			11+	
4-				

Figure 1: An example 5x5 KenKen board. The yellow square is just the one that the user has selected to enter a value.

- Addition - If you add all of the cells together, the result must be the number associated with the cage.
- Multiplication - If you multiply all of the cells together, the result must be the number associated with the cage.
- Subtraction - If you subtract the smaller number from the larger number, the result must be the number associated with the cage.
- Division - If you divide the larger number by the smaller number, the result must be the number associated with the cage.

Note that for all cages, the numbers can be in any order. For example, for the cage labeled 6X on the left edge of the board in Figure 1 has to contain a 2 and a 3, but they can be in either order. The rows and columns then need to be used to determine which number is in which position.

In the 5 x 5 board shown in Figure 1, the largest cage is 3 cells large. Although the subtraction and division cages are always just 2 cells in size, the multiplication and addition cages can be considerably larger. In the 9 x 9 board shown in Figure 2, notice that there is a cage in the bottom left that is labeled 3240X and contains 6 cells. These large and oddly-shaped cages make parsing a KenKen board much more difficult, which we will now discuss.

30x		14+		2-		20+	8-	
4÷				2-			7	3-
	3-		135x				6x	
2÷		224x	3-		8-			21x
3240x			6+			30x		
			24+	4-	35x	15+		8
							6x	19+
243x				16+	7-	1-		
2-								

Figure 2: An example 9x9 KenKen board. The yellow square is just the one that the user has selected to enter a value.

1.2 A Brief Discussion on Creating AI

The goal of this work is to create a parser for a KenKen board, to take it from an image on a computer screen to a back-end representation that scripted solver and/or machine learners could use to solve a KenKen board. Typically, there are two ways to write an artificial intelligence, or AI, for a game. The first is to somehow tap into the game’s code or data. There are some games where people have created ways to do this (for example, Starcraft II [1]), but they are in the vast minority. There are also some games with Application Programming Interfaces (or APIs), that are essentially a built-in way to interact with the game; however, these are also relatively very rare. As such, this would have to be done by the person making the AI. Not only would this take a lot of time before the AI can even be started, but it isn’t feasible at all in some cases.

The second option is to completely re-implement the game. KenKen is a relatively simple game, so it would be feasible to do so. However, it would take many hours of work before the main focus, the creation of an AI, could even be attempted. Additionally, many games take entire teams of programmers months or years to complete - there’s no way a single or small team of researchers could feasibly recreate it in any reasonable amount of time.

As such, there needs to be a way to give access to these games to an AI solver without employing either of these methods. The main focus of our paper is taking an important step in that direction by creating an internal representation of a KenKen board using only an image of one on the screen. Not only would this allow the community to create a solver without any access to KenKen’s code, an API, or recreating the game, but it would also to an extent mimic the way that humans would solve the boards. In fact, by parsing the board visually,

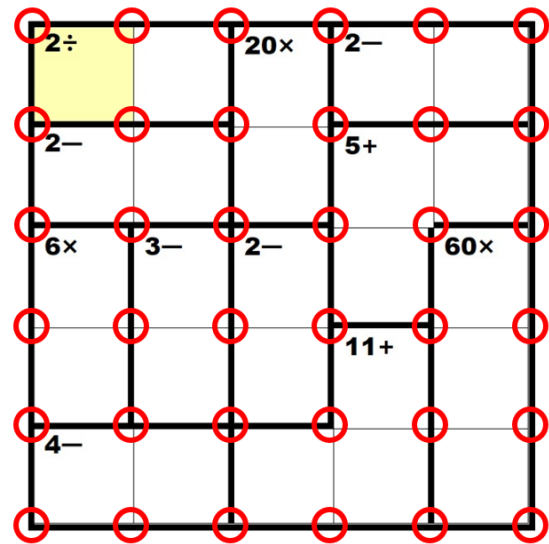


Figure 3: An 5x5 KenKen board with the intersections marked. These intersections are pivotal in parsing the board.

we are solving it under the same constraints that a human has to face. In the end, we are able to create a parser that outputs a representation of a KenKen board that a solver would then be able to use to play the game.

We start this paper by talking about some of the related work in this area in Section 2. Next, we discuss exactly how the different elements of the KenKen board were parsed in Section 3. Finally, we end with conclusions, limitations, and ideas for future work in Section 4.

2 Related Work

As far as we are aware, no attempt to visually parse a KenKen board from an image into a more usable representation has been done in the literature, but some work with parsing games has been done and some KenKen solvers have been made.

2.1 Parsers

The idea of visually parsing a game to create a solvable representation of that game is relatively new. In the past, similar techniques have been used to parse the games of Minesweeper[2] and Rullo[3]. Unlike KenKen, Minesweeper is very straightforward, because it is a nice, neat grid with no additional features. Rullo is a little more complicated, adding row and column labels as well as double digit numbers into the mix, but KenKen is much trickier due to the cages on the grid (discussed in Section 1.1).

There are also some machine learning domains that rely on the visuals of a game, such as the Arcade Learning Environment (ALE) [4]. This domain works by using the individual pixels of a simulated Atari 2600 as the features for learning,

whereas we attempt our task by searching for pre-specified images on the computer screen. Additionally, KenKen is not an Atari game so it falls outside the scope of that library, and it has a much higher resolution than Atari games, which would make reconstructing the board pixel by pixel an extremely arduous task. Furthermore, using visual parsing for scripted AIs is not well represented in the literature.

2.2 Solvers

There are multiple instances in the literature of KenKen solvers. For example, Gerlach solves the problem by finding all of the viable sets that could be in the cages, and then finds the combination of them which solves the puzzle [5]. Others have used Hybrid Genetics Algorithms [6], Hopfield neural networks [7], or integer programming [8]. However, none of these solvers visually parse the board. Although not specified, they most likely use pre-encoded KenKen boards.

3 Parsing The Board

Traditionally, most AIs for games are written in one of two ways. The first method is that the game's source code is found, ported, or rewritten (such as with MochaDoom [9] or the Super Mario version used in the 2009 Mario Artificial Intelligence Competition [10]). The second method is to read the program's memory to figure out key values, and then edit those values or inject code into the running program to send commands [11]. However, each of these has limitations. For the former, a researcher would have to either find the source code or essentially re-write it themselves before they could get started on creating an AI for the game, which can easily take more time than creating the AI itself in many cases. For the latter, hacking into the memory and finding out bit by bit (or rather, byte by byte) what values correspond to what the researcher wants to know can be a very time-intensive task, and directly interacting with a program's memory is a very difficult task. As such, there are many tasks for which neither of these methods are very feasible.

Rather, it is valuable to be able to visually parse a game, taking it from a visual representation meant for humans to a data representation that could be used to solve the game. By doing so, we can enable the creation of AIs that can play a game without having to recreate the game itself. KenKen is a particularly interesting example, as the way the cages and values are arranged make it much more complicated to parse programmatically than a grid-based game such as Minesweeper.

3.1 PyAutoGui

The main tool used to this end was a Python library known as PyAutoGui [12]. This library allows for simple GUI-related tasks, such as typing keys from the keyboard or clicking a position on the screen. The most important function for this work, however, was PyAutoGui's locateAll() function. This





Lines	
Corners	
Ts	
Cross	

Figure 4: The 11 types of markers that can be found at intersections.

function searches for an image on the screen. It then returns the coordinates and size of all images that matched, as a 2D list. Although it has disadvantages (see Section 4.1 for more details), it is a very powerful tool for this purpose.

3.2 Parsing the Grid Structure

The first, and hardest, aspect of parsing the game board is parsing the grid of cages. As seen before, the cages are all randomly sized and shaped. A first instinct in how to approach this would be to have images of each cage, and search for those. However, there are far too many possibilities for it to be done this way - the image would have to include the number and operation as well, and each image that we search for has to be taken by hand via screenshots.

Instead, the strategy we chose is to not look at the cage themselves, but at the intersections between squares of the grid, as shown in Figure 4. When looking at it this way, there are 11 possible "markers":

- Line - Has lines on 2 opposite sides of the intersection point. Can be horizontal or vertical.
- Corner - Has lines on 2 adjacent sides of the intersection point. Can be in one of 4 rotations.
- Ts - Has lines on 3 sides of the intersection point. Can be in one of 4 rotations.
- Cross - has lines on all 4 sides of the intersection point.

However, there is a catch. The bottom and right edges of the board have a very thin line running through them. Visually, this makes almost no difference. However, it does completely throw off the image matching. As such, some markers have 2 or 3 different variants. All in all, 22 images of markers had to be taken and cropped by hand. These images were then sorted into 11 lists, one for each of the marker types above.

This means that we now have unsorted lists for each marker. However, to have them represent the board, we need to know where each marker is. To do this, we first find the corner that is in the top left of the board, by finding the one with the lowest x and y coordinate. We also find the width of the board in pixels, by finding the bottom right corner as well, and subtracting its x-value by the top-left corner's x-value. We then divide the width in pixels by the number of grid squares wide the board is, which gives us how wide far apart the markers are on the board. This process is illustrated in Figure 5.

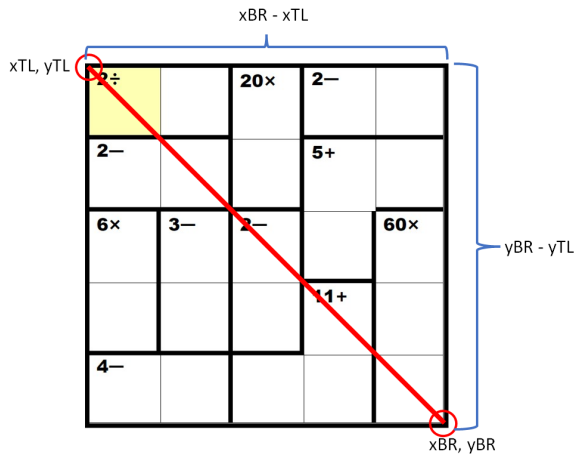


Figure 5: An illustration of calculating the height and width of the board using only the pixel coordinates of the top-left corner (TL) and bottom-right corner (BR).

Once we have that, we are ready to create our grid of markers, as a 2D list. We create a 2D list that is 1 larger than the size of the game board. For example, if the board is 5x5, we create a 6x6, because there is always one more marker per row/column than there are grid squares. Now we have 11 lists of markers with pixel positions, but we need to figure out where in our 2D list each of those markers should be. To do this, we first subtract the x-coordinate of the top left marker (TL) from the x-coordinate of the current marker we are examining (*marker*). We then divide by how far apart markers are on the board (*spacing*) and round to the nearest whole number. We then do the same thing with the ys, as seen in the formulas below:

- $i = \text{round}((\text{marker.x} - \text{TL.x}) / \text{spacing})$
- $j = \text{round}((\text{marker.y} - \text{TL.y}) / \text{spacing})$

The values of i and j are then used as indices into the 2D list. A string is stored at that location to indicate which of the 11 markers it is. At this point, we have a 2D list of markers. An example of this is shown in Figure 6.

In order to turn this into a meaningful representation, we then have to employ a key insight. When looking at a square on

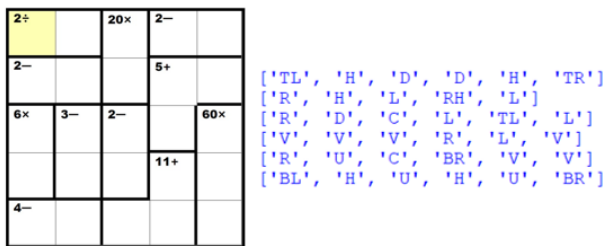


Figure 6: An image of the markers placed into a 2D (right) next to the board (left). Lines are {H,V}, corners are {BL,TL, TR, BR}, Ts are {U,R,D,L} and crosses are {C}.

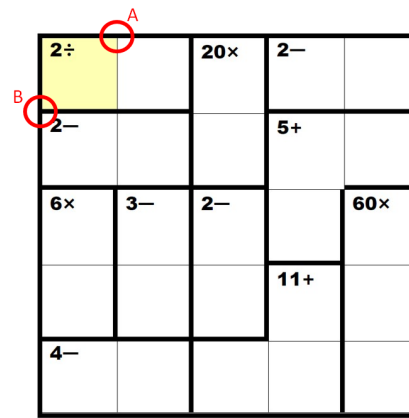


Figure 7: An examination of the top left cell. Marker A is a horizontal line, so there is no line between this square and the square to the right of it, meaning they must be in the same cage. Marker B is a T facing right, so there is a line between the top left square and the square below it, meaning they are not in the same cage.

the grid, we can use the markers around the square to tell whether each of the squares in the 4 cardinal directions are part of that same cage or not! For example, take the top left square in Figure 7. If we look at marker A, it is a horizontal line. This means that the square to the right is in the same cage, because there is no line dividing them. If we look at the marker labeled B, it is a T facing to the right. This means there is a line between the top left square and the square below it, and therefore those squares are not part of the same cage.

We then repeat this process for each new square that we found to be in the same cage, in flood-fill-esque manner. Once we have found every square in the same cage, we give it a number as a unique identifier. We then repeat this process for each square on the grid, but if a square has already been assigned to a cage then we skip over it. We write the resulting cage numbers into a 2D list the same size as the game board. The result is a representation of the cages that could be worked with by a solver, as shown in Figure 8, next to the board it represents (which is the same one we've been using, but put here again for ease of comparison).

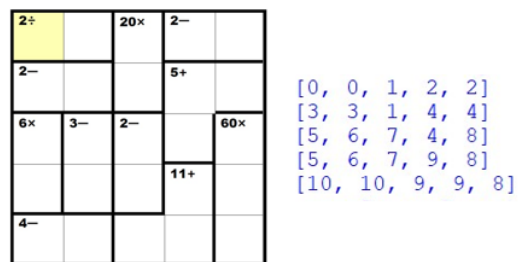


Figure 8: An image of how the cages are represented in our 2D list (right) next to the board it represents (left). Positions with the same number indicate board squares that are in the same cage.

[0, 0, 1, 2, 2]	0 /	0 2
[3, 3, 1, 4, 4]	1 *	1 20
[5, 6, 7, 4, 8]	2 -	2 2
[5, 6, 7, 9, 8]	3 -	3 2
[10, 10, 9, 9, 8]	4 +	4 5
	5 *	5 6
	6 -	6 3
	7 -	7 2
	8 *	8 60
	9 +	9 11
	10 -	10 4

Figure 9: The representation of the cages (left), operations (middle), and values (right) for a KenKen board.

3.3 Parsing the Operations

The hardest part is done, but we still need to get the number and operation associated with each cage. The process for both of these is very similar. First, we parse the screen for the 4 math operators (addition, subtraction, multiplication, and division), and sort them into lists. Next, we use the same formula as we did with the markers, to find out what grid square that operator is. We can then use our 2D list of cages to determine to which cage that operator belongs. We store this as a list of pairs, with the cage number and the corresponding operation. One thing to note is that it's possible for a cage to have no operation, if it is only 1 square large. In this case, we store the operator as None. A visual example of this representation can be seen in Figure 9, again using the same board as before. Notice that we have the cage number and the corresponding operation implemented as a list of pairs.

3.4 Parsing the Numbers

Finally, we need to get the numbers associated with each cage. This process is very similar to the operations, but with one catch - numbers can be more than one digit! As such, we need a way to "reconstruct" the numbers. To do this, we follow the same process as the operations, but instead of just 1 thing, we might get multiple numbers for the same cage! As such, we store these numbers in a list, in no particular order.

Once we have the lists of numbers for all of the cages, we loop through the cages one at a time. We sort the list of numbers for each cage according to their x location on the screen in pixels. We then make the farthest right value be the ones place, the next farthest right be the 10s place, and so on until we have consumed all of the numbers. This gives us the true value stored in that cage. An example of this representation is also in Figure 9, and consists of the cage number and the value associated with that cage, as with the operations.

3.5 Final Representation

We now have a final representation of our board, all shown together in Figure 10. Our 2D list tells us which cage each

2÷		20×	2-	
2-			5+	
6×	3-	2-		60×
			11+	
4-				

[0, 0, 1, 2, 2]	0 /	0 2
[3, 3, 1, 4, 4]	1 *	1 20
[5, 6, 7, 4, 8]	2 -	2 2
[5, 6, 7, 9, 8]	3 -	3 2
[10, 10, 9, 9, 8]	4 +	4 5
	5 *	5 6
	6 -	6 3
	7 -	7 2
	8 *	8 60
	9 +	9 11
	10 -	10 4

Figure 10: A final representation of a KenKen board, combining the cages, operations, and numbers all into one.

grid square is in, and the two lists tell us the number and operation for those cages. As such, we have everything we could possibly need to create a solver for the board, and our visual parsing of a KenKen board is complete.

Note that our parser works for any size of KenKen board, as long as it fits on one screen and all of the correct images to search for have been acquired and saved.

4 Conclusions

In conclusion, there is value in being able to visually parse what is on a computer screen and turn it into a representation that is usable by a computer program. One contemporary application of this is writing AI solvers for video games. Both traditional methods of writing an AI - recreating the game or somehow accessing the source code or data of the game, can be quite arduous and are not feasible in many cases. Instead, we can visually parse some games and then have an AI work with the extracted representation.

This work is an important step in that direction. Despite KenKen being a grid-based puzzle game with easy-to-understand rules, visually parsing it algorithmically is a somewhat complex problem, and we hope that insights gained from doing so will lead to the visual parsing of more and more complex games.

4.1 Limitations

PyAutoGui is a great tool for our purposes, but it does have some limitations. First, it requires very precise matches. This often leads to having to take multiple images of what is logically the same thing. For example, the KenKen board has a line on the bottom and a line on the right, as discussed in Section 3.2. This causes us to need two different images of each marker that could appear on the bottommost or rightmost edge of the board. Additionally, some of the markers are different sometimes for no apparent reason. For example, if you look back to Figure 2, notice that the top left of the 60X

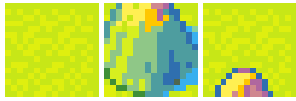


Figure 11: An illustration of more complicated tiling in games. On the left is a plains tile with no mountain under it. In the center is a mountain tile (notice how the entire mountain does not fit on the tile), and on the right is a plains tile that has a mountains tile below it. On a 16x15 tile map, a total of 148 individual tiles were identified due to these combinations, before factoring in that a variety of units could be placed on each tile, leading to a very high tiling complexity.

cage in the bottom right appears to be partially missing. Each of these little differences requires more separate images to be able to correctly capture the board. In this case, it turned what should have been 24 separate images into 35 separate images. This is doable, but in some cases it could increase the number of images tenfold or more and become a serious issue. Furthermore, for the 9x9 board, all of the images had to be completely retaken, because they are smaller and in some cases proportioned differently.

In addition, PyAutoGUI is very slow, sometimes taking a second or more to search the screen for a single image. PyAutoGUI does have an option to only search part of the screen, but as previously shown in the literature [13], it doesn't make a significant difference in practice. For puzzle games, board games, etc., incredibly slow parsing is acceptable. For real-time games, another method may need to be employed, or else the situation will change by the time it is parsed.

Finally, some games just tile in a "non-regular" way. For example, some games such as Advance Wars perform more complicated tiling. This game has plains tiles and mountains tiles. If a mountains tile is right below a plains tile, it changes the plains tile (See Figure 11). This leads to some games either having far more tiles than one can reasonably encode. In other games, such as Creeper World 2, the tile for the same object can sometimes be different heights (See Figure 12).

4.2 Future Work

The most obvious area of future work would be to write a solver that uses our representation of the game, to measure the true usability of our representation - even though it contains everything that is needed, is there a better way to store it? Another exciting area of future work would be to continue



Figure 12: Another illustration of more complicated tiling. This is a set of 4 rocks from Creeper World 2. The right image, which has an overlay put on it, indicates that the top two rock tiles in black are smaller than the bottom two rock tiles in red (23 versus 24 pixels tall). This irregularity makes visually parsing the game much harder.

parsing more and more complicated games into representations that can then be worked with by programs. Hooking our visual parser into a machine learning algorithm and seeing if we can train a learner to play KenKen would also be an interesting endeavor. We would also like to explore the idea of making a more robust visual parser that is either faster than PyAutoGui or can handle irregularities much more easily. Finally, we would like to be able to parse puzzles that were scanned into a computer - a far more complex problem.

References

- [1] sc2automation, Tutorial 1: Getting started, https://blizzard.github.io/s2client-api/md_docs_tutorial1.html, 2020, accessed: 2024-01-31.
- [2] B. Packard, Solving a minesweeper board by visually parsing it, in *35th Annual Conference of The Pennsylvania Association of Computer and Information Science Educators*, page 32.
- [3] B. Packard, Solving a rullo board by visually parsing it, in *38th Annual Conference of The Pennsylvania Association of Computer and Information Science Educators*.
- [4] M. G. Bellemare, Y. Naddaf, J. Veness, M. Bowling, The arcade learning environment: An evaluation platform for general agents, *Journal of Artificial Intelligence Research*, 47:(2013), 253–279.
- [5] J. R. Gerlach, Solving kenken puzzles—by not playing, *Pharmaceutical Programming*, 3(2):(2010), 92–98.
- [6] O. Johanna, S. Lukas, K. V. I. Saputra, Solving and modeling ken-ken puzzle by using hybrid genetic algorithm, in *International Conference on Engineering and Technology Development (ICETD)*, volume 2012 (2012).
- [7] M. Fitzsimmons, H. Kunze, Combining hopfield neural networks, with applications to grid-based mathematics puzzles, *Neural Networks*, 118:(2019), 81–89.
- [8] V. Melkonian, et al., An integer programming model for the kenken problem, *American Journal of Operations Research*, 6(03):(2016), 213.
- [9] AXDoomer, Mochadoom, <https://github.com/AXDOOMER/mochadoom>, 2015, accessed: 2020-01-01.
- [10] S. Karakovskiy, J. Togelius, Mario ai competition, <http://julian.togelius.com/mariocompetition2009/>, 2009, accessed: 2020-01-01.
- [11] blizzo, Code injector, <https://www.ownedcore.com/forums/world-of-warcraft/world-of-warcraft-bots-programs/wow-memory-editing/136913-code-injector-code-exe.html>, 2008, accessed: 2020-01-01.
- [12] A. Sweigart, Welcome to pyautogui's documentation!, <https://pyautogui.readthedocs.io/en/latest/>, 2016, accessed: 2020-01-01.
- [13] *Proc. 25th Annual Spring Conference of the Pennsylvania Computer and Information Science Educators (PACISE 2020)*.