Model for
Human Killer Sudoku Solving

Bjorn van de Sand

Supervisor: Dr. T.B. (Tomas) Klos
Second reader: Dr. S. (Silja) Renooij

11-10-2018
Bachelor Artificial Intelligence
Utrecht University
7.5 ECTS

Abstract

I research how Killer Sudokus are solved by humans, primarily through introspection. I settle on modelling the process with a Priority Queue for steps, or Rules. I thoroughly examine these Rules and Killer Sudokus in general. The model has merit, for it is able to solve most puzzles of low to medium difficulty and can represent varying preferences for the order of the solving process. It even allows for emulating logical errors humans might make by changing priorities. The model is lacking in that its ruleset is not yet strong enough to solve problems that experienced humans can solve. The approach seems sound, but requires refinement and additional work.

Preface

When planning this period's project I got together with my supervisor and decided to try and find a subject that I might, proverbially speaking, lose myself in, as was the case in earlier assignments during my bachelor courses. This was accomplished all too well and I happily spent many a day lost in peering over puzzles, making notes of my thought process and designing and programming my solver.

Writing my bachelor thesis has been no simple undertaking and I would like to extend special thanks to my VIP support group, that stuck with me through all the trials of this past year: David, Joni, Merel & Robin. You are literally the best and I wouldn't have made it to this point without your support.

Last but not least, my sincere gratitude to my supervisor Tomas Klos for his solid feedback and advice, generous time investment and contagious enthusiasm for puzzle solving.

Table of contents

1 Introduction

The Sudoku puzzle has been teasing brains for well over a century, though it has only become popular in the mainstream for the last two decades. Many people all over the world, including myself, have entertained themselves trying to solve them. There is even a World Sudoku Championship that has been organized annually since 2006. This popularity has drawn the eye of the academic world and there has been a considerable amount of interest in automating the solving of these puzzles, usually with the goal of doing so as swiftly as possible. Since the problem of solving a Sudoku has been proven to be NP-complete (Aaronson, 2006, p.16), more heuristically based solvers are frequently of interest. During my own AI bachelor's Computational Intelligence course, my fellow students and I designed and implemented a multi-threaded hill climbing algorithm to solve arbitrarily sized Sudoku puzzles. While that was an interesting and educational experience, I have gone for a more intuitive and logic based project this time.

The classic Sudoku puzzle has many variants like the Tectonic, Calcudoku, Samurai Sudoku, Killer Sudoku and Hyper sudoku. I have chosen the Killer Sudoku as the subject of my thesis for multiple reasons:

- There has been little research on this specific variant, although there has been some (Haynes & Corns, 2013).
- The problem is simple enough to build a basic solver within the time allotted.
- I find the mix of Mathematical and Logical reasoning required interesting.
- Most importantly, it is relatively new to me, and this allows me to look at this problem with fresh eyes and more consciously keep track of the solving process as I discover it myself.

I will cover the specifics of Killer Sudokus in the following chapter. The Calcudoku is another variant specifically considered, but its Rules allow for operations like subtraction and division, requiring a far too sophisticated solver to build, within my limited timeframe. Even so, I built the solver keeping this variant in mind as well, and a future conversion of my algorithm should require only minimal adaptation.

For my thesis, rather than research optimal solving algorithms, I have looked into how humans solve Killer Sudokus and I have attempted to model this process. I am less concerned with speed and efficiency of automated solving and more concerned with using automation as a tool to model the procedure humans may use. This approach can be categorized within the Thinking Humanly definition of Artificial Intelligence (Russell & Norvig, 2010, p.3). My goal is not only to create a program that replicates input-output behaviour from an unsolved puzzle to a solved one. I also aspire to model the internal workings of the human solving process. If I succeed in building such an algorithm and it exhibits the solving behaviours of humans, I will have created one possible model for how humans actually solve Killer Sudokus.

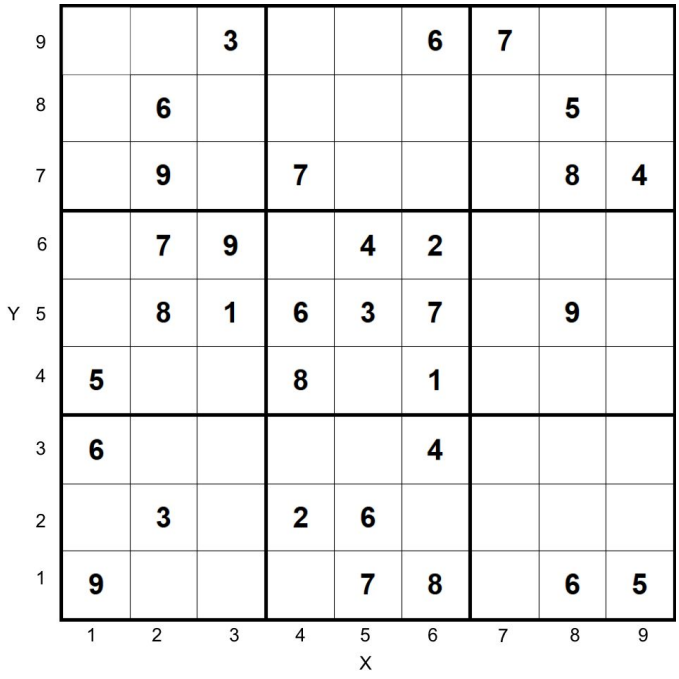
While it is difficult to determine precisely how humans solve any kind of puzzle, and I have no large scale studies or brain scanning equipment at my disposal, I shall start out with introspection and, additionally, the questioning of others who are either in the process of solving, or are already familiar with, Killer Sudokus. It is my belief that my lack of experience with these puzzles and my Artificial Intelligence background make me a suitable subject for the introspective method of investigating the human solving process.

In the next section on Sudokus, I will explain the basics of Sudokus in general and Killer Sudokus specifically. The third chapter Algorithm concerns the algorithm itself and it is there that I will expound my algorithm's origin, design, composition, and control flow. Chapter four Results covers the data gathered from having the algorithm solve puzzles of varying difficulty. The fifth and final chapter Conclusions contains my evaluation of this project as a whole and my recommendations for the future.

2 Sudokus

2.1 Regular Sudokus

In order to fully understand Killer Sudoku solving it is useful to first have an understanding of the original Sudoku problem. “Sudoku” is a Japanese abbreviation of “suuji wa dokushin ni kagiru”, which means as much as “the numbers (or digits) must remain single”. This is a reasonably descriptive name as a Sudoku is a two dimensional logical number placement puzzle. The objective is to fill a square Grid, without allowing more than one instance of any one Value per Row, Column or Block. Although some Sudokus also enforce the previously mentioned constraint in the Diagonals, the example provided in Figure 1 below, does not.



		3			6	7		
	6						5	
	9		7				8	4
	7	9		4	2			
	8	1	6	3	7		9	
5			8		1			
6					4			
	3		2	6				
9				7	8		6	5

Figure 1: A Regular Sudoku

A Sudoku Grid consists of N^2xN^2 (typically 9x9) Cells and therefore contains 81 (N^4) Cells distributed over a square Block. The Values 1 through 9 ($Dimension^2$) must be distributed over these Cells. Although numbers are used, solving the puzzle requires no calculations and any set of distinct symbols could have been chosen.

There are 6670903752021072936960 possible Sudoku Grids that meet the constraints (Felgenhauer & Jarvis, 2006, p.7). Even when culling those Grids that could be considered symmetrical versions of other Grids (through sequences of actions such as relabelling symbols or rotating Grids) we are still left with 5472730538 unique possibilities. Since any of these would be an acceptable solution for an empty Grid, a Sudoku is always partially completed and only considered well-posed when sufficient Cells are given so that only one unique solution remains.

It quickly becomes clear that one need never solve the same puzzle twice. Even so, a great many variant puzzles have gained popularity.

2.2 Killer Sudokus

The Killer Sudoku, which is the subject of this thesis, is a Sudoku variant. An example is shown in Figure 2 below. The basic layout is the same and all the rules that apply to a regular Sudoku also apply to Killer Sudoku. It is indexed from the bottom left to the top right with an X and a Y axis as a graph would be.

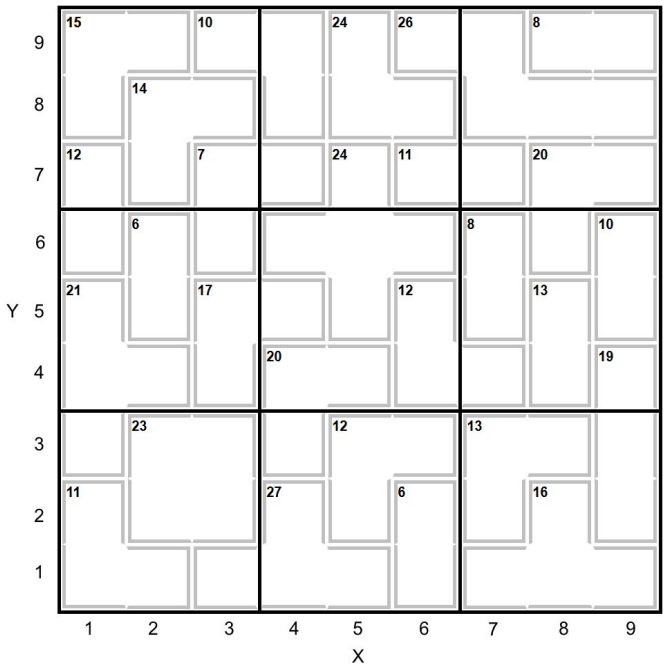


Figure 2: An unsolved Killer Sudoku

The differences between the regular Sudoku and this variant stem from the inclusion of a number of additional containers. These containers, known as Cages, each have a number associated with them that is equal to the sum of all Values contained within that Cage. The inclusion of this mathematical concept means that the numbers in the puzzle are no longer simply replaceable with symbols, as is the case with the original Sudoku.

Due to the rules associated with the Cages, the Killer Sudoku places additional constraints on the Possible Values in any one Cell. This leads to a puzzle where with any Cell the puzzler manages to determine the Value for, more information on the remaining unresolved Cells is gained. To counter this increased information gain and to promote the use of Cage rules, there are usually no initial Cell Values provided with a fresh puzzle. Providing no initial Cell Values with a regular Sudoku would result in a puzzle with a maximal number of possible solutions, whereas a unique solution is required for a well-formed puzzle.

2.3 Puzzle Definitions

In this section I will provide the definitions, that concern the puzzle itself, used throughout this thesis. The aspect of the puzzle being defined is labelled red in each figure.

Grid

The Grid is the complete field of the puzzle.

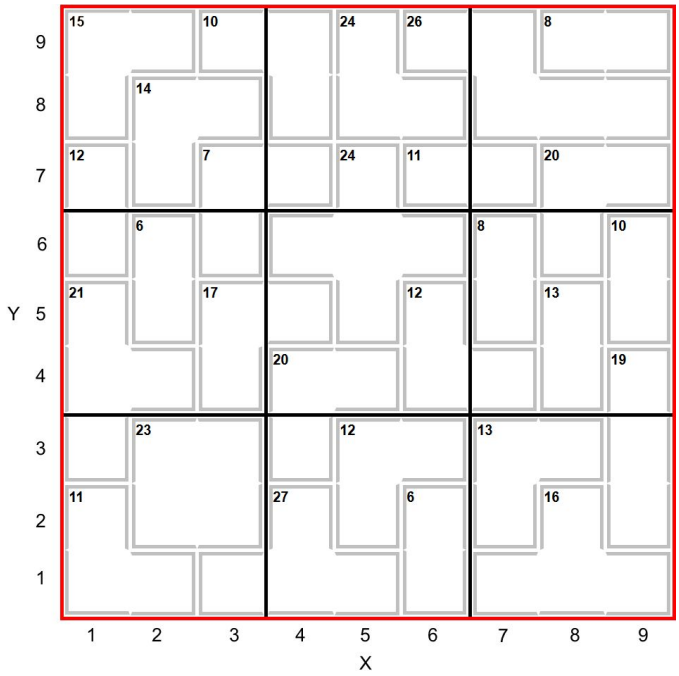


Figure 3: The Grid

Dimension

The Dimension directly determines the size of the Grid and thereby the total number of Cells, Rows, Columns and Blocks.

The typical Dimension for Sudoku and Killer Sudoku puzzles is 3, resulting in a 9x9 Cell Grid.

Cell

A Cell is the smallest element of a puzzle. The Grid contains a total of Dimension^4 Cells.

As such, a Grid of Dimension 3 contains a total of 81 Cells.

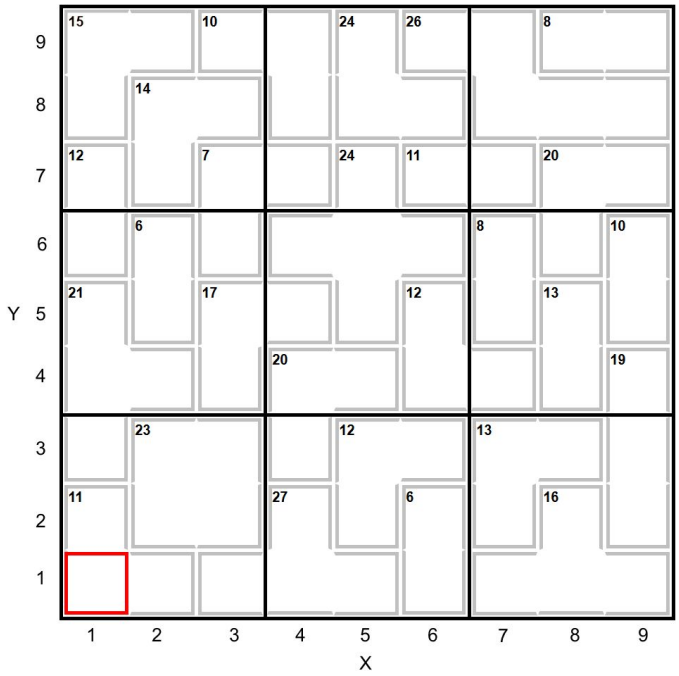


Figure 4: A Cell

Value

A Value is a single natural number between (inclusive) 1 and Dimension^2.

As such, a puzzle of Dimension 3 contains Cells with Values between (inclusive) 1 and 9.

Final Value

A Final Value is the Value a Cell holds in the puzzle's solution.

Possible Value

A Possible Value is a number associated with a Cell that has not yet been eliminated as either a possibility or determined to be the true Final Value. I have chosen for a finalized Cell to no longer contain any Possible Values. This may seem un-intuitive at first, since surely the Final Value must also be a possible Value, though it works very intuitively for the algorithm internally. With my chosen interpretation, simply taking the union of all Possible Values in a House requires no extra checks and is not "contaminated" with Final Values.

House

A House is any container of Cells with the exception of the Grid itself. Rows, Columns, Blocks, Cages and Diagonals are all Houses. The total number of Houses differs even between puzzles of the same Dimension as will become clear when defining Cages. It is worth noting that Cages are not always categorized as Houses, due to their size deviation, but for the purposes of my thesis, they are. It is useful to categorize them this way, due to both Houses and Cages being the Target of solving Rules. There is more on that in the next chapter. All Houses are subject to a unique Value constraint, meaning no Value may occur twice within their Cells.

Row

A Row is a horizontal and rectangular House, containing Dimension^2 numbers, as seen in Figure 7. This is also equal to the number of Rows present in the puzzle.

As such, a puzzle of Dimension 3 has 9 Rows.

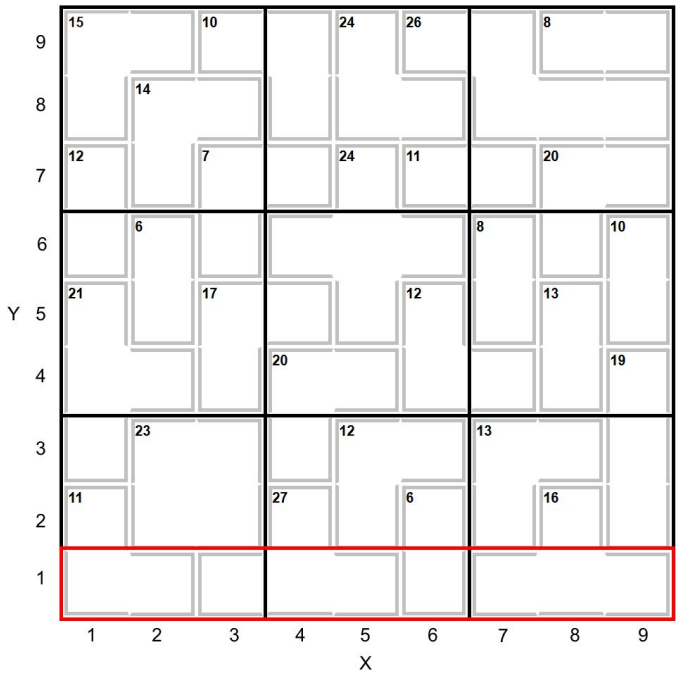


Figure 7: A Row

Column

A Column is a vertical and rectangular House, containing Dimension^2 numbers, as seen in Figure 8. This is also equal to the number of Columns present in the puzzle.

As such, a puzzle of Dimension 3 has 9 Columns.

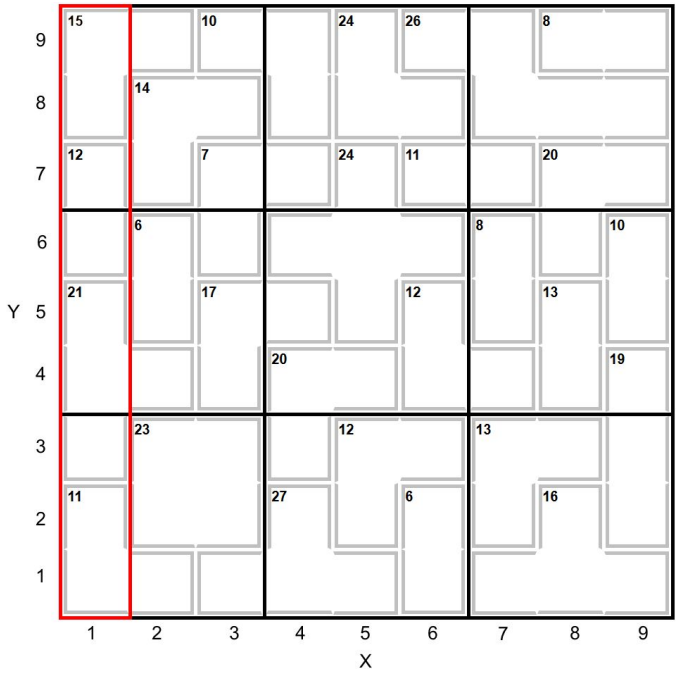


Figure 8: A Column

Block

A Block is a square House, containing a Dimension^2 number of Cells. The number of Cells in a Block also equals the number of Blocks in the puzzle. The term “Nonet” is sometimes used to describe Blocks, but since that word means “a group of nine people or things” it implies a size of 9. My algorithm is designed for puzzles of varying Dimensions, I have therefore decided upon this more generic name.

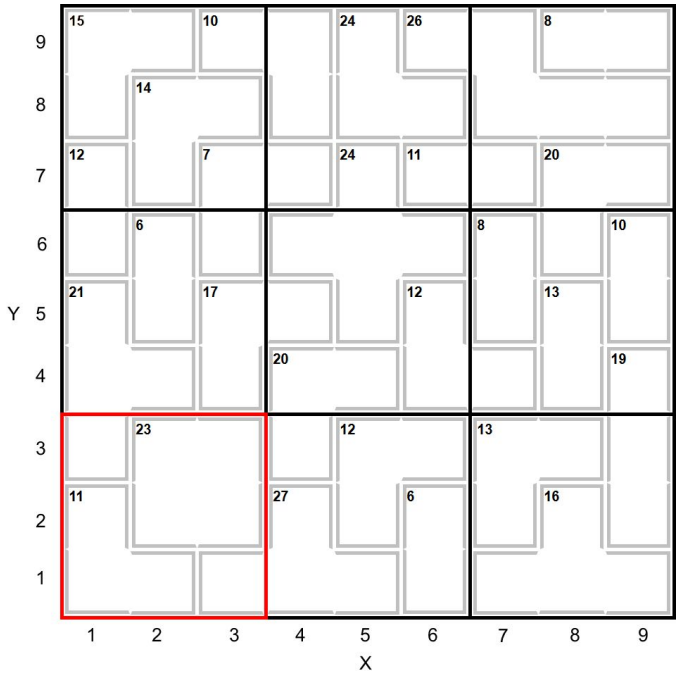


Figure 5: A Block

Diagonal

A Diagonal is a House spanning one corner of the Grid to the corner across from it on both axes. These Houses are only included on the Killer X variant of the Killer Sudoku and as such, are not standard, though the solving algorithm this thesis can optionally make use of this extra information. While the number of all other House types is completely based on the Dimension, there are only ever two Diagonals present, as illustrated in Figure 9.

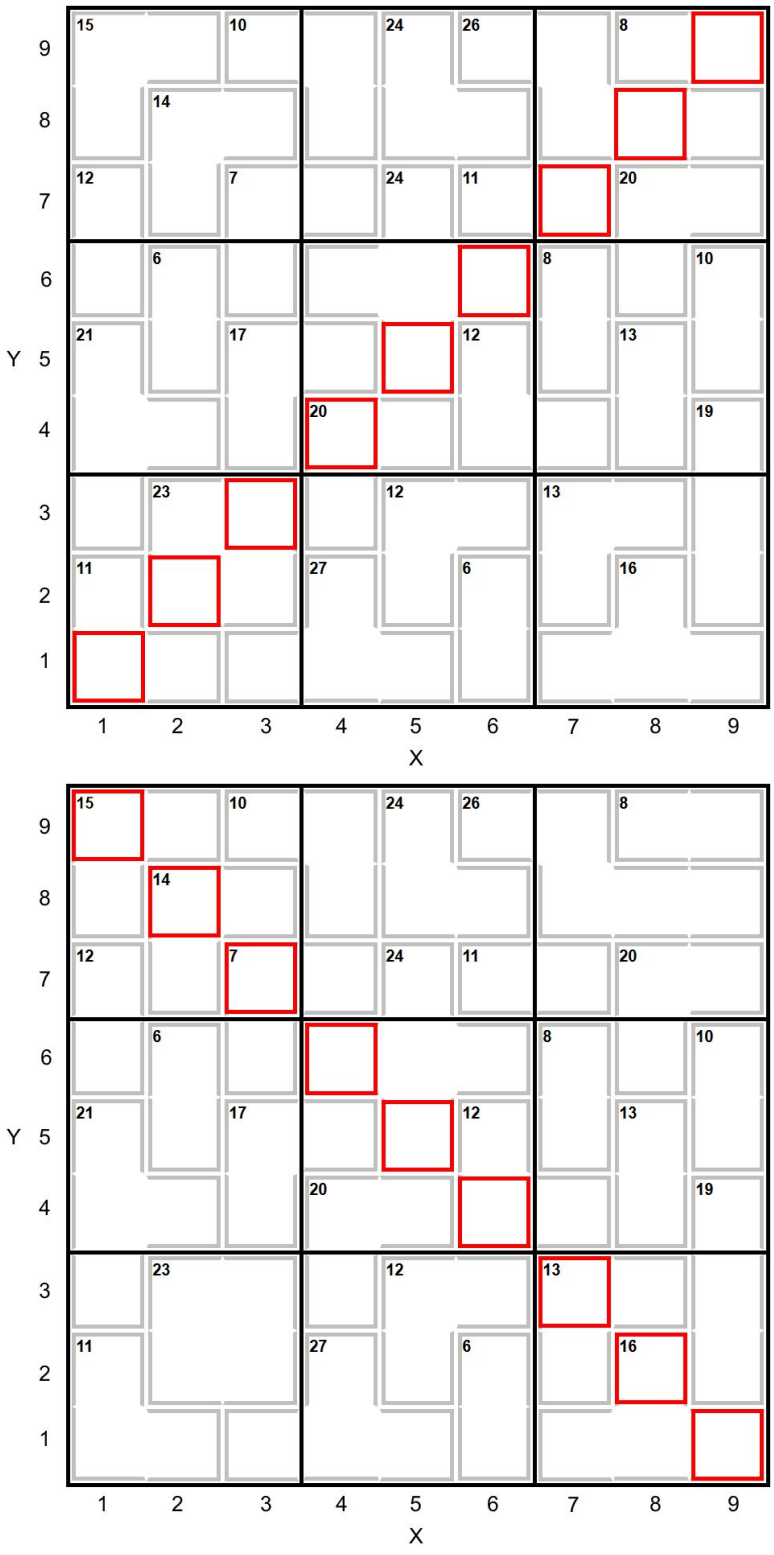


Figure 9: The first and second Diagonal respectively

Cage

A Cage is a House associated with a Goal as seen in Figure 5. Unlike the other Houses, the number of Cages and their sizes vary from puzzle to puzzle and are not determined solely by the Dimension. They are always contiguous however and a Cell can only belong to one Cage. Cages are what distinguish a Killer Sudoku from a regular Sudoku and they are the only tool initially available to determine the Possible Values of Cells.

There is a minimal number of Cages required to make a puzzle well-posed and solvable. For instance, if all Cages overlap exactly with the Blocks of the puzzle, there is no information to base the elimination of Possible Values on. Determining this minimal number of Cages is not within the scope of this thesis however. The number of Cages for any Killer Sudoku puzzle is only limited to the total number of Cells, but is usually much lower and in the 25-40 range.

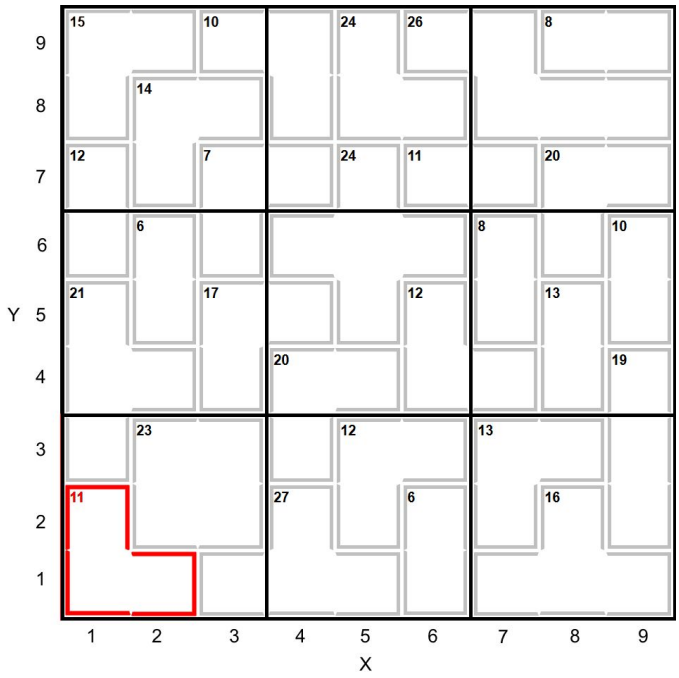


Figure 6: A Cage

Goal

A Goal is a value associated with a House, usually a Cage as seen in Figure 6. It represents the sum of the Final Values of all Cells within the House. Due to Cages being limited by size and bound to the unique values House constraint, a Goal cannot exceed:

$$\sum_{i=1}^{Dimension^2} x_i$$

As such, a puzzle of Dimension 3 cannot have Goals lower than 1 or exceeding 45.

3 Algorithm

3.1 Parsing

To compute the solution to a puzzle, it must first be represented within the system. In order to facilitate that transition from image to usable data, I store the puzzles as a text representation. An example, representing the same puzzle used to illustrate the various Houses in the previous section, is provided in Figure 10. Although he does not describe it, the format I used here is heavily influenced by the one used by J. Salam's for his bachelor thesis' (Salam, 2018). The first line contains additional data (for dynamic Dimension and Killer X compatibility).

```
3 27 False
1 1 1 2 2 1 11 +
1 3 1 4 1 5 2 4 21 +
1 6 1 7 12 +
1 8 1 9 2 9 15 +
2 2 2 3 3 2 3 3 23 +
2 5 2 6 6 +
2 7 2 8 3 8 14 +
3 1 4 1 4 2 5 1 27 +
3 4 3 5 4 5 17 +
3 6 3 7 4 7 7 +
3 9 4 8 4 9 10 +
4 3 4 4 5 4 20 +
4 6 5 5 5 6 5 7 6 6 24 +
5 2 5 3 6 3 12 +
5 8 5 9 6 8 24 +
6 1 6 2 6 +
6 4 6 5 7 4 12 +
6 7 7 7 11 +
6 9 7 8 7 9 8 8 9 8 26 +
7 1 8 1 8 2 9 1 16 +
7 2 7 3 8 3 13 +
7 5 7 6 8 +
8 4 8 5 13 +
8 6 8 7 9 7 20 +
8 9 9 9 8 +
9 2 9 3 9 4 19 +
9 5 9 6 10 +
```

Figure 10: A Killer Sudoku in parsable format

The first line of the file contains the base specifications of the puzzle:

- The first value represents the Dimension.
- The second value represents the total number of Cages.
- The third value is either True or False, declaring whether or not this is a Killer X variant Sudoku.

The second to the very last line all contain the information of one Cage each:

- Every (except the last) two values represent the X and Y coordinates for a Cell within the Cage.
- The second to last value on each of these lines represents the Goal associated with this Cage.
- The last value on each line represents the operator used on the Values of this Cage, which is '+' for a Killer Sudoku. The system at present cannot handle other operators, as included in a Calcudoku, because of the rework that would require on many rules. I made this choice to facilitate future extension.

Every other aspect of the puzzle, like House information, can be and is inferred from these values.

3.2 Verification

Unfortunately, transforming a Killer Sudoku from a page in a puzzle book or image from the internet into the parsable format described into the previous section is a laborious and error-prone task. After discovering several errors in transformed Killer Sudokus, I wrote a method to verify whether the resulting puzzle that is parsed is likely correct. Since a well posed Killer Sudoku puzzle only has a single solution, verifying that a puzzle is absolutely correct is a task about as complicated as actually solving it. Because of this, the verification method provides no guarantees, but will not be fooled by anything less than multiple errors cancelling each other out. These are cases like one Cage's Goal being 1 too high and another Cage's Goal being 1 too low. Fortunately, that is unlikely to occur.

The method performs the following checks:

1. Does the number of Cages in the file match the number stated on the first line?
2. Does the number of parsed Cells equal $Dimension^4$?
3. Is every Cell in the Grid initialized? (combined with check 2, rules out double Cells)
4. Does the sum of all the Cage Goals equal the required $\frac{(1+Dimension^2)}{2} * Dimension^4$? (this rules out missing Cages and Goal errors unless multiple mistakes balance each other)
5. Do none of the Cages contain more than $Dimension^2$ Cells? (larger Cages can not conform to the unique Values constraint)

3.3 Approach

Having started solving Killer Sudoku puzzles and keeping track of my reasoning with a notepad, I have formulated the steps I took to the best of my ability. As time went on I involved several people in my direct vicinity, like my supervisor, in sharing their methods for specific sub problems.

I find that I make a mental model of which Possible Values are still valid for each Cell and attempt to eliminate those possibilities until only one, which must be the true, Possible Value remains. In simple cases like single Cell Cages, my mind does seem to jump to the conclusion that the Goal associated with the Cage must be the sole Cell's Value. Deliberate process of elimination is the norm however. This is made especially apparent when difficulty increases and I must resort to writing out the Possible Values for each Cell rather than attempt to maintain this model exclusively in my own consciousness. The process of elimination is therefore the foundation of my model and algorithm.

People use a variety of steps, tricks, or Rules as I have decided to call them, to help them solve puzzles. Figuring out which exist, which are needed and when to apply them, has been the majority of my work on this thesis. Process of elimination is the foundation and Rules are the bread and butter to eliminate Possible Values.

I have not found a clearcut hierarchy to people their usage of Rules and I will prefer attempting to apply one over another depending on circumstance. With a basic Queue data structure to contain these Rules I would not be able to capture the dynamics of sometimes pursuing one thread of progress into different Rules and at other times executing the same Rule in as many places as possible. One way or another, there must be a choice of Rule depending on circumstance. It occurred to me that it might be interesting to model human behaviour through use of a Priority Queue data structure (Cormen, Leiserson & Stein, 2009, p.162). A Priority Queue functions much like the Queue it takes its name from, but requires a Priority value on Enqueue (inserting), and Dequeues (extracting) elements in order of Priority (high to low). Rules are intended to capture all human solving behaviour and since I have found no evidence of performing multiple Rules at once. Therefore I pose a suitably sophisticated function to determine the Priority value of any Rule at any point, and an expansive enough set of Rules, should be able to closely approximate this human serial solving behaviour. The challenge then is to design a sophisticated Priority function and a robust set of Rules.

The determination, design and implementation of the model and Rules was a time-consuming process and therefore a more intelligent way of dynamically determining Rule Priorities fell outside of the scope of this Bachelor thesis. The solver is presently capable of solving puzzles of low to moderate difficulty, while using predetermined Priorities per Rule, but fails to find a Final Value for all Cells on higher difficulties. This means that the logic behind the Rules is not yet strong enough and one or more critical Rules are still missing.

3.4 Solving Definitions

Successor

A Successor House is a new House that is constructed from a House of which one or more Cells have attained their Final Value. This Successor House no longer contains the finalized Cell(s), has Final Value(s) subtracted from its Goal and might no longer be contiguous. This is very useful for the application of certain Rules, as we rarely need to consider finalized Cells.

For example, consider the Possible Values for a Cage with a Goal of 15, containing three Cells, of which one has a Final Value of 5. We are no longer interested in this 5, as it will have already been eliminated as a Possible Value for the remaining two Cells. Instead, we would much rather consider a Cage of two Cells, with a Goal of (15 - 5 =) 10. A Successor House provides this functionality seen in figure 10:

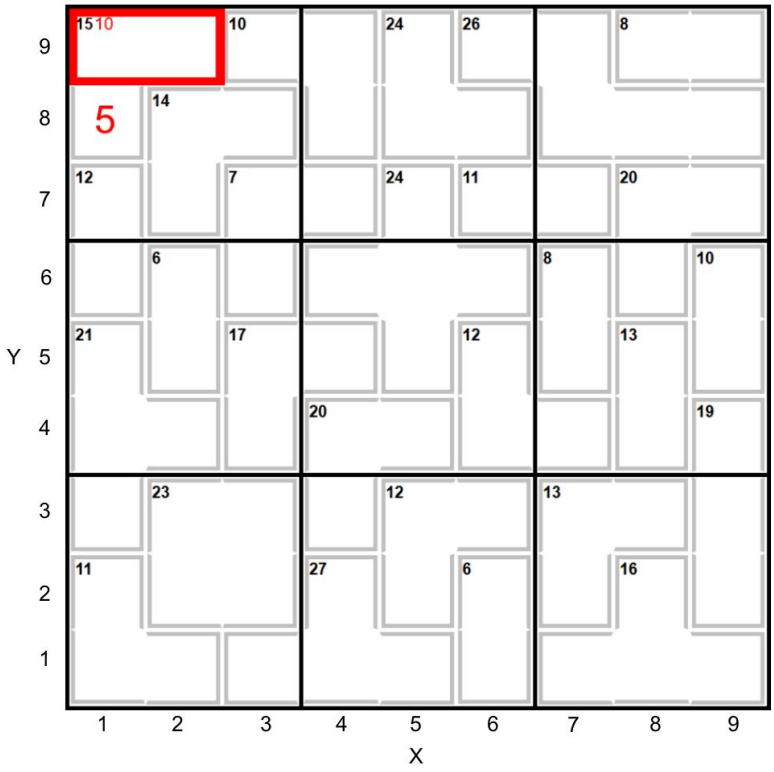


Figure 10: A Successor Cage

Rule

A Rule is a representation of a solving operation or trick that people use to help them solve Killer Sudoku puzzles. A Rule is associated with a Target and a Priority.

Target

A Target is a House in the puzzle that a Rule is set to be applied to.

Priority

A Priority is a value that determines where in the order of execution a Rule falls. Higher Priority Rules are executed before lower Priority Rules. This can be used to model human preference and is currently used to give more sophisticated (though weaker) Rules precedence over more brute-force rules.

3.5 Rules

This section discusses the rules that I discovered and analyzed, though not always fully implemented for reasons described for each individual Rule below. They can be categorized as Mathematical, when they make use of the Cage Goal property, or as exclusively Logical in all other cases. Of course, Mathematical Rules do also use Logic. In my algorithm, all Rules keep track of the Cells they had an effect on and return them for re-evaluation and potential re-Enqueuement.

All Rules are currently associated with a Priority that is predetermined for their type. This Priority is used to determine in which order the Rules are executed by the algorithm. The goal of this approach is to allow the intelligent choosing of Rules: making sure that the easier and more intuitive Rules are executed first. These Rules are generally weaker in that they eliminate fewer Possible Values per execution, but do so by smarter means than simply trying all permutations of Possible Values within a House and retaining only the valid ones. As a side bonus, these Priorities should also result in faster execution time when well balanced. To be clear, I consider a Rule stronger when it is capable of eliminating more Possible Values than another Rule when used in comparable circumstances. This strength doesn't necessarily make them better choices.

Math: RemoveHighLow

Priority: 1

Summary:
Possible Values in a Cage's Cells that are too high or too low to permit summing up to the Cage's Goal should be eliminated.

The purpose of this Rule is to remove all Possible Values from a Cage that are too high or too low to be viable when considering the Goal. Which Possible Values are too high or too low is calculated separately.

Example:
The Cells of a 2 Cell Cage with a Goal of 6 should not hold any Possible Values higher than 5 as the summation of those Values would exceed the Goal. Similarly the Cells of a 2 Cell Cage with a Goal of 17 should not hold any Possible Values lower than 8 as the summation of those Values would never reach the Goal.

Implementation:

- Create a Successor Cage of this Cage, as we do not need the finalized Cells.
- Create a set that is the Union of the Possible Values for all Cells in this Successor Cage.
- Too high:
    1. Fill all but one Cell in this Cage with the top lowest Possible Values in this union set.
    2. Sum up the Values of these Cells.
    3. Subtract this sum from the Cage's Goal. The resulting Value is the largest Value this Cage can contain without exceeding the Goal: max.

- Too low:
    1. Fill all but one Cell in this Cage with the highest of these gathered Possible Values.
    2. Sum up the Values of these Cells.
    3. Subtract this sum from the Cage's Goal. The resulting Value is the lowest Value this Cage can still contain while reaching the Goal: min.

- Go over the Possible Values and remove any that are higher than our max or lower than our min.

Context:

Due to the lack of any other initial information in a fresh Killer Sudoku, starting with consideration of the Cage Goals is the only option. RemoveHighLow is the first Rule that was discovered and added. Initially it was intended to be Queued only upon starting the solving process and not to be reused. It therefore assumed the full set of Possible Values based directly on the Dimension of the puzzle, rather than considering the Possible Values in its composing Cells. It quickly became clear however that this Rule would come in handy time and again as more Final Values for Cells are determined and the Cages' size could be adapted to remove them. This allows the Final Value to be subtracted from the Cage's Goal, effectively resulting in a new Successor Cage that this Rule could be applied to once again.

It has proven particularly effective on large Cages with a low Goal and small Cages with a high Goal, due to their relatively low number of acceptable Possible Values. Due to Cages typical size of 2-5 Cells, this is an intelligent, and relatively cheap in terms of processor time, Rule.

Logic: RemoveDuplicatePossibilities

Priority: 2

Summary:

If the Final Value for a Cell is determined, it should be eliminated as Possible Value for all Cells in all Houses this Cell shares to conform to the unique Value constraint.

Example:
With 6 as Final Value in the central Cell of the bottom Row, 6 should be removed as Possible Value from all Cells in the original Cells' Row, Column, Block and Cage.

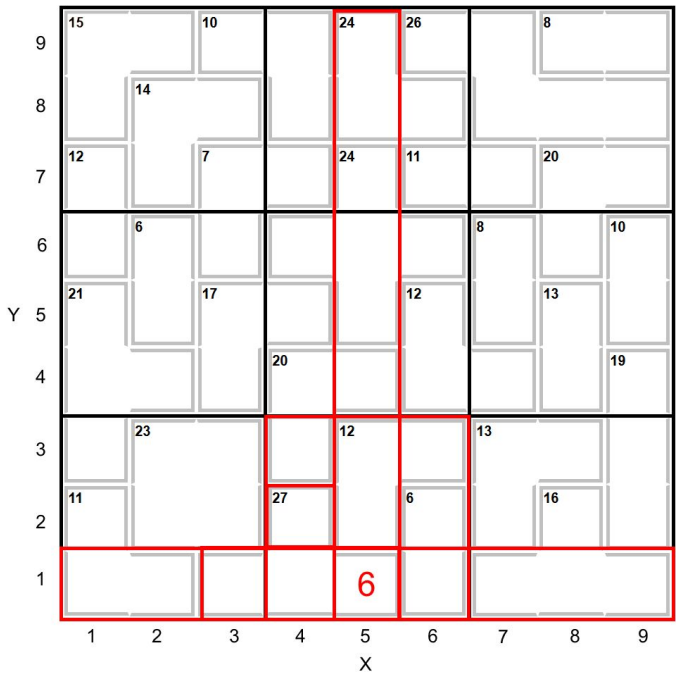


Figure 11: Applying the RemoveDuplicatePossibilities Rule

Implementation:

- Go through every Cell in every House that the original Cell is located in.
- Remove the original Cell's Final Value as Possible Value in all of these.

Context:
This operation initially was not a Rule, but simply part of any Cell's operation when it realizes its Final Value. The reasoning behind this was that it would be good practice to continue with work which you are almost certain will bring progress and I tend to jump on this while puzzling myself. For the same reason Cells immediately finalize themself into having a Final Value once they realize they only have only one Possible Value left. The combination of modelling these two operations as automatic results in problematic behaviour however. Due to how fast a Cell's Houses interconnect them to every other Cell in the puzzle, large chains of finalizations can occur to the point where the solving algorithm is essentially just performing operations depth first and control is handed away from the Rules system for an extended period of time. I consider that to be undesirable and have further limited instances of seemingly sensible automatic operations.

Logic: LastPossibleValue

Priority: -

Summary:
If a Cell has only one Possible Value left, that must be its Final Value.

Example:
There can be no question that a Cell whose Possible Values are {1} must receive the Final Value of 1.

Implementation:

- Set this Cell's Possible Value as its Final Value.
- Eliminate the last Possible Value.

Context:
Within the algorithm, this operation is not technically a Rule that can be queued, but a check ran by a Cell whenever it removes one of its Possible Values. It is included here for the sake of clarity. Unlike with RemoveDuplicatePossibilities, this works well and reduces the overhead of handing back Cells for re-evaluation slightly, without breaking what I consider to be intuitive human behaviour.

Logic: OnlyPossibilityLeftInHouse

Priority: 1

Summary:
If a certain Possible Value is only present in one Cell in a House, it must be the Final Value for that Cell.

Example:
Consider a Row in a puzzle of Dimension 3 that contains Cells with these Possible Values {1,2}, {2,3}, {2,3}, {4,5}, {3,5}, {4,6,8}, {6,9}, {7,8}, {2,9}. The Possible Value 1 only occurs in the first Cell. Therefore it must be the Final Value and {2} can be removed from the Cell.

Implementation:

- Create a Successor of this House as we do not need the finalized Cells.
- Create a set composed of the union of all the Possible Values in its Cells.
- If this House is a Cage, break off execution if the number of Possible Values is not equal to the number of Cells.
- For each Possible Value:
    1. Go through all the House's Cells' Possible Values and keep track of the last instance and how many times you found it.
    2. If there was only one instance, the last instance must be the correct one, so remove all other Possible Values from the Cell.

Context:
This Rule was initially overlooked and illustrated to me how hard it can be to register processes that seem to have become automatic to the point of being almost subconscious.

It is important to note that this Rule is not always correct for Cages unless the following is considered: It can occur that a Possible Value is only present in one Cell of a Cage, but this Possible Value not being the Final Value for that Cell. This is possible because this Value may not be present in the Cage at all as a Final Value. This can occur in any Cage that is smaller than Dimension in size and for that reason the check in the second step of the implementation was added.

Math: GoalSum *(unimplemented)*

Priority: 1

Summary:
Possible Values that can not sum up to the Goal, considering the other Possible Values, should be removed. This is essentially a stronger version of RemoveHighLow in that it removes the same Possible Values as that Rule, but potentially additional ones as well.

Example:
Consider a Cage with a Goal of 16 and 2 Cells with Possible Values {7,9}, {8,9}. The RemoveHighLow Rule can not help us here as the union of these Possible Values is {7,8,9} and none are too high or too low to reach the Goal by themselves. A human would quickly discover that due to Values having to be unique in a House, 8 is not a valid Possible Value here. GoalSum would generate the following possibilities:

{7}, {7}	{8}, {7}	{9}, {7}
{7}, {8}	{8}, {8}	{9}, {8}
{7}, {9}	{8}, {9}	{9}, {9}

And then proceed to disregard all, but the {7}, {9} and {9}, {7} for violating uniqueness and Goal matching constraints. All Possible Values that are not a member of the new Set of Possible Values {7,9} are then removed from the Possible Values in each Cell.

Implementation:

- Create a Successor of this House as we do not need the finalized Cells.
- Create a set composed of the Possible Values of all Cells it contains.
- Generate all combinations (of a size equal to the Cage) and see if their sum matches the Goal.
- If the sum matches the Goal, all Possible Values in this list are valid for this Cage and any Possible Value not present in any of these lists should be removed from all Cells in the Cage.

Context:
This Rule was designed after the realization dawned that RemoveHighLow would not be powerful enough to provide the logic I needed, as the example below shows best. Possible Values too high or too low are removed properly by RemoveHighLow. Values that are in between these extremes, but can not sum up to the Goal because there is no possible complement to them, are retained however. Humans swiftly identify such problems, especially in small Cages. As such, these Possible Values should be eliminated and this Rule was designed to accomplish that.

While programming it out, I realized that simply gathering the Possible Values and going through all combinations (of a size equal to the Cage) of Possible Values, to see which would sum up to the Goal, was still too limited a logic. An experienced puzzler is well capable of keeping track of which individual Cells are involved when considering Possible Values, but this Rule stops considering individual Cell possibilities. A good Rule should be capable of this human feat as well. Not only that, but with a small tweak, to not worry about overshooting the Goal, it would work over all Houses and not just Cages.

As such this particular Rule was not implemented and its stronger successor RemoveImpossibles was born in its stead.

Logic: NCageN

Priority: 1

Summary:
If the number of Possible Values left in a (Successor) Cage equals the number of (non finalized) Cells, those Possible Values should be removed from any Houses that encompass such a Cage.

This rule is used to eliminate Possible Values from Successor Cages that are contained within a Block, Row, Column or Diagonal.

Example:
Let us say the Final Value for Cell (1,8) is 5 and the Possible Values left in Cell (1,9) and Cell (2,9) are both {2,8}. We know 2 and 8 can only be located in either of these Cells and this (red) Successor Cage is located entirely in the top Row and top left Block (both purple). Thus we can be certain 2 and 8 can't be the Final Value for any other Cell in these Houses and they should be removed from the Possible Values.

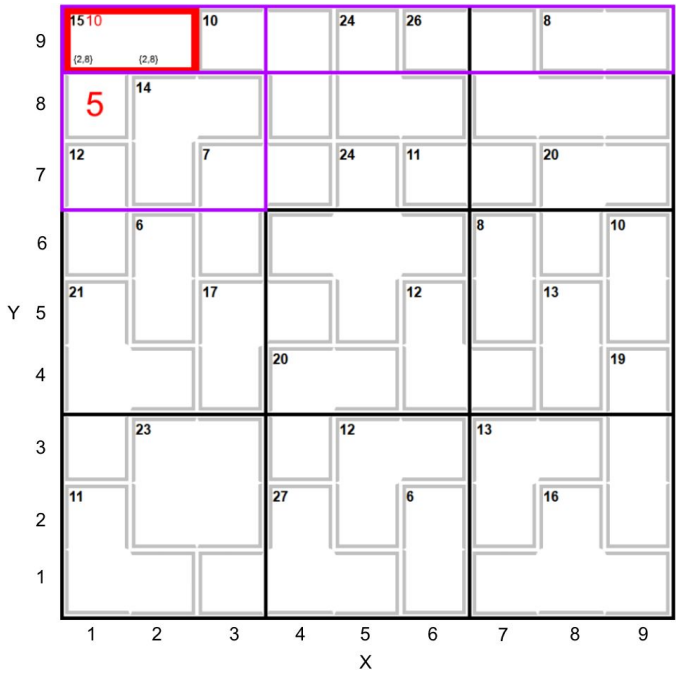


Figure 12: Applying the NCageN Rule

Implementation:

- Create a Successor of this Cage as we do not need the finalized Cells.
- Create a set composed of the Possible Values of all Cells it contains.
- If this Successor Cage does not contain any Cells or the number of Cells doesn't match the number of Possible Values, break off execution.
- For all House types except Cages (Row, Column, Block, Diagonal):
    1. Check if all Cells in the Successor Cage belong to the same House.
    2. If they do, remove the Possible Values of the Successor Cage from all Cells, except those in the Successor Cage, in the current House.

Context:

This Rule is implemented for Cages and performs well as such. The basic principle is valid for any group of Cells that have the same set of Possible Values, the same number of Cells and are encompassed by a House. In order to determine which sets of Cells meet those conditions, every House will have to be checked for just about every possible combination of its Cells. A Rule that performs the same operation as this one does, over all those sets of Cells would be strong, but also extremely expensive processor wise to execute. NCageN is a compromise of sorts that helps the algorithm avoid getting stuck without progress, but avoid the combinatorial explosion. Successor Cages are the House of choice, because their size is typically limited to a point where an equal number of Cells and Possible Values is more likely. This Rule could be expanded to a "NHouseN" Rule to perform the same operation on all House types, though I expect much less success there, due to their greater size.

Math: InnieOrOutie *(unimplemented)*

Priority: 1

Summary:

If a Block consists of Cages that have only a single Cell protruding into (innie), or out of (outie), the Block, the Value of the Cell can be inferred.

Example:

Consider the 7th Block, located in the bottom right of figure 13. We see a single Cell protruding from the Cages of the Block, into the Block above it. We know that the Goal of this Block is 45 as the Goal of a Block is: $\sum_{i=1}^{Dimension^2} x_i$

By taking the sum of all the Block's composing Cages and subtracting the Block's Goal we can determine the Value of the portuding Cell (9,4): $(16 + 13 + 19) - 45 = 3$. This same principle holds true for any House that contains Cages.

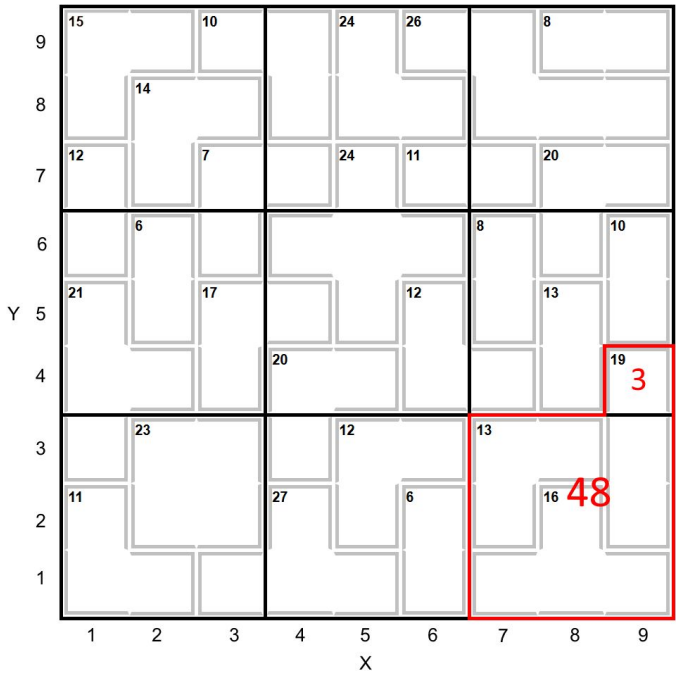


Figure 13: Applying the InnieOrOutie Rule on an "outie"

The Outie works through a similar principle. By taking the sum of all Goals of the Cages involved and subtracting the Goal of the House, we learn the amount of Value outside the House. Subtracting that number from the outie Cage Goal yields the Value of the Cell that lies inside in the original House.

Context:
This Rule was discovered relatively late in the project and remains unimplemented. I know of at least one current unsolvable puzzle: Medium2 (in the accompanying project files) that would be able to progress further were this Rule to be implemented, proving that this Rule provides logic not yet present in the other Rules.

Math/Logic: RemoveImpossibles

Priority: 0

Summary:
There are many reasons for which a Possible Value cannot be the Final Value in the solution. All Rules are designed to represent such a reason and are executed to see if this reason is applicable. RemoveImpossibles looks past these usually higher level constructs and simply looks if any valuation of the Cell leads to any problems with the current degree of knowledge. It does so by generating all possible combinations, considering the Possible Values of the individual Cell, and maintaining only those Possible Values in each Cell that lead to a consistent House.

Example:
Consider a Cage with a Goal of 7 and 3 Cells with Possible Values {1,4}, {1,2}, {1,2}. The RemoveHighLow Rule can not help us here as the union of these Possible Values is {1,2,4} and none are too high or too low to reach the Goal. Better yet, all are required. Even so, a human will quickly realize that the first Cell should not contain 1 as a Possible Value, as the rest of the Cells would never be able to sum up to the Goal of 7 that way. Of all the possible valuations only {4},{1},{2} and {4},{2},{1} will lead to the Goal, generating to these new Possible Values {4}, {1,2}, {1,2} where 1 has been successfully eliminated from the first Cell.

Implementation:
-Create a Successor of this House as we do not need the finalized Cells.
-Recursively construct all valuations for these Cells that are consistent with the uniqueness of Values constraints and sum up exactly to the Goal. This is achieved by a recursive function that moves through all the Cells and branches off once for each Possible Value that is consistent with the picked Possible Values so far. Branches that can find no more consistent Possible Values, go over the Goal, or do not reach the Goal in the end are pruned.
-Create a new set of Possible Values for each of the Cells that is the union of only those Possible Values for that Cell in consistent valuations.
-Replace the Possible Values in each individual Cell of the original House with their new union counterparts.

Context:
This Rule was designed as a stronger successor to GoalSum. It is by far the most powerful and computationally expensive Rule I have created. Unfortunately, its approach is somewhat akin to brute-force in that it simply generates and checks a large amount of possibilities. The result can be a combinatorial explosion and as such, RemoveImpossibles has been given a very low Priority in the solving process. It is executed only when no other Rules still yield improvement. Despite its flaws I am pleased to have found such a strong case to illustrate the utility of a Priority Queue data structure for this problem. This Rule is best compared to a human puzzle solver finding herself unable to make progress with more comprehensive techniques and deciding to take out pencil, eraser and some elbow grease.

Math: RuleOf45 *(unimplemented)*

Priority: 0

Summary:
The InnieOrOuti does not only apply to Cages within single Blocks. The principle used can be extended to many groups of Houses.

Example:
Consider the puzzle in figure 14. The top two Rows only have a single Cell extending from their combined Cages. Since we know the Values in both Rows sum up to 90 and all their Cages sum up to 97, the Value of Cell (2,7) must be $97 - 90 = 7$.

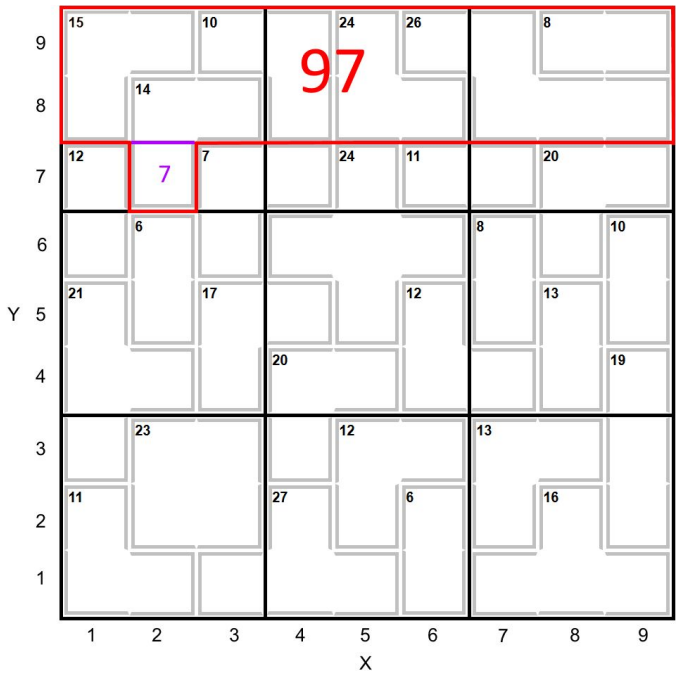


Figure 14: Applying the RuleOf45 Rule

Context:

This Rule is a stronger version of the InnieOrOuttie Rule as it applies to any group of Cages held in any group of other Houses (Rows, Columns, Diagonals, Blocks). Figure 13 and 14 apply Rules to single Cells protruding into or out Houses, but it can also be used to learn what the combined value of multiple protruding Cells is. These Cells could be formed into Successor Cages and have other Rules like RemoveHighLow performed on them. There is a huge amount of information to be extracted from the puzzle this way and the challenge is finding operations that make good use of this while avoiding the worst of the combinatorial explosion that is the result of checking any combination of Houses to innies or outies.

3.6 Control flow

The program starts out by parsing, validating and loading the requested puzzle. Assuming the provided information is verified as correct, the solving process itself begins, with the code displayed in Figure 15.

```
PriorityQueue<Rule> rulesQueue = new PriorityQueue<Rule>();
HashSet<Cell> improvedCells = new HashSet<Cell>();
HashSet<Cage> improvedCages = new HashSet<Cage>();
HashSet<House> improvedHouses = new HashSet<House>();

foreach (Cage cage in puzzle.cages){
    rulesQueue.Enqueue(new RemoveHighLow(cage, 1));
}

while (rulesQueue.Count() != 0){
    improvedCages.Clear();
    improvedHouses.Clear();
    improvedCells = rulesQueue.Dequeue().Execute();

    foreach (Cell cell in improvedCells)    {
        rulesQueue.Enqueue(new  RemoveDuplicatePossibilities(cell, 2));

        foreach (House house in cell.Houses) {
               if (improvedHouses.Add(house)) {
                      if (house is Cage) {
                             if (improvedCages.Add(cell.Cage)) {
                                    rulesQueue.Enqueue(new RemoveHighLow(cell.Cage, 1));
                                    rulesQueue.Enqueue(new NCageN(cell.Cage, 1));
                             }
                      }

                      rulesQueue.Enqueue(new OnlyPossibilityLeftInHouse(house, 1));
                      rulesQueue.Enqueue(new RemoveImpossibles(house, 0));
               }
        }
    }
}
```

Figure 15: The solver's main loop simplified

The solving process runs within a loop that only ends when the Priority Queue contains no further Rules to execute. The initial Rules Enqueued into the Priority Queue are one instance of RemoveHighLow for every Cage in the puzzle. As stated earlier, Cage information is the only information available at the outset of the solving process and RemoveHighLow is a relatively cheap and effective Rule to execute. I have not yet encountered a puzzle where the initial RemoveHighLow Rules are insufficient to get the process started, though I believe such a puzzles may exist. If this is indeed the case stronger Rules, like RemoveImpossibles, may have to be Enqueued first.

An executed Rule returns all the Cells for which it eliminated a Possible Value. All Houses that these Cells belong to are then used as Target for new Rules that are promptly Enqueued. Due to their designated higher Priority, more intuitive and cheap rules always take precedence when Dequeuing. Once their strength proves insufficient, stronger Rules are executed until they result in an improvement. The Houses of these improved Cells are then Enqueued with the higher Priority, cheaper Rules, so that those are up again.

There is sadly a fair degree of inefficiency in the algorithm at this point: After a Rule is executed and its improved Cells are evaluated, every House connected to these Cells is Enqueued with a Rule. Overlap between these Houses is not only possible, but partially guaranteed due to the improved Cells sharing at least the House that was the Target for the Rule that returned them. That particular overlap is accounted for by preparing a HashSet (Cormen, Leiserson, Rivest & Stein, 2008, p.262) of the improved Cell's Houses (improvedHouses in the Figure 15) before creating the New Rules. The HashSet identifies Houses by their memory location by default and ensures there will be no doubles. The resulting set is unique to each Dequeue however and the next Dequeue may possibly Enqueue the same Rules with the same Targets once more. This does not affect the outcome in any way, but does bloat the contents of the PriorityQueue and ultimately the number of non-improving Rule executions. I will elaborate more on Hashing and reducing overlaps in the Future Research section.

The Priority Queue will always empty eventually, due to Rule executions being finite and new Rules only being Enqueued if progress is made. Which is to say if at least one of the (finite number of) Possible Values is eliminated.

An empty Priority Queue guarantees that there are no improvements left that the algorithm can make. This is only the case when the puzzle is solved or the Rules at its disposal are not powerful enough to solve it fully. In either case, the loop terminates and the puzzle's (partial) solution is displayed as shown in Figure 16 below. Figure 17 shows a solved puzzle.

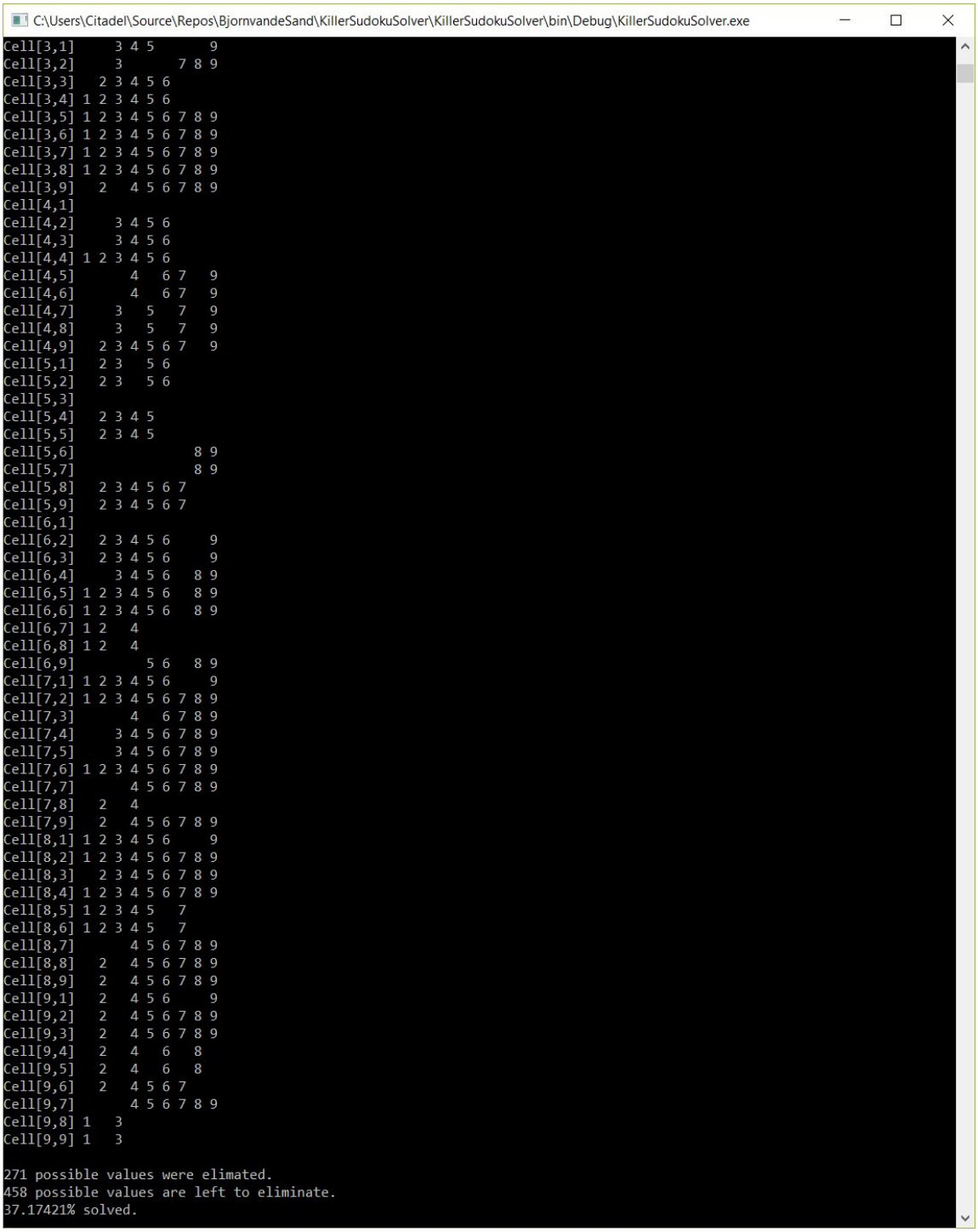


Figure 16: Program output for an unsolved puzzle

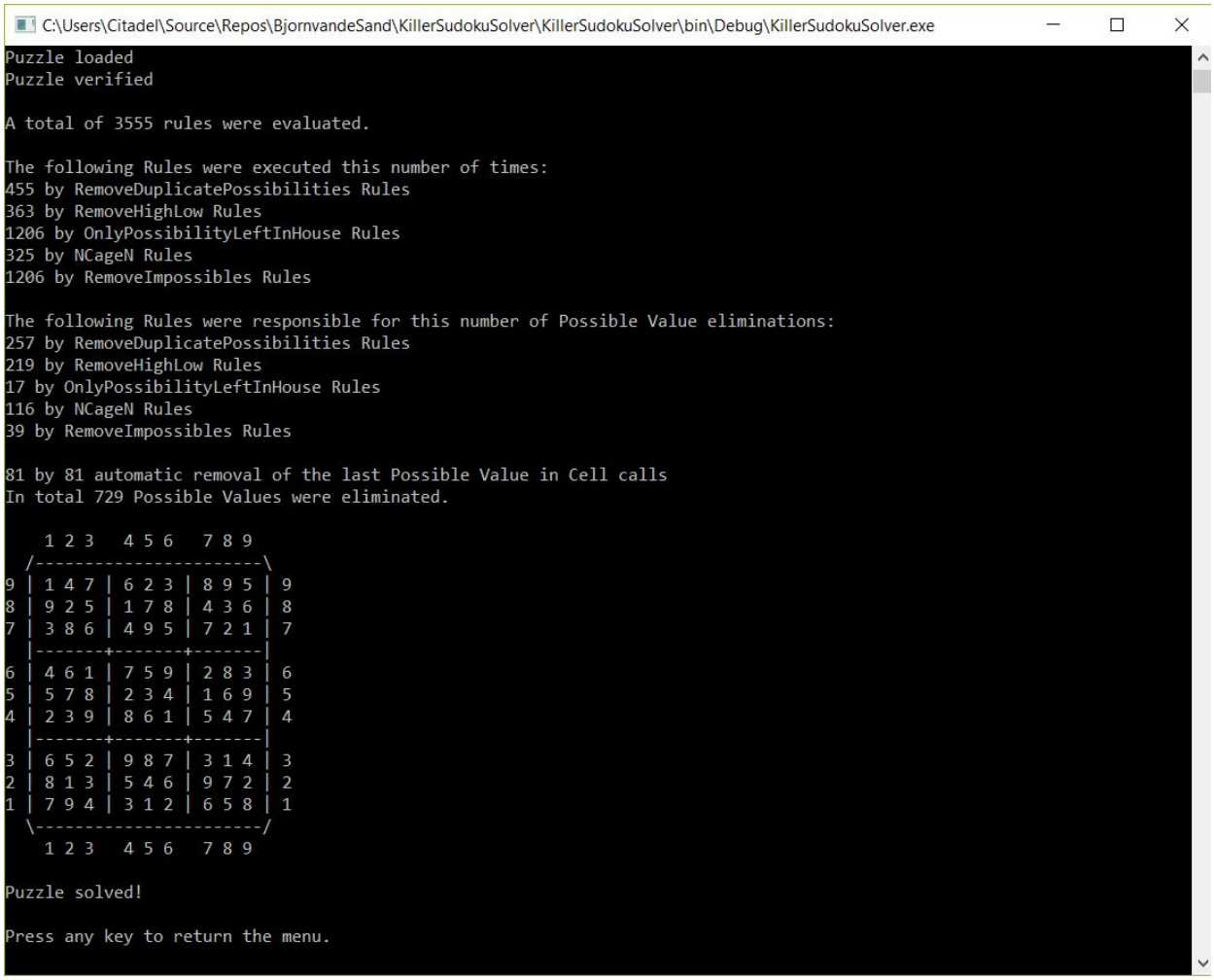


Figure 17: Program output for a solved puzzle

4 Results

4.1 Sources

In order to test my program I have had it attempt to solve Killer Sudoku puzzles of varying difficulty from both a puzzle book and a website for enthusiasts (Min, 2013). In this chapter I analyze the results for the puzzle book. Website puzzles are not included as those are not clearly tagged with a difficulty.

4.2 Puzzle book

Figure 18 shows the results of the first 5 puzzles per level: Warmup, Easy, Medium, Tricky and Difficult.

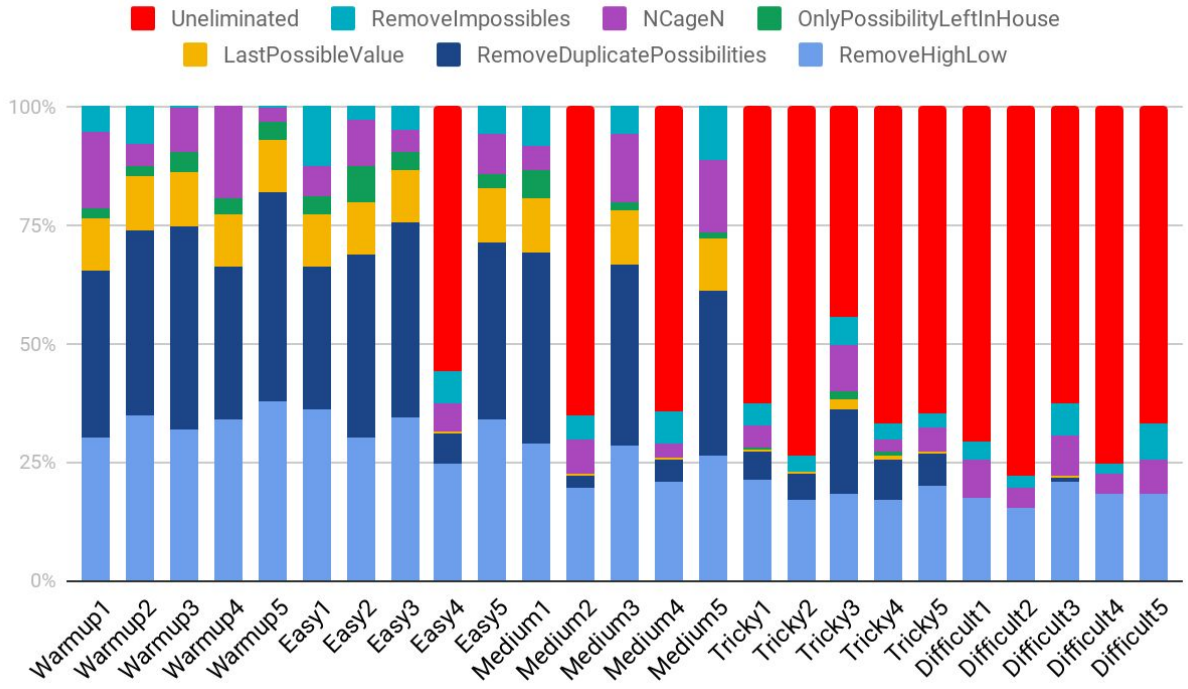


Figure 18: Share of Possible Value eliminations per Rule

Unfortunately the program is not yet capable of solving Tricky and Difficult puzzles and occasionally even fails to eliminate all Possible Values on easier puzzles. As described in the previous chapter, the RemoveHighLow Rule is queued first (though its Priority is not higher than most other Rules) and its effectiveness is more or less consistent throughout the difficulties, declining only slightly compared to the massive drop in effectiveness of most other Rules. LastPossibleValue is only executed if a Cell can be finalized, meaning it sits on a steady 81 ($Dimension^2$) eliminations if the puzzle is solvable by the algorithm and declines steadily if not. RemoveDuplicatePossibilities relies on LastPossibleValue for its utility and its effectiveness drops along with it. I had high hopes for RemoveDuplicatePossibilities due to its strength, but most of the Values it can eliminate are already eliminated by RemoveHighLow.

4.3 Difficulty

In order to better understand the difficulty rating of the puzzles in the previous section, I've compiled the number of Cages involved with each in Figure 19 below:

Puzzle	Warmup	Easy	Medium	Tricky	Difficult
1	38	41	35	32	31
2	42	37	34	35	31
3	43	36	34	32	28
4	42	39	35	31	31
5	33	42	36	32	29
Average	≈40	39	≈35	≈32	30

Figure 19: Number of Cages per puzzle

There is a strong correlation between the number of Cages and the difficulty rating, where fewer Cages implies a higher difficulty. This makes sense, due to the fact the number of potential valuations for any Cage increases greatly with its size. Not only that, but since there are more Cells present in the Cage, fewer Values can be dismissed out of hand. See Figure 20 below for a breakdown of the number of possible Valuations for a Dimension 3 puzzle like the ones tested:

Cage Size	**Possible valuations**	**Example**
1	9	{1}
2	72	{2,5}
3	504	{8,4,3}
4	3024	{4,9,6,2}
5	15120	{3,1,2,8,7}
6	60480	{5,1,9,5,2,4}
7	181440	{9,3,2,8,1,4,5}
8	362880	{7,2,1,4,5,6,3,9}
9	362880	{9,4,5,2,1,8,3,7,6}

Figure 20: Permutations for each Cage size

5 Conclusion

5.1 Conclusion

At this point the algorithm's most notable shortcoming in modelling human behaviour lies in that it is not capable of solving problems that many humans are capable of solving. Being able to perform the same input-output behaviours as humans by at least generating a solved puzzle was to be only the first step of my research. Although I believe I have found the Rules that are the missing links in logic required to accomplish consistent solving, especially in the RuleOf45. As such I'm pleased with the work I've done and what the algorithm can do. I must conclude that I sorely underestimated the amount of work required to solve even the most basic Killer Sudoku. I had expected to achieve that benchmark in the third week at the latest, but over half my time had been used when I finally reached it. I underestimated the complexity of the required Rules, but mostly I underestimated how slow the process of introspection and personal Rule learning was.

A successful part of the algorithm is that my idea of using a Priority Queue, ironically initially considered a low priority while building, quickly became a necessity while designing the Rules. This first came to pass when discovering errors sneaking into the logic through edge cases where RemoveDuplicatePossibilities wasn't executed directly after LastPossibleValue. This lead to a single Possible Value of 6 remaining in a Column where 6 was already the Final Value for a different Cell. The algorithm thus ended up producing two Final Values of 6 in one House. I could have solved this by enforcing a RemoveDuplicatePossibilities Execution immediately after OnlyPossibilityLeftInHouse. Doing so can cause a solving chain reaction all over the puzzle within a single Rule execution though. I was unwilling to give any Rule that kind of broad responsibility and end up falling into depth-first behaviour to prevent the original issue. By choosing to solve potential logic errors by enforcing the order of Rules, through Priorities, I lost the elegant property of Rules always being correct, regardless of the order of execution. Yet, what I gained in turn was something I consider a very human property: When a person settles on a method to tackle a problem, but for whatever reason does not take their actions in the order they intended, mistakes can be made. Similarly my Rules are all fully consistent at a glance, but there are a few rare cases of interaction where the order of execution is very important. This reminds me of a person who forgot what they were presently doing and carry on with information they wrote down, without considering the implications of their earlier discovery. In a way the model allows for representing and studying errors of this kind, should that be desired and the priorities tweaked accordingly.

I also found it noteworthy that when I started compiling the Rules, I started with low complexity ones like RemoveHighLow. I did this both to represent my own level of skill and to ease my way into the process of building my model. I had expected these low complexity Rules to become redundant as I learned stronger ones. My intention was therefore to rid myself of them when I could. As I realized the workload many stronger Rules involve, the thought occurred to me that, even when I myself knew stronger Rules, I didn't execute these until it became necessary. Using weaker Rules to solve easy problems first makes for a more relaxed and faster puzzling experience. I do not resort to a brute force approach where I simply write down all possible grids until I absolutely have to. Similarly the weakest Rules still hold an important place in my model and the stronger are only executed when progress stops. I believe this actually the most human part of the algorithm.

5.2 General improvement

For the purposes of conciseness, improved modelling of human behaviour and improved performance, the problem of duplicate Rule/Target combinations present in the Priority Queue should be tackled. I propose a system that, on Enqueuement, checks whether or not the Rule/Target already exists within the Queue and not Enqueueing them again if so. Due to there only being a finite number of Rule/Target combinations, a unique integer Hash for each could be calculated. This Hash function would have to use the Dimension of the puzzle as an argument as well, but would render checking whether or not a Rule/Target combination is already in the Priority Queue trivial. If the algorithm were enhanced with dynamic priorities, the HashSet of Rules should replace Enqueued Rules only if the new Priority is higher than the already Enqueued one. This will prevent delaying a Rule that was earlier determined to be important.

While on the topic of dynamic Priority assignment to Rules, I think a considerable amount of additional research is warranted into learning which variables play a role in this determination for humans. Currently Priorities are mostly based on the strength and workload a Rule brings to the table. This is consistent with the human inclination to avoid unnecessary work, but it is not very sophisticated. Factors such as the present part of the person the subject is looking at, the numerical order of Values and the size of a to be evaluated House may play a currently underrepresented role. Lastly, while the task of solving puzzles is a completely rational one, humans may have irrational personal preferences such as preferring to solve certain numbers first. I would also very much like to see the Rules I did not get to implement in action and learn if they are sufficiently strong to solve any well posed Killer Sudoku puzzle. I believe that the RuleOf45 in particular is the missing element required for breaking through on harder puzzles, though this is difficult to verify without implementing it.

On a more trivial note, my program's output of remaining Possible Values for unsolved puzzles is currently adequate, but could be improved by a Dimension number of lines for each Row, rather than 1. This would allow for the printing of all remaining Possible Values within the Grid itself, rather than in a long list as is the case now. This would increase readability, particularly for Grids of higher Dimension.

5.3 Future Research

An improving and all around interesting step would be to design a function that dynamically assigns a Priority to specific Rules or preferably even Rule/Target combinations. The Priority function would have to consider to all variables that humans take into consideration when deciding. This would be primarily about how promising any Rule is, regarding the elimination of a maximum number of Possible Values, given the current state of the puzzle. The human element of preference for certain Rules or numbers should be considered as well though. Mapping all the human variables for the Priority function would no doubt be a considerable undertaking. Alternatively, I would also be interested in a less complex version that simply seeks to optimize efficiency and reduce the number of Rule evaluations and runtime. Information pertaining to Successor House size and Cage Goals would be a good place to start due to their effect on the total number of possible valuations for the House being considered.

It seems to me that this problem, of determining what any Rule's Priority should be, is one for which machine learning would be a very suitable application. Especially if the deductions a machine learning algorithm could make could be analyzed and translated back into advice for human puzzlers. That way the model could be used not only to analyze, but also to improve human puzzle solving.

I would also be very interested in a larger study on how humans solve (Killer Sudoku) puzzles. I have no reason to believe different puzzlers would use a fundamentally different approach, but my own introspection and discussion with a handful of fellow puzzlers is insufficient data to be certain.

References

L. Aaronson. (2006). Sudoku Science. 43 Issue 2, 2006.

S. Russell & P. Norvig. (2010). Artificial Intelligence: A Modern Approach (Third Edition), Pearson.

D. Haynes & S. Corns. (2013). EA-EMA Optimization Applied to Killer Sudoku Puzzles. Procedia Computer Science, 20, 2013.

B. Felgenhauer & F. Jarvis. (2006). Mathematics of Sudoku I. Mathematical Spectrum, 39, 2006.

T. Cormen, C. Leiserson, R. Rivest & C. Stein. (2008). Introduction to Algorithms (Third Edition), MIT Press.

J. Salam. (2018). The superior search method. Bachelor Thesis Artificial Intelligence, Utrecht University.

P. Min. (2013). The Huge Killer Sudoku Book (Second Edition).

Links

Bjorn van de Sand's Killer Sudoku Solver: https://github.com/BjornvandeSand/KillerSudokuSolver

Patrick Min's Calcudoku website: https://www.calcudoku.org/

James McCaffrey's PriorityQueue class:
https://visualstudiomagazine.com/articles/2012/11/01/priority-queues-with-c.aspx