Final Year Project Report

**Full Unit – Final Report**

Solving Sudoku and Killer Sudoku Using AI

Manfred Sandy Alex Araujo

A report submitted in part fulfilment of the degree of

**BSc (Hons) in Computer Science**

**Supervisor:** Emma Lieu



Department of Computer Science

Royal Holloway, University of London

January 27, 2024

**Declaration**

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 9,878

Student Name: Manfred Sandy Alex Araujo

Date of Submission:

Signature:

Table of Contents

[Chapter 1: Introduction 5](#_Toc157093317)

[1.1 The Problem 5](#_Toc157093318)

[1.2 Aims and Goals 5](#_Toc157093319)

[1.3 Literary Survey 6](#_Toc157093320)

[Chapter 2: Constraint solvers 8](#_Toc157093321)

[2.1 Introduction 8](#_Toc157093322)

[2.2 Backtracking and Recursion 8](#_Toc157093323)

[2.3 Arc Consistency 8](#_Toc157093324)

[2.4 Back jumping 9](#_Toc157093325)

[2.5 Dynamic Variable/Value ordering 9](#_Toc157093326)

[2.6 Forward checking 10](#_Toc157093327)

[Chapter 3: Recursion and Backtracking 11](#_Toc157093328)

[3.1 Introduction 11](#_Toc157093329)

[3.2 First Sudoku Solver 11](#_Toc157093330)

[3.3 Improving the Solver 13](#_Toc157093331)

[3.4 Comparing the Sudoku solvers 14](#_Toc157093332)

[3.5 Killer Sudoku 15](#_Toc157093333)

[3.6 Comparing the Killer Sudoku solvers 16](#_Toc157093334)

[3.7 Conclusion 17](#_Toc157093335)

[Chapter 4: Data Structures in Solvers 18](#_Toc157093336)

[4.1 Priority Queue 18](#_Toc157093337)

[4.2 Dictionaries and Sets 20](#_Toc157093338)

[Chapter 5: Machine Vision 21](#_Toc157093339)

[5.1 Grid extraction 21](#_Toc157093340)

[5.2 Straightening the Image 23](#_Toc157093341)

[5.3 Cell Extraction and Processing 24](#_Toc157093342)

[5.4 Machine Learning and Classification 25](#_Toc157093343)

[5.5 Killer Sudoku Machine Vision 25](#_Toc157093344)

[5.6 Conclusion 27](#_Toc157093345)

[Chapter 6: Puzzle Generation 28](#_Toc157093346)

[6.1 Sudoku Generation 28](#_Toc157093347)

[6.2 Killer Sudoku Generation 29](#_Toc157093348)

[Chapter 7: Solving Technique 32](#_Toc157093349)

[7.1 Obvious Singles/Pair/Triples 32](#_Toc157093350)

[7.2 Hidden Single/Pair/Triple 32](#_Toc157093351)

[7.3 Pointing Cells 32](#_Toc157093352)

[7.4 Rule of K 32](#_Toc157093353)

[Chapter 8: Software Engineering 33](#_Toc157093354)

[8.1 Testing and Documentation 33](#_Toc157093355)

[8.2 Methodologies and Design Patterns 33](#_Toc157093356)

[Chapter 9: Professional Issues 35](#_Toc157093357)

[Chapter 10: Conclusion 36](#_Toc157093358)

[10.1 Evaluation 36](#_Toc157093359)

[10.2 Planning and timescale 36](#_Toc157093360)

[Chapter 11: Bibliography 38](#_Toc157093361)

[Chapter 12: Appendix 40](#_Toc157093362)

[12.1 Diary 40](#_Toc157093363)

[12.2 User manual 42](#_Toc157093364)

[12.3 Video Link 42](#_Toc157093365)

[12.4 UML diagrams links 43](#_Toc157093366)

[12.5 Code snippets 43](#_Toc157093367)

# Introduction

## The Problem

Sudoku is a popular logic-based game that involves a 9 by 9 grid with some given numbers and the aim being to fill all the cells in the grid with the numbers 1 to 9. The rules of the game involve each row, column and box must use the numbers 1 to 9 exactly once. Killer Sudoku is a similar game to Sudoku but extends the game by adding "cages" around a group of cells that must add up to a specific number. A case study in 2011 [1] shows that the mean time to solve a Sudoku puzzle is between 8-23 minutes. Two web sources were used for this study, the first was used by expert puzzle solvers, while the second was used by a mix of skill levels. This explains the difference in solving time. However, Helmut Simonis [2] describes Sudoku as a constraint satisfaction problem using different propagation techniques to solve them within a few milliseconds.

Sudoku or killer Sudoku puzzles are commonly found in places such as a newspaper or a magazine, with various levels of difficulty. Since they are available easily, they invite people with all levels of ability, but without the knowledge of solving them efficiently, it can lead to a frustrating end. Both these types of puzzles use different deduction techniques such as Lone Ranger, elimination techniques, seeking twins and triplets in Sudoku as described by Lee Wei-Meng [3], and sum elimination, rule of k, and rule of necessity in killer Sudoku as described on killer Sudoku online [4]. Often the answer to the puzzle is not available straight away. For example, you may have to wait until the next day to find the solution. However, by using a solver, the puzzle could be solved in a few seconds [2] and can be used to compare solutions. In most cases, however, receiving the whole solution is not an ideal learning mechanism, instead providing hints or given solutions to a particular cell is more beneficial.

Therefore, constraint propagation can be used to deduce how the solution to a cell was identified which can make people aware of the different techniques used to solve these problems. Another problem is that using a solver requires the user to enter the values for the Sudoku puzzle and cages for Killer Sudoku manually. This can be a tedious task and prone to error, which can waste time and cause users to quit. The most practical solution to this problem is to use machine vision to allow the user to input an image of the puzzle and then convert the puzzle to a form with can be used by the solver.

## Aims and Goals

The first and main goal of the project is to create my solver for Sudoku and killer Sudoku so I can solve these types of puzzles incredibly quickly. I will research this topic and use techniques such as backtracking, value heuristics, and forward checking to create an efficient solver. I will first create a base solver with minimal efficiency techniques and then compare it with more efficient solvers. The final solver should be able to provide the user with an explanation of how the solution was achieved, for example, which technique was used. This will be used to write the report on what constraint solvers are the techniques involved and how the different solvers compare to each other. I will also write a separate report on human techniques for solving Sudoku and killer Sudoku problems.

The next milestone of the project is to create a machine vision algorithm that will allow users to upload images and then directly convert them into a 2D array. The purpose of this feature is to prevent the user from having to enter the values into the puzzle manually. There is also a sub-milestone where I will need to use machine learning to convert the number in the image to an integer which can be inserted into an array that represents the puzzle. Next, I can then pass the array to the solver and then generate a solution to the puzzle. I will also write a section on the report on what machine vision is and how I used OpenCV to build the algorithm.

Another milestone is to use the Django framework to create a website that will host the solver. I will create an interface to allow the user to play the game on the website and a feature that allows them to upload images of puzzles. The interface will display to the user how a solution was found for a puzzle. Game playing will be a key feature of the interface, it will allow users to play the puzzles and support common features such as taking notes, hints, and undoing actions. The purpose of establishing the application on a website is because they are very accessible and can be developed for any device with little code change. The interface should be simple, easy to use, and have a minimalistic design, so the focus is on the intended features.

In addition, I will also generate my own Sudoku and killer Sudoku puzzles, which can then be played on the website. I will try to generate puzzles of different difficulties so it can accommodate beginners to experts. I will also write a section in the report on how I could generate my puzzles and the algorithms I used to create them. As well as coding milestones, I will also have several more report milestones such as a section on the NP-hardness of Sudoku and killer Sudoku and the time complexity of my solutions. Additionally, I will write reports on the data structures I have used to develop my application and the different software engineering processes I have used.

## Literary Survey

In solving the Sudoku puzzle, CSPs are a widely used approach. CSPs operate by modelling the problem as a set of variables to optimise the domains of the variables and constraints between variables. One such approach [5] describes base backtracking, which tries all values in the domain to find a solution. They also use constraint propagation techniques such as arc consistency, forward checking, and value and variable ordering heuristics to improve the speed of the algorithm. This approach relies on the principle of "fail fast" described in [6] which tries to choose variables that have the smallest domains and hence fail quicker if the wrong value is assigned. This approach was built upon in [7] which performs constraint programming but after each run, if a conflict occurs it swaps the value which causes the conflict and carries on. The approach described in [8] uses the recursive as a last resort, it first tries to apply reasoning to reduce the domain space of the variable and when it can't continue, the recursive algorithm is used. For my use case, I want to show users how a solution was used, therefore using the algorithm described in [8] will help me accomplish that.

Machine vision is effectively used to allow a computer to see using a camera. One approach described in [9] is to extract a puzzle to process the image and use a machine learning model to detect if a puzzle is present in the image and find the edges of the puzzle. Another approach [10] is to avoid using machine learning and directly use the OpenCV library to filter out the noise in the image and try to detect the corners of the puzzle. After finding the edges of the grid the next step is to extract each cell and then use machine learning for number recognition which converts the image of the cell to a number. Techniques such as Multilayered Perceptron, Support Vector Machine, and Convoluted Neural Networks [11] are all useful methods used to recognise digits. This will improve the usability of the application, as the user is not required to manually enter the values for the puzzle. However, it's unlikely the model will have 100% accuracy therefore to account for the errors made, the user can then manually replace values in the grid. Instead of using machine learning, another option is to use the Tesseract engine which performs text recognition all by itself [12], this approach is widely used in the world for various tasks such as license plate detection. For my solution, I will mainly use the machine vision section as [10] requires an extensive dataset to be created from scratch, which is not feasible. As for the number detection, I will initially try using a machine learning model and if it performs badly, I will switch to the Tesseract engine.

Another key area of my application is generating valid Sudoku and killer Sudoku puzzles, for a puzzle to be valid it must satisfy all the constraints of the puzzle but also only have one solution. One such approach to do this is described in [13], this method first gets a Sudoku grid which has all its cells filled, it then performs the inverse of known solving methods to remove a set number of values. Another approach described in [14] starts again with a filled-in grid. It then removes values one by one and checks if the puzzle is valid. This is done until the desired number of clues remain, this approach is a lot slower but always guarantees that a puzzle is valid. There are also more complex approaches such as using DNA computing as described in [15] which uses graphs and graph colouring to generate the puzzle. For my use case, I am going to try to implement the method described in [14] which will highly depend on how fast my solver is, and try to integrate [13] in cases of generating different difficulties of puzzles.

# Constraint solvers

## Introduction

Constraint Satisfaction Problems (CSPs) are a type of programming paradigm where the problem is modelled as a set of variables that we are trying to optimise, the domain space of the variable, and the constraints of the model [13]. They are very powerful for solving problems such as the N-queen problem, scheduling rosters, and playing games, where the goal is to find a solution where no constraints have been broken. Constraint programming (CP) tries to find the solution to these problems typically by a search algorithm and uses specific algorithms such as arc consistency, dynamic value ordering, forward check, and back jumping to increase its efficiency. Sudoku is one such problem that can be solved very efficiently using CPSs and in this section, I will write about the different algorithms which are used within CPSs.

## Backtracking and Recursion

Recursive backtracking is a type of algorithm that tries to find a solution to the problem by trying all values. It is usually implemented within a single function and is very easy to build. With Sudoku, the algorithm will get the first empty cell, assign it a value from its domain, and then call itself to fill the next empty cell. The backtracking part of this algorithm is what allows it to find solutions, if a constraint has been broken then the correct solution is wrong and will not lead to a correct solution. Therefore, the algorithm will backtrack to the last state in which the constraint was satisfied and try a different value.

def RecursiveBacktrack():

if isBrokenConstraint() == True

return False

if isComplete() == True:

return True

variable = getNextUnassignedVariable()

for value in variable.domain:

variable = value

if RecursiveBacktrack() == True:

return True

return False

The above code snippet shows the basic structure of a recursive backtracking algorithm. It backtracks if a constraint is broken, or no value leads to a solution and succeeds when the problem is complete. The benefit of this is that if a solution is possible, it will always find it. However, a major drawback is that it is highly inefficient since it may fail numerous times time until it finds the correct solution. For Sudoku, there is a possible 6.6 × 1021 valid Sudoku grid [16] and therefore trying every single value can take a very long time to complete making this algorithm very inefficient. In most cases the backtracking algorithm should not be used on its own, instead, the algorithm should be improved by adding other techniques such as arc consistency and dynamic value ordering.

## Arc Consistency

One way of improving the backtracking algorithm is using arc consistency. This algorithm does not solve the problem instead, it tries to reduce the domain space for each variable. It is often described as a preprocessing step conducted before the search algorithm and its goal is to remove any inconsistent values from the domains [17]. Here, inconsistency is defined as any value that does not lead to a solution, and by doing this, we can reduce the search space and therefore make the search algorithm faster.



Figure 1 – a row within a Sudoku puzzle.

From Figure 1, you can see that the values 2, 1, 3, 6, and 4 already belong in the row. Enforcing arc consistency here would mean that the domain for the empty cells would not contain those values because they will never lead to a solution. Without arch consistency, the search algorithm would have included those values in the variable domains and ultimately failed resulting in significant wasted time.

## Back jumping

Back jumping is another algorithm that is used to make the standard backtracking algorithm faster. When we have reached a dead end, where a variable domain is empty, we usually backtrack in chronological order, but in back jumping, we backtrack directly to the variable assignment which causes the dead end to be reached [18]. By doing this, we can avoid searching with a variable that will not lead to the correct solution and backtrack further to a safe state. Programmatically, during each recursive call the variable will have a conflict set that contains the variables that were involved in the conflict, the values assigned to those variables and the specific constraint that was broken. The conflict set tells us the conflict variable that is most responsible for the conflict and hence backtracks to change the value assigned to that variable and then resume the search. This way, we have pruned the branches between the conflicted variable and the last variable and thus improved the speed of the algorithm.

## Dynamic Variable/Value ordering

Dynamic variable ordering is another concept within CSPs, and its main concern is the order in which the variables are processed. The idea is that if we carefully choose the next variable, then we can minimise the size of the search tree and prune a branch as quickly as possible if it is incorrect. This approach is also called MRV (most restricted variable) [19] which implements the "fail first" [6] approach. In this concept at every recursive call, we choose the variable with the smallest domain which reduces the number of branches compared to a variable with more options. Also, with a smaller domain, there is a higher probability that the chosen value will be correct for example if the domain of a cell is just 1 or 2 then there is a 50% chance that we are right on the first try. Dynamic Variable ordering can be implemented efficiently using a priority queue, we can first put all variables into the queue with their domain sizes to be ordered. Then, when we make a recursive call, we choose the next value by popping from the queue.

Another approach, which is like the MRV is Dynamic Value Ordering, but instead of thinking about the variable to choose, we look at the value to assign. The main idea is that if we choose the values carefully, then we can increase the probability that the value we assigned is correct and reduce backtracking. There are many approaches such as a LEX which chooses the least fixed value and a MFV where the heuristic chooses the most fixed value [19]. However, in the case of Sudoku, this doesn't work, but we can observe that each value 1-9 must appear exactly 9 times. Therefore, we can choose the value which is least used so far and the heuristic.

## Forward checking

Forward checking is an algorithm that is concerned with verifying and updating the domains of the related variables after assigning a value to a variable [2]. The main idea behind this algorithm is to reduce the domain of the variables and thus reduce the size of the search tree. For Sudoku, after assigning a value to a variable we can implement forward checking by then removing that value from all other cells which are in the same row, column, and box. During this process, if any variable domain is empty then we know that the current assignment was wrong so we can immediately backtrack and prune the branch earlier. In addition, we can exclude a value that will lead to a conflict later and narrow down the search even more.

Def forwardCheck(current\_cell, assigned\_value):

For cell in relatedCells:

If assigned\_value in cell.domain:

cell.domain.remove(assign\_value)

If len(cell.domain) == 0:

Return "backtrack"

The above pseudocode describes an implementation of the forward checking algorithm, which is called each time a value is assigned to a value. It loops over all the related cells and removes the assigned value if it exists in its domain. Then it checks if the domain for the cell is empty. If it is, then it triggers a backtrack.

# Recursion and Backtracking

## Introduction

The first task which I worked on was working on a solver for Sudoku which involved learning about constraint solvers and then onto the killer Sudoku solver. Various libraries are designed for constraint satisfaction problems such as python-constraints, CPMpy, Google OR-tools, and more. However, I wanted to create a solver from scratch without using a predefined library, as it gave me more control over how I wanted the solver to work. First, I created a baseline model, which used only backtracking and Consistency. Then I will try to create an improved model using value ordering heuristics to make the model faster by allowing incorrect branches to be pruned earlier.

## First Sudoku Solver

An important part of constraint solving is to identify the domains for the variables within the problem. I started by creating a getDomain method, which takes a row and column and returns an array containing all the values the cell can take. An obvious starting place for this would be to assign the numbers 1-9 to every cell and let the backtracking function deal with the incorrect values. However, I realized that this was not a good idea because some values would be guaranteed to not be correct so I could remove those values from the domain.

### Setting up a Sudoku class

The first step to building the solver is to have a base class that will be used to represent the problem better. I started by defining a simple class called Sudoku and its constructor, which accepts a 2D array. A zero in a slot represents that the cell is empty and if it contains a number between 1-9, then it contains a given hint. While this is the main behavior, I also decided to add an isValid method which checks whether a puzzle is valid. For a puzzle to be valid, it needs to satisfy all the constraints, such as each cell in a row, column, and box must be unique. Another requirement is that the puzzle must have exactly one solution, however, this cannot be implemented yet because it relies upon the solver being built.

    def checkBox(self, row, col):

        row = row \* 3

        col = col \* 3

        unique\_values = {}

*for* i *in* range(3):

*for* j *in* range(3):

*if* *self*.grid[row+i][col+j] != 0 and *self*.grid[row+i][col+j] in unique\_values:

*return* False

                unique\_values[*self*.grid[row+i][col+j]] = 1

*return* True

The above code snippet is used to if a box is valid, it does this by first identifying the cells involved in the box and then putting all the values into the dictionary as it sees them. If a new value is already in the dictionary, it means there is a duplicate, therefore the puzzle is invalid.

### Arc-consistency

Arc consistency is a constraint propagation technique used within constraint solvers to reduce the size of a domain by filtering out inconsistent values. With Sudoku, an example of an inconsistent value would be any number that appears anywhere else in the cell row, column, or 3x3 box. This is because adding this number would not satisfy the Sudoku constraints defined. I implemented this technique by removing all the values within the cells domain which would violate the defined constraints.

   def getDomain(self, row, col):

        used = []

*for* i *in* range(9):

*#Get all values in the row*

*if* *self*.Sudoku.grid[row][i] > 0:

                used.append(*self*.Sudoku.grid[row][i])

*#Get all values in the column*

*if* *self*.Sudoku.grid[i][col] > 0:

                used.append(*self*.Sudoku.grid[i][col])

*#Get all values in the box*

        box\_row = (row // 3) \* 3

        col\_box = (col // 3) \* 3

*for* i *in* range(box\_row, box\_row + 3):

*for* j *in* range(col\_box, col\_box + 3):

                used.append(*self*.Sudoku.grid[i][j])

*# getting all unique values*

        used = set(used)

*return* set([1,2,3,4,5,6,7,8,9]) - used

The above code listing the two for loops iterates through the given cells' rows, columns, and boxes to get all the values that have already been used. It then simply returns all the values not within this set which are in the range one to nine. By using this approach, the domain space for a cell is decreased and we avoid trying values that were guaranteed to be incorrect and therefore improve the efficiency of the solver.

### Recursion and Backtracking

With the completion of the getDomain method, I can now work on the core part of any constraint solver which is the backtracking algorithm. The algorithm works by first getting a cell that doesn't already contain a value and then assigning it a value from its domain and then recursively calling itself until every cell has a value. If a cell has no value in its domain (due to an incorrect value being assigned to a variable earlier) the algorithm will backtrack and try another value.

    def solve(self):

        row, col = *self*.findNextEmpty()

*if* row is None:

*return* True

        domain = *self*.getDomain(row, col)

*for* value *in* domain:

*self*.Sudoku.grid[row][col] = value

*if* *self*.solve():

*return* True

*self*. Sudoku.grid[row][col] = 0

*return* False

The findNextEmpty method returns the first slot in the array which does not contain a 0 in it. If the method returns None it means that there are no more empty cells therefore, the puzzle has been completed. The backtracking algorithm is usually the backbone of a constraint solver because, while you may have other constraint propagation methods, they may not always find a solution. The benefit of backtracking is that it will always find a solution to a problem if the puzzle is valid. However, a problem of backtracking is that it is slow because it is using trial and error to find a solution and in the worst case it would have to try every value in the domains for all cells. Therefore, it is important to have other constraint propagation techniques (explained in a future section) to do most of the heavy lifting and use the backtracking algorithm.

## Improving the Solver

The solver can be made faster by carefully choosing the next cell to be assigned a value. Currently, the algorithm chooses the first empty grid slot. However, we can do better by choosing the slot with the smallest domain space first. This strategy is often called the "fail-first" approach described by Haralick [5] and by choosing the cell with the smallest domain, we can discard values that do not lead to a correct solution quicker. Also, with a smaller domain, we have a higher probability that the value assigned is the correct one.

### Fail first value ordering heuristic.

To implement this strategy, I decided to implement a priority queue to store all the cells and their domain space in order of the smallest domains first. Since I wanted the heap to be efficient, I adapted the code provided by the Python documentation [20] to implement the priority queue using the heapq library. Before running the backtracking algorithm, I first need to instantiate the queue by giving it the domains for each empty cell.

    def setupHeap(self):

*for* i *in* range(9):

*for* j *in* range(9):

*if* *self*.Sudoku.grid[i][j] == 0:

                    values = *self*.getDomain(i, j)

*self*.heap.addToHeap((len(values), (i,j), values))

In the code snippet, I show how the queue is instantiated and the reason I store the actual domain values in the heap is so that we do not need to recompute the domain values each time. However, this approach means that we need to manually keep the domains for each cell up to date every time we guess a value for a cell. I will discuss the implementation of the queue in a later section.

Now with the queue established, I no longer require the getNextCell method as I need to pop the first value in the heap to get the next best cell and assign it a value from its domain. After assigning a value, all the cells in its row, column and box need their domains to be updated to remove the assigned value.

        removed = []

*for* i *in* cells:

*if* val in *self*.key\_map[i][3]:

                m\_set = *self*.key\_map[i][3]

                m\_set.remove(val)

                removed.append((i, val))

*self*.addToHeap((*self*.key\_map[i][0] - 1, i, m\_set))

*return* removed

In the above snippet, I check each related cell's domain and see if it contains the value assigned, if it does then I keep the original copy of the cell and its domain and push the updated copy into the queue. After processing all the cells, the array containing the original domains and cells is returned to the recursive frame. The reason I store the original copy is that if the cell was assigned the wrong value, then we need to put the original domains back into the queue. So instead of recomputing the previous domains, I can simply put back the stored domains.

*for* updated *in* updatedCells:

            m\_set = *self*.key\_map[updated[0]][3]

            m\_set.add(updated[1])

*self*.addToHeap((len(m\_set), updated[0], m\_set))

In the above code, the updatedCells variable is the array from the previous code snippet. So, we just put the removed values back into the domain and push it into the queue.

I can keep the updating of the queue efficient by keeping each cell in a dictionary as a key and its entry in the queue as the value. Therefore, locating the entry for a cell can be done in O(1) time. Also, instead of removing old domains from the queue, I can mark them as being ignored. While this means that the queue gets filled with old values, making the pop method less efficient, it makes removing values significantly quicker.

## Comparing the Sudoku solvers

In this section, I will compare the two different solvers have built and look at how they perform with different difficulties of puzzles.

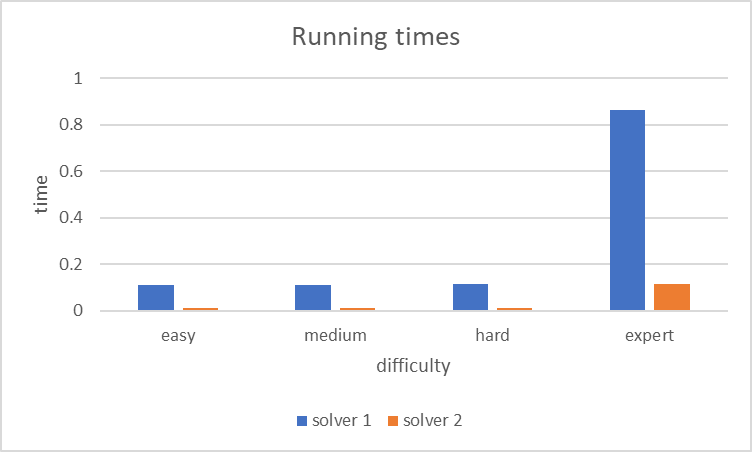


Figure - Graph showing how the solvers perform on different puzzles.

In Figure 2, the bars for the easy, medium, and hard puzzles represent ten thousand runs of both the solvers. As for the expert puzzle, it is built specifically to be as difficult as possible, and the graph shows just one run of the puzzle compared to the ten thousand runs of the other difficulties. From Figure 2, it's very easy to see that the second solver with values ordering heuristics and forward checking is better than the first solver with just arc consistency. On average, the second solver took around 0.01 seconds while the first solver took around 0.1 seconds, so it's roughly 10 times better. Another interesting point is that the running time for the easy, medium, and hard puzzles took roughly the same time for the solver to complete, whereas for humans, as the puzzles get harder, the time to complete would also get harder. However, while the solvers are fast, the expert puzzle shows that there is still room for improvement. This is because backtracking algorithms on their own is not very efficient.

## Killer Sudoku

My approach to the killer Sudoku solver was very similar to the Sudoku, with some minor changes due to how the class was established. I started by implementing a baseline solution, which just used a backtracking algorithm and arc consistency and then implemented a second version with dynamic value ordering and forward checking.

### Base class

My first step was to implement the killer Sudoku base class, which would hold the grid and the cages associated with it. For the grid, I used the 2D array and as for the cages; I opted for a dictionary that had the cage number than keys and another dictionary as the value. The second dictionary had a key to the cage sum and the values of the cells in the cage. The reason I set the cages up like this is so that I had a way to quickly get all the cells in a particular cage. However, a problem was that given a cage it was difficult to get its cage number, so I implemented a method that went through the cages dictionary and instantiated another array with the key as the cell and the value as the cell cage number. With this, I could get all the information about a particular cell in O(1) time. I also created a method that checks whether the puzzle was valid, by checking if all cells were contained in a cage, all cage sums added to 405, and if the Sudoku requirements were met.

### Backtracking algorithm

To implement arc consistency, I had to create a method to get the domains for the cells that will be used in the backtracking algorithm. The main part of the method was the same as the Sudoku getDomain method. however, I needed to add some code to factor in the cage cells. Since a rule of killer Sudoku is that there can be no duplicates with cages as well, I added the following code to the method.

*for* i *in* cageCells:

*if* *self*.KSudoku.grid[i[0]][i[1]] != 0:

                cageSum = cageSum - *self*.KSudoku.grid[i[0]][i[1]]

                used.append(*self*.KSudoku.grid[i[0]][i[1]])

                count += 1

*if* cageSum <= 0:

*return* {}

*if* count == len(cageCells) - 1:

*if* cageSum > 9 or cageSum in set(used):

*return* {}

*return* {cageSum}

        used = set(used)

        validGuesses = set([1,2,3,4,5,6,7,8,9]) - used

        validGuesses = {i *for* i *in* validGuesses *if* i <= cageSum}

*return* validGuesses

In the above code first, I retrieve the cage number and then the cells in the cage, and next the for loop, add all the given values to the used array. Instead of just returning the numbers not in the used array as before, I added an extra check. If this was the last cell to be filled, then there can only be one value, which is the cage sum minus the other cage values. Then I can return the cage value if it is not present in the used array, otherwise it returns an empty array which triggers a backtrack. Also, I added a condition that returned an empty array if the remaining sum of the cage was 0 or below because it meant there was no value in the 1-9 range which could get a 0 or below. The final task was to implement the backtracking algorithm, but this was a very easy task because the code was exactly as described for the first Sudoku solver. The only difference was the getDomain method as described above.

### Improved Solver

To improve the solver, I once again implemented dynamic variable ordering into the solver to decrease the size of the search tree. I used the same concept as the priority queue, as described in section 2.3.1. While the concepts used were the same, they were implemented slightly differently; the difference was in the updating of the priority queue.

## Comparing the two Killer Sudoku solvers

In this section, I will compare the two solvers I have implemented for the Killer Sudoku. For this section, I have used Sudoku.com [19] to generate some puzzles with different difficulties and run the two solvers on them.

Figure - A graph showing the running time for the killer Sudoku solvers.

Figure 3 shows that both solvers are very quick at solving the easy puzzles, with both completing the puzzles in 0.3 seconds. For the medium puzzle, the initial solver seems to perform better by approximately 5 seconds. One reason for this is that because I need to constantly update the priority queue after making a guess, the running time is increased. However, as shown above, the running time for the hard puzzle is significantly different, the second solver is approximately 12 minutes faster than the first. This demonstrates that the second solver performs significantly better on tougher puzzles while the first does better with easier puzzles. One observation is that the killer Sudoku solvers are significantly slower than the Sudoku solver, that is because while Sudoku puzzles usually contain a few prefilled cells, a killer Sudoku is empty. This means that the algorithm must fill the grid from scratch, as there are 6.6 × 1021 valid Sudoku grids [16] and more for killer Sudoku since there are many cage arrangements, meaning this can take a long time. Figure 3 demonstrates the need for a better solver because taking more than a minute would decrease interest in the application.

## Final Killer Sudoku Solver

The main issue with the first two versions is that they do not deal with the domains of the cell accurately. This means that often cell will contain values in its domain when they can be logically ruled out. The role of arc consistency and forward checking is to keep the domains as accurate as possible, which means the algorithm does not try values which are guaranteed to fail. However, with Killer Sudoku, this is not enough because the current algorithm does not consider the cage constraint well enough.

The first change I made was to update the way the domains are initially calculated. Initially, it was set to contain all values not already in the row, column, row, or cage and be below the cage sum. However, with cages whose cage sum is over 9, the domain would just contain all values because with an empty puzzle no values can be removed. Since this is often the case, most of the cells will contain large domains and make the algorithm very inefficient. Through research, I found that for each cage length and sum, there are specific combinations which allow for a much more accurate domain calculation [21].

A screenshot of a computer

Description automatically generated

Figure 4 - all possible combinations for cages of length 3. [21]

Figure 4 shows an example of some combinations. By defining a class to containing all the combinations, I now had a more accurate way of determining the domains for cages and avoid values which would be wrong. The last challenge was to understand how the domain should change during the algorithm. To explain the logic, if a cage had 3 cells and a cage sum of 11, figure 4 shows its domain could have the combination 128, 137, 146, 236 and 345, so the domain would be 1,2,3,4,5,6,7. If the algorithm assigned 7 to one cell, we can see that the only remaining combination would be 137, so the domain of the other two cells would only be 1 and 3. In the previous iteration of the algorithm, it would just remove 1 from the domain leaving 2,3,4,5,6,7 as the domain for the other two cells. This is a big improvement, and it allows for the domain to be as small as possible and therefore a faster solving time. The code which keeps the domains consist of is the same as the code described in section 7.5, which is the cage reduction.

Figure 5 - The running time for the final Killer Sudoku Solver

Figure 5 shows that the new and improved solver performs extremely well compared to the previous two versions of the solver. While the easy and medium shows a smaller improvement, the hard difficulty is where the new solver displays its strength. The previous versions took a few minutes to solve while this version only takes about 8 seconds which is significantly faster.

## Conclusion

To summarise this topic, we have seen that a basic backtracking algorithm is quite slow for both puzzles. However, using techniques such as arc consistency, forward checking, and variable order heuristics we can significantly improve the speed of the solvers. The reason these methods improve performance is because they keep the sizes of the domains to be as small as possible which means that the search depth can be as shallow as possible and lead to a solution much faster. This works very well for Sudoku puzzles, but as for Killer Sudoku, these techniques are still not enough. These techniques do not work as well for Killer Sudoku puzzles because they do not consider for the cage’s constraint. By having access to tables of combinations for all length of cages and cage sums we make sure cells only contain values which could lead to a solution. This means we can keep domain values small and thus lead to a solution quicker.

# Data Structures in Solvers

## Priority Queue

### Adding to the heap

The Constructor for the class initialises the following items:

*self*.pq = []

*self*.key\_map = {}

*self*.REMOVED = '<removed-task>'

*self*.counter = itertools.count()

The pq array represents the priority queue, which is implemented using the heapq python library, the key\_map dictionary is used to map the query entry, so accessing random elements in the queue can be done in O(1) time. The REMOVED variable is used as a placeholder for removed items and, finally, the counter is used to rectify cases when the items have the same priority. With the code set up, I can implement the first method which is adding elements to the queue.

    def addToHeap(self, item):

*if* item[1] in *self*.key\_map:

*self*.remove\_cell(item)

        count = next(*self*.counter)

        entry = [item[0], count, item[1], item[2], "available"]

*self*.key\_map[item[1]] = entry

        heapq.heappush(*self*.pq, entry)

The method first checks if the cell entry already exists within the queue, if it doesn't then we can push the element into the queue and save a pointer to the entry via the key\_map dictionary. If the cell is already in the queue, then we need to remove the item, so we only have one "active" entry per cell. Finally, an individual entry contains the priority of the entry, which is represented by the length of the cell's domain. The second element is the count, which is a unique incrementing value. So, where two cells have the same priority, the cell pushed first has more priority. The third is the row and column of the cell, the fourth is the actual domain of the cell and, finally, the string represents whether the elements are "active" or not. The reason the actual domain is in the entry is so that we do not need to recalculate the domain of the cell each time, I can use the store values.

### Removing and popping items

When discussing items in the queue, I referred to them as "active" or not in the last sections. This is because when I want to update an entry, instead of popping it and reordering the queue, we can ignore the element by giving the entry the REMOVED tag as defined in the constructor.

    def remove\_cell(self, task):

        entry = *self*.key\_map.pop(task[1])

        entry[-1] = *self*.REMOVED

The above code does exactly that it, finds the entry using the key\_map dictionary and changes the last element to the removed status. As for the pop method, it uses the heappop method predefined in the heapq library.

    def pop\_cell(self):

*while* *self*.pq:

            priority, count, cell, domain, status = heapq.heappop(*self*.pq)

*if* status is not *self*.REMOVED:

*del* *self*.key\_map[cell]

*return* priority, cell, domain

*return* None, None, None

The while loop is needed so that we can keep popping entries until we get an entry that doesn't contain the removed tag, hence ignoring the entry. If the entry is available, then we delete the mapping and return the cell, priority, and domain to be used by the solver. If the queue is empty, then we have successfully assigned a value to every cell and so return None for all, which internally signals the end of the solving algorithm and returns the completed grid.

### Updating the queue

After the solver makes a guess for a particular cell, all the cells within the row, column, box, and in killer Sudoku, the cage needs to be updated. Also, where we need to backtrack, the changes made previously need to be reverted so to re-update the affected cells. Since updating and reverting are going to be needed a lot, multiple times each recursive call, this needs to be efficient. The heapq library implements the priority queue using a binary heap internally, therefore removing random entries is very inefficient, so by using the available and removing tag we can ignore "removed" elements. The first part is updating the cell domains after a guess has been made:

*for* i *in* cells:

*if* val in *self*.key\_map[i][3]:

                m\_set = *self*.key\_map[i][3]

                m\_set.remove(val)

                removed.append((i, val))

*self*.addToHeap((*self*.key\_map[i][0] - 1, i, m\_set))

*return* removed

The above is a snippet of the decreaseKey method which updates cell entries. For the case of Sudoku and even non-cage cells in killer Sudoku, only one value is inconsistent, which is the value used when the guess was made. Therefore, instead of manually recalculating the domains every time we need to check if that value exists in the cells domain which is saved in its entry in the queue. If it does, then we remove it and add a new entry for the cell in the queue, otherwise, we move on. At the end, we return all the cells that were updated. This is important for the reverting stage.

When a backtrack occurs, we need to revert the queue to the state before the last guess was made. By getting all the entries that were updated in the decreaseKey we can implement a simple method that just puts those entries back into the queue.

    def increase key(self, updatedCells):

*for* updated *in* updated cells:

            m\_set = *self*.key\_map[updated[0]][3]

            m\_set.add(updated[1])

*self*.addToHeap((len(m\_set), updated[0], m\_set))

The above code deals with the reverting of the queue. It takes all the entries that were changed previously and puts them back into the original state. By doing this, the previous action is undone, and the queue is kept consistent.

When updating cells that are part of the cage, we cannot remove the values guessed, as is the case with non-cage cells. Instead, the domains change in a few ways but dealing with this manually is around the same as just recalculating the domain therefore that is the approach I went with.

## Dictionaries and Sets

When developing the solver there are various places where I have implemented a dictionary or a set. This is because they are very efficient and allow access to an element to be done in O(1) time, while they can be space inefficient, this is not a concern and I have prioritized the time complexity. The cages in my killer Sudoku class are implemented using a dictionary, this is a convenient way to store them and identify cages. However, finding which cage a cell belongs to and hence getting all its other cage cells is very inefficient. Therefore, I implemented another dictionary that maps each cell to its cage number, now I can get all information about the cages in O(1) time. This is important because the solver frequently needs to know which cells are grouped together and if implemented inefficiently, then the running time could grow by a lot.

Sets are also a useful data structure that is used mainly when dealing with the domains of a cell. When checking if a domain contains an element, using sets is very useful because I do not need to loop through the set like an array. Instead, it uses hashing to check if the values exist within a set that is O(1) in time complexity.

# Machine Vision

The ability for humans to see is something that happens seamlessly without needing to deduce what it is that we are seeing. When looking at a tree, we just "know" it is a tree we do not need to process it. Machine vision is the outcome of allowing computers to see [21]. A computer can see through images and videos but requires time to process the information and extract meaningful information from it. In this section, I will describe how machine vision was used to extract the Sudoku and killer Sudoku puzzles from an image and convert it to a form understood by the solver. The algorithms in this report are based on the algorithms described in [10].

## Grid extraction

The first task for the machine vision is to find and extract the puzzle from the image and hence work an image containing only necessary information. To do this, we can take advantage of the signature thick borders of a Sudoku and killer Sudoku puzzle. However, before doing this, we must convert the image the image to grey-scale and apply a threshold that aims to reduce the noise in the image.

image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

*# Applying adaptive threshold to the block*

threshold = cv2.adaptiveThreshold(image, 255,

cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C, cv2.THRESH\_BINARY, 91, 0)

The above code snippet first converts the image to a gray-scale and then applies an adaptive threshold to remove the noise in the image. Thresholding works by looking at the neighbors of a pixel in this case, 91 neighbors and finds the mean pixel value to be assigned. All values below a mean are set to 0 (white) or 255 (black) otherwise.

A square with numbers on it

Description automatically generatedA square puzzle with numbers and a square grid

Description automatically generated with medium confidence

Figure – An image containing a Sudoku puzzle (Left) and the same image after applying the threshold (Right)

Now that the image is in the correct format, we must find all the contours which are contained. A contour is defined as the continuous border that encompasses an object, the border of the Sudoku is a contour because of the outline that surrounds the puzzle. OpenCV provides a method findContours to do exactly this, however, because the image is still noisy, it will detect a lot of fake contours shown in figure 4 below.

A green square with black numbers on it

Description automatically generated

Figure – the original image with all the edges drawn on it.

This problem can be solved by sorting all the contours by area and ignoring all the smaller contours. Looking carefully at Figure 5, the results show that the meaningful contours such as those around the numbers, words, or the puzzle itself have a larger area. Through sorting we can then identify the contour which surrounds the puzzle by getting the second largest contour, this is because the largest will be the image border. This is the assumption I am going to make because in most cases the puzzle may not be the second-largest contour, however, it makes the code significantly easier to write.

contours, hierarchy = cv2.findContours(image.copy(),

cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_SIMPLE)

*# sorting the contours by areas*

cnt = sorted(contours, key=cv2.contourArea, reverse=True)

*# The second contour in the puzzle the first is the border of the image.*

puzzleContour = cnt[1]

The above code snippet shows how to find the contour that surrounds the puzzle. It first finds all the contours in the image, cv2.RETR\_TREE simply states to find all contours inside the image. Next, the contours are sorted through the sorted method and finally, we just take the second contour from the image to be the puzzle. Figure 6 is the final output which shows the image with its contours, which contains the puzzle drawn on it.

A crossword puzzle with numbers and a green border

Description automatically generated

Figure – the original image with the puzzle contour drawn on it.

## Straightening the Image

Before moving on to extract each cell from the image, the image must be straightened to avoid dealing with unnecessary information. As shown in Figure 6, the puzzle itself within the image does not form a perfect square, particularly on the left side, the line appears slightly tilted. We solve this problem by effectively stretching the image to fit inside a 450 by 450 box. To do this, we first need to find the four corners of the contour, this is because the contour comprises a set of points, so we need to identify which one's make up the corners.

To find the corner I used the Ramer-Douglas-Peucker algorithm [22] which recursively tries to remove points within the contour if they are not within a set parameter distance apart to get a smoother contour. Since the puzzle has four corners and are straight lines, the algorithm will remove all contours in between the corners and give exactly the four corners of the puzzle. Now that we have the corners, we can stretch or shrink the puzzle to fit within our set box.

def straightenImage(self, processedImage, edges):

*# Creating a 450 by 450 template image*

        dst = np.array([[0, 0], [450, 0], [450, 450], [0, 450]],

dtype='float32')

*# Calculate the perspective transformation matrix*

        M = cv2.getPerspectiveTransform(edgePoints, dst)

*# Apply the perspective transformation to the image*

        output = cv2.warpPerspective(processedImage, M, (450, 450))

*return* output.

The above code snippet is used to convert the puzzle into a perfect square. It starts by creating a canvas for the final image, which is 450 x 450 pixels. Next, the getPerspectiveTransform uses the corner points of the contour and the target canvas to create a matrix M which is a transformation of the puzzle to fit within the canvas. Finally, warpPerspective performs the transformation of the image giving us the final image which is shown in Figure 7.

A square grid with numbers

Description automatically generated

Figure – The puzzle after applying the Ramer-Douglas-Peucker algorithm.

## Cell Extraction and Processing

Now that we have an image of just the puzzle and are fitted within a 450 by 450-pixel image, the next step is to extract the individual cell within the image. The reason that I wanted the image to be fitted perfectly within a 450 by 450 image was so I could divide the image into 50 by 50 blocks. Due to Sudoku and killer Sudoku being 9 by 9, each 50 by 50 block will give us each cell within the puzzle.

    def CellExtraction(self, image):

*# defining the height and width for each cell*

        cell\_height = 450 // 9

        cell\_width = 450 // 9

        cells = []

*# looping through each cell in the image.*

*for* y *in* range(0, 450, cell\_height):

*for* x *in* range(0, 450, cell\_width):

*# getting each 50x50 block from the image*

                block = image[y:y+cell\_height, x:x+cell\_width]

*# Appending the extracted block to the cell array*

                cells.append(block)

*return* cells

The code snippet above is a simple algorithm that loops through every 90 by 90 blocks, extracts it, and appends it to an array. At the end, the cells array will contain all the 81 blocks that capture each cell in the puzzle. The next step is to identify which cell contains an integer in it and which does not. To do this effectively, I made another assumption which is that the integer must be in the center of the cell not touching the edges and would have a width of more than half the cell height.



Figure – the individual images after extracting.

  digit\_Cells = []

        contours, \_ = cv2.findContours(image, cv2.RETR\_TREE,

cv2.CHAIN\_APPROX\_SIMPLE)

*# Only process cells with more than one contour.*

*if* len(contours) > 1:

            cnt = sorted(contours, key=cv2.contourArea, reverse=True)

*# check if the contour is centered and the right size for a*

*digit.*

            x, y, w, h = cv2.boundingRect(cnt[1])

*if* h >= 46 // 2 and *self*.isCentered(x, y, w, h) == True:

                digit\_Cells.append(image[y:y + h, x:x + w])

*else*:

                digit\_Cells.append(-1)

*else*:

            digit\_Cells.append(-1)

The above code snippet above tries to code the assumptions I made earlier. It starts by finding all the contours of the image. If there are none, then we can be confident that no integer exists in the image. Otherwise, we check whether the biggest contour is of the correct dimension and centred. If it is, then we can assume it is an integer. If an integer has been found in a cell, it needs to be extracted from the image, which is performed by converting the contour into a rectangle and using list slicing to remove and extract the integer. Figure 8 shows the images of the numbers after being extracted. Now these images of integers can be put into a machine learning model to classify and finally convert to an array.

## Machine Learning and Classification

The final step is to convert the images of integers found into the corresponding integer, which can be done through machine learning. As described in [11] for number classification a CNN (convoluted neural network) performs the best, therefore I will use A CNN on the MNIST handwritten dataset.

# Setting up the model

model = Sequential()

# Adding a convolutional layer

model.add(Conv2D(32, kernel\_size=(3, 3), activation='relu', input\_shape=(28, 28, 1)))

# Adding a max-pooling layer

model.add(MaxPooling2D(pool\_size=(2, 2)))

# Flatten the output to get it in one dimension

model.add(Flatten())

# Adding dense layers

model.add(Dense(512, activation='relu', input\_shape=(28, 28, 1)))

model.add(Dense(10, activation='softmax'))

The above code snippet is my neural network, which is a sequential model that adds each layer one by one. The first layer is the Conv2D, which uses 32 filters to scan the images and extract specific features, such as edges and shapes. The second layer is the MaxPooling2D is used to make the image smaller to discard irrelevant information, for every 2 by 2 block I will replace the block with the max value. The third layer flattens the image into a 1D image, which is a necessary input to the next layer. The final 2 layers are dense, and their role is to determine the probability of the image being each number, the integer with the highest probability is the classification. This model works very well as it gave me a test accuracy rate of 98.64% which is good enough for this purpose.

Finally, I can use the model to convert the image to their corresponding integers, I loop through the digit\_cells array and if an index has a -1, then it doesn't contain an integer, so I place a 0 otherwise I pass the image to the CNN and replace the image with its prediction. With this, I now have the final array, which contains the translation of the original image containing a puzzle to a 2D array that represents it.

## Killer Sudoku Machine Vision

With Machine vision for killer Sudoku, the same steps apply up to and including the cell extraction. However, the next steps differ because of how the puzzle is laid out. Figure 9 shows the extracted cell from the Killer Sudoku. The next step is to determine which sides the cages are on and the cage sums.

A black square with a number

Description automatically generatedA black and white square with dots

Description automatically generatedA black and white square with a number

Description automatically generated

Figure 11 – Extracted cells from killer Sudoku.

The first part is to get the cage sums. This works largely the same as the Sudoku, but instead of the integers being in the centre, they are on the top left. Therefore, we can extract the top left of the cell and extract the digits as before. The other difference is that instead of single digits, there can be multiple digits. However, this does not change much.

*if* w \* h > 80 and w \* h < 1500 and  h < 40 and w < 40:

*if* len(sums) == 0:

                    sums.append(canvas)

                    current\_x = x

*else*:

*if* x < current\_x:

                        sums.insert(0, canvas)

*else*:

                        sums.append(canvas)

Above is the code snippet for identifying integers, since there can be more than one integer, we get all the contours and check whether the area of the contours is within the specified bound and the width and height have a minimum value. If the condition is met, then the digit is stored to be classified.

To find the cages, we can once again use the contours to identify where the cages are located. Figure 9 shows the cages are represented by a dashed line and, using findContours in the OpenCV library, each segment is a contour. Therefore, we can count where all the contours are located, and that is where the cages are located .

*if* center\_x < margin and x > 0:

                sides[0] += 1

*elif* center\_x > 110 - margin and x + w < 110:

                sides[2] += 1

*if* center\_y < margin and y > 0:

                sides[3] += 1

*elif* center\_y > 110 - margin and y + h < 110:

                sides[1] += 1

This code snippet is used to count all the contours and where they appear, the margin is simply 20 pixels from the edges and represents the side of the cell. After counting the contours, if any side is greater than 5, then we can consider it a side otherwise it is not. I chose 5 due to potential noise and, in most cases; the side has 8 contours, and hence a reasonable value. I can then arrange each cell as an array that contains whether it contains a sum, which side has cages 1 means has a cage on that side and 0 means it does not, and finally a checked variable.

    def constructCage(self, i, j):

        cageCells = [(i, j)]

        cell = *self*.cages[i][j]

*if* cell[5] == 1:

*return* []

*self*.cages[i][j][5] = 1

*if* cell[2] == 0:

            cageCells = cageCells + *self*.constructCage(i+1, j)

*if* cell[3] == 0:

            cageCells = cageCells + *self*.constructCage(i, j+1)

*if* cell[1] == 0:

            cageCells = cageCells + *self*.constructCage(i, j-1)

*return* cageCells

The above code snippet is a recursive algorithm that is called for all unchecked cells. If it is not checked, it navigates through the cell and calls itself with the next open cell, if there is one. If there's no open cell, then we have explored all open cells and so found all cells in the cage. At the end, each call will return a list of all the cells which are in the same cells. Finally, we have the grid and the cages with cage sums so, it can be passed to the killer Sudoku solver to be solved.

## Conclusion

In this section, I will summarise all the sub sections needed to implement machine vision for Sudoku and killer Sudoku. The first step is to extract the grid from the image. To do this, we assume that the puzzle is the second biggest contours within the image and then extract that contour from the image. The next step is to straighten the image to not be tilted, which would affect the following steps. To straighten the image, we get the 4 corner points of the puzzle and then stretch or shrink the image to fit within a square canvas. Now, the canvas can be divided into 9 by 9 chunks, where each chunk will contain a singular cell from the puzzle. For Sudoku, each cell is processed to extract the digit in the cell if there is one and then sent into a machine learning model to get the digit contained in the cell. For killer Sudoku, we also must account for the cages and cage sums in a cell. Since the cages are represented as dashed lines, each part of the dashed line is a contour, thus we can count the contours and on which sides they appear to create the cages. As for the cage sums, this process is like Sudoku, the section containing the cage sum is processed to extract each individual digit and then sent into the machine learning model to be classified. These are the steps which I have implemented for machine learning and while there may be improvements that can be made, I am happy with its accuracy for now.

# Puzzle Generation

## Sudoku Generation

In this section, I will describe my implementation of the Sudoku puzzle generation of different difficulties. The algorithm used is inspired by Wei Meng in the book "Programming Sudoku" [3]. The first step to generating a Sudoku puzzle is to generate a nine-by-nine array, which will be the solution to the puzzle. One way to do this step would be to adapt my Sudoku solver to use the constraints to generate the grid and use a randomness property to generate a unique puzzle each time. However, an easier way to do this is described in [23] which tries to fill the top left, middle and bottom right boxes with the number 1 to 9. After the three boxes are filled, we can then use the Sudoku solver to complete the rest of the grid. This technique works is because the initial three boxes which are filled do not interact with each other. Therefore, no matter which way the boxes are filled, there will always be a solution to that puzzle.

        options = [1,2,3,4,5,6,7,8,9]

        random.shuffle(options)

*# getting random values for top left, middle and bottom right boxes as they do not interact.*

*for* i *in* range(0, 3):

*for* j *in* range(0, 3):

*self*.grid[i][j] = options[(i\*3)+j]

        random.shuffle(options)

*for* i *in* range(3, 6):

*for* j *in* range(3, 6):

*self*.grid[i][j] = options[((i-3)\*3)+(j-3)]

        random.shuffle(options)

*for* i *in* range(6, 9):

*for* j *in* range(6, 9):

*self*.grid[i][j] = options[((i-6)\*3)+(j-6)]

        sudoku = Sudoku(*self*.grid)

*self*.sudokuSolver = SudokuSolver2(sudoku)

*self*.grid = *self*.sudokuSolver.solver()

The above code is the implementation first step, which first fills the three boxes using the random library. It then uses the Sudoku solver to complete the rest of the grid to get the solution to the puzzle.

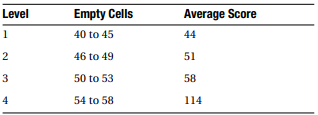


Figure 12 - Table showing the number for empty cells for each difficulty level based on average score [3].

The next step is to remove values from the grid one by one to leave a partially filled puzzle which will be presented to the user. However, the number of values to be removed will depend on the difficulty of the puzzle. Figure 10 shows the number of empty cells required for each difficulty level and this is how I have implemented my difficulty levels. The algorithm used for this feature, uses the random library to first choose the number of empty cells from the range depending on the difficulty and then again to generate random points on the grid and set it to 0.

*for* i *in* range(emptyCells):

*if* len(cells) == 0:

*self*.grid = copy.deepcopy(*self*.solution)

*return* False

            x, y = random.choice(cells)

            cells.remove((x,y))

            removed\_value = *self*.grid[x][y]

*self*.grid[x][y] = 0

In this code snippet, emptyCells in the number of cells to be removed, the for loop then randomly set a cell in the grid to 0. However, while this gives a random grid each time it does not satisfy an important property of a Sudoku puzzle which states that each puzzle must only have one solution. Therefore, to satisfy this property, I first adapted my recursive Sudoku solver to give me the number of solutions for a particular grid instead of the first solution. Now, each time that I attempt to remove a value, I test to see whether the current puzzle is unique. If it is unique, then I can move on to remove another cell, otherwise I put back the value and try another cell.

While testing this code, I found that the algorithm something entered an infinite loop because it could not find the next value to remove. This happened because the current puzzle reached a point at which no other value could be removed without adding an extra solution to the puzzle. Therefore, to fix this, I added a while loop which checks if we have tried to remove all cells. If it has and the number of empty cells is not reached, then the puzzle has reached its minimum state, and we can restart the algorithm until a puzzle is found with the correct number of empty cells.

Code snippet 1 in section 10.5 shows the final method for removing cells from the grid. The code first gets the number of empty cells the puzzle will have. Then it has a for loop and in each iteration, a cell is removed at random, each time the puzzle is checked to determine if it is valid or not. There is an also a while loop within the for loop, which is used to make sure the puzzle has not reached a minimum state where no more values can be removed. If we reach a point where all cells have been iterated through to be removed and we still have not reached the number of empty cells, then the minimum has been reached and we can start again.

## Killer Sudoku Generation

In this section, I will describe my implementation of the killer Sudoku puzzle generation. To begin, we first need to obtain the solution to the puzzle. To do this, we can use the same technique used in the Sudoku generation to get a completed grid. Since the killer Sudoku is empty to begin with, the initial puzzle presented to the user will just be a blank grid. The second part is to generate the cages around the cells.

To generate cages, I have implemented a method which can create cages for any cage lengths. Snippet 2 in section 10.5 shows the method for creating the cages. It is a recursive method, which tries to expand the current cage by one cell in each recursive call. The method takes as parameters the number of cells the cage should have and the starting point for the cage. If the current cell is not already in another cage it is added to the current cage and makes a recursive call to a neighbouring cell. It chooses the next cell to explore randomly, the reason for this is so we can generate different ![A white square with black border

Description automatically generated](data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAADsAAAC3CAYAAACsVyYQAAAAAXNSR0IArs4c6QAAAARnQU1BAACxjwv8YQUAAAAJcEhZcwAADsMAAA7DAcdvqGQAAAEHSURBVHhe7dvBDYQwEARB+8IiU+KCfDgeRFHbJVnab8vv2dd1PWuI/by+m3bf9/p99wjFqopVFasqVlWsqlhVsapiVcWqilUVqypWVayqWFWxqmJVxaqKVRWrKlZVrKpYVbGqYlXFqopVFasqVjVqnjbrZ9vioYpVFasqVlWsqlhVsapiVcWqilUVqypWVayqWFWxqmJVxaqKVRWrKlZVrKpYVbGqYlXFqopVFasqVlWsqi2eqi2eqlhVsapiVcWqilUVqypWVayqWFWxqmJVxaqKVRWrKlZVrKpYVbGqYlXFqopVFasqVlWsqlhVsapiVaNi9/tGjJjO85wTexzHrOFhWzzTWn/RwC2c7f95FQAAAABJRU5ErkJggg==)puzzle even within the same parameters.

A blue square with black lines

Description automatically generated

Figure 13 - A box with a cage missing.

If a particular path does not give the required number of cells, then that path is still used, and the remaining cell are obtained through other paths. If a particular path does not give the required number of cells, then that path is still used, and the remaining cell are obtained through other paths. Figure 11 shows why this is necessary, because if the method started with the first cell in the second row no matter what cell it went to next, the maximum cage length it obtained would be two. However, by adding even partial paths, the method can, for example, explore the above cell and the below to create a cage of length three. A required property of cages is that values within a cage cannot be repeated. Therefore, to implement this the method also keeps track of all the values already used and if the current cell's value has already appeared then it is not added to the cage.

One problem with the above method is that because the cell passed to the method is random, it will leave cells which are not in cages. Therefore, I have implemented a method which puts all uncaged cells into a cell.

    def fillRemainingCells(self, cells):

*while* len(cells) != 0:

*if* *self*.difficulty == "expert":

                triple = *self*.findConnectCells(3, cells[0], [])

*if* triple != -1 and len(triple) == 3:

                    cells = *self*.addCage(triple, cells)

*continue*

            pairs = *self*.findConnectCells(2, cells[0], [])

*if* pairs != -1 and len(pairs) == 2:

                cells = *self*.addCage(pairs, cells)

*else*:

                cells = *self*.addCage([cells[0]], cells)

The above method takes as parameters an array of all cells not already in a cage. It then tries to form cages of length two or one depending on if it has neighbouring cells which are also not caged. The method only creates cages of length three if the difficulty selected by the user is "expert".

    def addCage(self, cells, allCells):

        count = 0

*for* i *in* cells:

            count += *self*.grid[i[0]][i[1]]

*self*.grid[i[0]][i[1]] = -1

            allCells.remove((i[0], i[1]))

*self*.cages[*self*.count] = {count : cells}

The above code snippet is a helper method which takes an array of cells which form a cage. The method then stores the cells and the cage sum in a dictionary, which contains all cages. Now, with these three methods, we can generate the cages for a puzzle. The difficulty of the puzzle comes from the length of the cages, meaning puzzles with larger cages are tougher. To implement this, I have set a dictionary containing a range of different cage length for each difficulty level as shown below. It shows more difficult the puzzle is, the greater the average cage length is.

*self*.easy = {"3" : [5,6,7], "4" : [1,2,3], "5" : [0]}

*self*.medium = {"3" : [8,9], "4" : [2,3,4], "5" : [0]}

*self*.hard = {"3" : [6,7,8], "4" : [4,5], "5" : [1,2]}

*self*.expert = {"3" : [4,5], "4" : [3,4,5], "5" : [3,4]}

Finally, after the cages are created, the last step is to make sure that the puzzle only has one solution. Therefore, just as I did for the Sudoku solver, I adapted my killer Sudoku solver to return the number of solutions to the puzzle. If it returns one, then the puzzle is valid and can be returned to the user, otherwise, we need to restart the algorithm to generate a new puzzle.

# Solving Technique

When solving a Sudoku or killer Sudoku puzzle, players will often need to know some of the wide range of techniques available to complete the puzzle. In all the tougher puzzle, finding the answer to a particular cell within a puzzle is often not possible simply. Therefore, these techniques exist to reduce the possibilities for the cell and eventually deduce the answer to the puzzle. In this section, I will discuss my implementation of the techniques which are the most important and useful. It is also important to note that all the techniques used in Sudoku can be used for Killer Sudoku, but the opposite does not hold true.

## Obvious Singles/Pair/Triples

This technique can be spilt in three ways, but they all do the same thing. It works by checking systematically checking each row, column, and box for the same domains. For example, in obvious pairs, there is a case where two different cells contain the same domain with two choices within a box. Here, because these two cells only have two options, one of those options must be assigned to one cell and the other option to the other cell. Therefore, because of the rule that a number cannot be repeated with the box, we can eliminate those two options in the rest of the box.

A screenshot of a game

Description automatically generatedA screenshot of a game

Description automatically generated

Figure 14 - An example of the obvious pairs (left) and obvious triple(right). Red shows the number eliminated and the blue shows fixed values.

The left image in figure 12 is an example of the obvious pairs, because two cells contain exactly 7 and 9. Those two values can now be removed from all other cells in the same region. Obvious Singles works in the same way, but the cell should only contain one value, in this situation the values will be the answer for the cell. Regarding obvious triples, it works similarly, but it does not necessarily need to involve each cell having 3 values in its domain. As shown in figure 12 in the right image, the bottom 3 cells each contain 2 values in their domains. However, the set of those cells combined equals exactly 3 values. This means that each one of these three values will appear in one of the three cells. Hence, those three values cannot appear elsewhere in the box and can be removed.

In section 12.5.3, I have listed my code for applying the obvious pair rule within a row. The code starts with iterating through all cells within the row. If any cell contains exactly two values in its domain, then it is remembered. After that point, if any other cell also has those two values in its domain exactly, then the rule is triggered. When the rule is triggered, all cells in the row are iterated through again and the two values are removed from all domains apart from the two remembered cells.

## Hidden Single/Pair/Triple

The hidden groups of techniques work similarly to the obvious counterparts, but they do not need necessarily need to look at the entire domain. Instead, they are concerned with the number of times a number occurs within the selected region.

A screenshot of a number game

Description automatically generatedA screenshot of a game

Description automatically generated

Figure 15 - The hidden single rule (left) and the hidden pair rule (right).

In the left image in figure 13, an example of the hidden single is shown. The image show that the number 1 only appears once in all the domains, but because the numbers 1-9 must appear and 1 only appears once, that cell must be 1. Therefore, in the top right cell, the other domain values can be ignored and 1 is assigned to the value. Hidden pairs work in the exact same way but instead of one value, it seeks 2. The image on the right in figure 13 shows the hidden pair rule. The number two and six only appear twice in the entire box, hence both these values must appear in these two cells. Hence, we can ignore all other domain values in these two cells apart from two and six. Finally, hidden triples use the same logic as the hidden pairs, but instead of finding 2 values, it tries to find 3.

The code snippet in section 12.5.4 shows the code implementation of the hidden pairs in a column. The code starts by looping through all values between 1 and 9 and for each value; it loops through all the domains within the column. For each value, if it appears exactly twice, then the two cells, and the value is stored in a dictionary. The program then proceeds and if it finds another value which is only shared between the two initial cells, then the hidden pair rule is triggered. When the rule is triggered, it just assigns those two values to the domain of the two cells and all other values are removed.

## Pointing Cells

A screenshot of a calendar

Description automatically generatedPointing cells work by relying on the rule that a number can only appear one in a row, column, or box. Pointing cells can only be used in a box, unlike the previous two techniques, and instead of reducing the domains within its box, it will reduce domains outside.

Figure 16 - Shows the application of the pointing cell rule.

Figure 14 shows the application of the pointing cells rule, as it shows that the values 1 only appears in the first row in the right box. This means that one will have to appear in one of those three cells and cannot appear elsewhere in the row. Therefore, we can now remove one from the rest of the row, leaving a reduced domain. This technique can also be applied column wise, where instead of looking at row, we check if a value only appears in one column and then reduce that value for the rest of the column values. In addition, the rule can also be applied with only two values, for example in figure 14, if only 2 ones appeared in the right top row, then the rule still works.

In Section 12.5.5, I have inserted my code implementation of the pointing cells rule for rows. The method starts off by iterating through the numbers 1-9. For each number it will loop through all each cell in the square, if a value is found in a cell, then that cells row index is added to the “occur” array. After iterating through each cell, if the set of occur only contains one number it means that the number, we are checking for only appears in one row and we can apply the rule. To apply the rule, we can remove the number we have found from the rest of the row, outside the box we have checked.

## Rule of K

A grid of squares with numbers

Description automatically generatedA grid of squares with numbers

Description automatically generatedThe rule of K techniques only applies to Killer Sudoku puzzles as it relies on using the setup of cages to reduce the domain options. It also relies on the knowledge that each row, column and box must add up to the sum of 45.

Figure 17 - An example of the rule of K.

Figure 15 shows an example of how the rule of K can be applied within a row. In the first column we can see that four of the five cages are exactly within the column and only the 14 cages at the bottom does not fit. Since each number must appear in every column, the sum of any column must sum up to 45. If we add up the sum of the cages which fit within the column, we get 15 + 9 + 9 +7 which gives a total of 40, this means in these four cages the combinations of its values must add to 40. This shows we have 5 remaining but since that 5 can only be distributed amongst one other cell, that cell must contain a 5. Then we can get the answer for the second cell by simply doing 14 – 5 which gives 9. This technique can also be applied within boxes and row and can span across multiple rows and columns. If the technique is used for multiple rows, then the total sum also changes, for example if we are looking at 2 rows then the total for those two rows should be 90. After this the same logic applies as before.

Section 12.5.6 contains my implementation of the rule of K in the case of any number of rows. Since the code is quite complicated, I will explain it briefly. It starts with finding out the sum of the rows and then iterates through every cell in all the rows its given, for each cell, its cage is checked to see if all its cells are contained within the chosen rows. If it is then we can subtract its cage sum from the total. If not, we then store the cage number and carry on. At any point if we find another cage which does not fit within the region, then the method returns false. However, if only one cage if found at the end then the rule is triggered. When triggered we can then reduce the domain using domain reduction. While the method contains more complicated code, they cover some more cases it which this method can also be applied.

## Domain Reduction

Domain reduction is a technique which is used to make sure the domains for all cells are kept as accurate as possible, so more of the other techniques can be applied. It relies on the number of cells, the sum the cells should and up to and their domains. As an example, consider 2 cells with the domains {1,2,4,6} and {2,3,5,6} with a sum of 7. From this we can see that the only combinations which add up to 7 are (1,6), (2,5), (4,3) however if we look closely, we can see that the number 6 is never used from the first domain and 2 is never used from the second domain. Therefore, we can remove these two values and get a reduced domain of {1,2,4} and {3,5,6}.

    def fit(self, tempDomain, idx, used, length, total):

*# combination found so return solution*

*if* idx == length and total == 0:

*return* [list(used)]

*# no solution for current values so backtrack*

*if* idx == length or len(tempDomain[idx]) == 0 or total < 0:

*return* None

        allCombos = []

*# tries all values in the domains*

*for* num *in* tempDomain[idx]:

*if* num not in used:

                used.append(num)

                combo = *self*.fit(tempDomain, idx + 1, used, length, total

[24]- num)

*if* combo is not None:

                    allCombos.extend(combo)

*# remove last added value*

                used.pop()

*return* allCombos

The above code is the recursive method which systematically tries to set all values to the cell and returns an array containing all the combinations of the domains which give us the correct sum. In each combination the index matches the index of the cell therefore all the first indexes belong to the first cell, the second to the second cell and so on.

## Conclusion

To conclude, we have discussed the different range of techniques I have implemented to give the user better hints when they are stuck on a puzzle instead of just the answer. This will help users gain a better understanding of how to solve puzzles and improve on their skills. However, due to the complicated nature of Sudokus and Killer Sudokus, there are many more known techniques which I could have implemented, but due to time constraints I have only decided to implement the most used ones. In general, these Sudoku and Killer Sudoku technique are all that’s necessary to complete any valid puzzles therefore it’s important to make the user aware of these methods.

# Software Engineering

## Testing and Documentation

One of the most important parts of software engineering is testing, and it is something I paid a lot of attention to. TTD (Test Driven Development) was used for most of the backend code excluding the machine vision code. The machine vision code cannot be tested through automated tests as they are handling image processing, therefore I performed manual testing to make sure that it all worked. I have also performed some integration testing to make sure that the methods work as intended when run sequentially. As for the front end, I have used selenium to perform tests. I have used Selenium because it works well with plain HTML, JavaScript, and CSS. While it's slow to run, it can still test all aspects of the frontend such as the HTML elements, CSS, and even the backend interactions. Therefore, I have used it to test that all elements are displayed correctly as well as testing that the JavaScript code is interacting with the backend as expected.

I have also made sure to plan out the general idea of the application through UML diagrams. Currently, I have created the UML diagrams for the main functionality I have implemented which is for the solver and the machine vision. It includes the class names, methods, class variables, and how the different classes are related. I have added the link to the UML diagrams in chapter 9.4 which is on my GitLab account.

I used GitLab to keep my project centralised and to avoid losing access to any of my work. I made sure to use feature branches to develop individual features and keep different development lines separate from each other. By using feature branches, the main branch can be kept clean and always in a working state. I made sure to push all work to the repository and added a commit that provided essential information about the commit, such as what changes were made and why.

To improve the readability of my code, I have made sure to include appropriate documentation. I have used Pydocs to explain the role of methods and classes and their parameters and return items. In addition, comments have been placed throughout the application to increase the understanding of what the code is doing, especially in complex methods.

## Methodologies and Design Patterns

For this project, the methodology I used for development was the waterfall approach. This approach is concentrates on following a sequential and linear process. Initially, I broke down the project into smaller chucks, which needed to be implemented. Some of these sections were the Sudoku solver, Killer Sudoku solver, machine vision. The waterfall methodology requires each subsection to be completed in its entirety until moving on to the next task. Since I have implemented the website, to consider a task complete, the backend, frontend and testing needed to be complete. Despite this approach being restrictive, I believe this was the right approach for me. This is because before I started to work on the project, my requirements were clearly defined and stable. This meant that it's unlikely that there were going to be any changes to the requirements, which meant that this approach was ideal.

Since I am using Django to connect the frontend to the backend, the MVT design pattern is automatically incorporated. This means that the model, view, and template (web pages) are designed separately, the user interacts with the frontend and when they need some processing on data; the controller directs the request to the appropriate backend API, which sends back a response. This means that each aspect can be developed independently without interfering with the other aspects. Another design pattern used is the behavioural pattern state. This pattern alters the behaviour of an object as if it has changed its class. This pattern was used in the upload webpage because depending on which button the user clicked on (Sudoku or killer Sudoku) the underlying class used changes. The final design pattern used is the structural pattern Façade. This pattern provides a simple interface to the user and hides away the details of the more complex methods needed. This pattern is implemented in the SudokuExtraction and KillerSudokuExtraction class because it provides just one method to the user which handles all the sub operations needed to convert the image to a solver readable form. These patterns make the code more readable and decouple the classes from each other, thus improving the maintainability of the code.

# Professional Issues

# Conclusion

## Evaluation

The main objective of this project was to create a solver that could solve Sudoku and killer Sudoku puzzles using AI. As described in Chapter 3, I have mostly accomplished this, as the solver can give correct solutions for any puzzle, and with Sudoku and the easier killer Sudoku puzzle, it can find solutions relatively quickly. However, there are sufficient improvements I need to make in the future. I have learned about constraint programming and how it can be optimised to solve various types of problems efficiently, such as forward checking, arc consistency, and more. However, my solvers can still be improved. One way is to create a new solver that does not use recursion as much. This is because recursive algorithms are slow, especially as shown in the running time for the killer Sudoku for harder puzzles. One other requirement I had for the solver was to give hints as to how a solution was reached. The current solvers do not do this. Therefore, I will implement human-solving Sudoku and killer Sudoku algorithms in the new solver to logically find solutions, which will rely less on a recursive algorithm.

The next objective I stated, was to implement a machine vision model which would allow users to enter puzzles through images and return the solutions. I have described in Chapter 5 my implementation for this, and it works well. However, the solution relies on a few assumptions to be made, which unfortunately is unavoidable. The algorithm successfully extracts puzzles from the images and converts them to a human-readable form. One area the algorithm needs to improve in is the digit algorithm. While the machine learning algorithm has a success rate of 98% on the test data, it seems to struggle with classifying my data. This causes a lot of misclassification and therefore requires the user to change the number manually more often. In the second semester, I will aim to find the solution to this problem and reduce the error rate of my algorithm.

# Bibliography

|  |  |
| --- | --- |
| [1] | Pelánek, Radek, *Human problem solving: Sudoku case study.,* Faculty of Informatics, Masaryk University, 2011. |
| [2] | Simonis, Helmut, *Sudoku as a constraint problem,* CP Workshop on modeling and reformulating Constraint Satisfaction Problems. Vol. 12. Citeseer,, 2005. |
| [3] | W.-M. Lee, “Basic Sudoku Solving Techniques / Intermediate Techniques / Advanced Techniques,” in *Programming Sudoku*, Apress, 2006, pp. 47-171. |
| [4] | “Tips,” Killer Sudoku, [Online]. Available: https://www.killersudokuonline.com/tips.html. [Accessed 24 9 2023]. |
| [5] | Brailsford, Sally C., Chris N. Potts, and Barbara M. Smith, “Constraint satisfaction problems: Algorithms and applications.,” *European journal of operational research,* vol. 119.3, pp. 557-581, 1999. |
| [6] | R. Haralick and G. Elliot, “Increasing tree search efficiency for constraint satisfaction problems.,” in *Artificial Intelligence*, 1980, pp. 14:263-313. |
| [7] | Musliu, Nysret, and Felix Winter, “A hybrid approach for the sudoku problem: Using constraint programming in iterated local search.,” in *IEEE Intelligent Systems 32.2*, 2017. |
| [8] | Bittner, Benjamin, and Kris Oosting, “IMPLEMENTING A CSP SOLVER FOR SUDOKU,” Amsterdam. |
| [9] | Merugu Suresh, Azhar Talha Syed, “Augmented Reality on Sudoku Puzzle Using Computer Vision and Deep Learning,” in *Advances in Cybernetics, Cognition, and Machine Learning for Communication Technologies* , 2020, pp. 567-578. |
| [10] | K. Gupta, S. Khatri and M. H. Khan, “A Novel Automated Solver for Sudoku Images,” in *2019 IEEE 5th International Conference for Convergence in Technology (I2CT)*, Bombay, India, 2019. |
| [11] | Ritik D, Rishika K, Samay P, “Handwritten Digit Recognition using Machine and Deep Learning Algorithms,” 2021. |
| [12] | Čakić, Stevan, et al., “The use of tesseract ocr number recognition for food tracking and tracing.,” in *24th International Conference on Information Technology (IT). IEEE*, 2020. |
| [13] | Team #2306, “Generating Sudoku Puzzles as an Inverse Problem,” University of Washington, Washington, 2008. |
| [14] | “Sudoku Generator Algorithm,” 101 Computing, 21 3 2019. [Online]. Available: https://www.101computing.net/sudoku-generator-algorithm/. [Accessed 2023 11 08]. |
| [15] | Sudha, V., and R. Kalaiselvi, “The Sudoku Puzzle Generator Using DNA Computing.,” in *International Conference on Computational Intelligence, Security and Internet of Things. Cham: Springer International Publishing*, 2020. |
| [16] | Bertram Felgenhauer and Frazer Jarvis., “Enumerating possible Sudoku grids,” University of Sheffield, 2005. |
| [17] | Bessiere, Christian., “Arc-consistency in dynamic constraint satisfaction problems.,” in *AAAI Conference on Artificial Intelligence*, Anaheim, 1991. |
| [18] | Dechter, Rina., “Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition,” *Artificial Intelligence,* pp. 273-312, 1990. |
| [19] | Cox, James L., Stephen Lucci, and Tayfun Pay, “Effects of Dynamic Variable-Value Ordering Heuristics on the Search Space of Sudoku Modeled as a Constraint Satisfaction Problem.,” in *Inteligencia Artificial 22.63*, 2019. |
| [20] | “Heap queue algorithm,” Python, [Online]. Available: https://docs.python.org/2/library/heapq.html#priority-queue-implementation-notes. [Accessed 13 10 2023]. |
| [21] | Davies, E. Roy, Machine vision: theory, algorithms, practicalities, Elsevier, 2004. |
| [22] | Wikipedia, “Ramer–Douglas–Peucker algorithm,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Ramer%E2%80%93Douglas%E2%80%93Peucker\_algorithm. [Accessed 06 11 2023]. |
| [23] | Kush, “Building a Sudoku Solver and Generator in Python (2/3),” Medium, 2 4 2021. [Online]. Available: https://medium.com/codex/building-a-sudoku-solver-and-generator-in-python-2-3-5eef3141702c. [Accessed 02 1 2024]. |
| [24] | “Sudoku rules,” Easybrain, [Online]. Available: https://sudoku.com/sudoku-rules/. [Accessed 2024 01 26]. |
| [25] | “Simple Killer Sudoku solving techniques,” Sudoku Solver , [Online]. Available: https://sudokusolver.com/play/killer/45/#:~:text=The%2045%20Rule,block%20will%20always%20be%2045.. [Accessed 26 01 2024]. |
| [26] | “Killer Sudoku,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Killer\_sudoku. [Accessed 2 01 2024]. |

# Appendix

## Diary

Week 1 - 28/09/2023:

This week I have design and implemented the interface for the Sudoku page on the website. It contains the sudoku grid which is currently hardcoded and buttons for the numbers which is used to fill the grid. I have also done some research on constraint solvers and the basics of machine vision.

Next, I will complete the web page for killer Sudoku and then start work on building a constraint solver for Sudoku.

Week 2 - 03/10/2023:

This week I have completed the web design for the killer sudoku page on the website. Now both sudoku and killer sudoku can be played. I have added some more styling to the web page to its nicer to look at. In addition, I have set up some classes for sudoku and killer Sudoku so the grids and cages can now be stored in objects.

Next, I will start work on the constraint solver for both types of puzzles using backtracking.

Week 2 - 07/10/2023:

Since the last update I have implemented a simple solver for both sudoku and Killer sudoku which using backtracking and node consistency to reduce the domain space for the cells.

Next, I will implement value order heuristic to both the solvers and see if and how they improve the solvers and then start working on the report for these sections.

Week 3 - 12/1-/2023:

This week I have implemented value order heuristics for both the solvers by created a priority queue and choose the next cell to guess based on which cell has the smallest domain space.

Next week I will write the report for the constraint solver section and being work on machine vision for sudoku.

Week 4 - 17/10/2023:

This week I have finished writing the constraint solver section with the different algorithms used within it in my report as well as my how I implemented the sudoku solver. In addition, I have started the section on the killer sudoku solver.

Next, I will start working on the machine vision and complete the report for killer sudoku.

Week 4 - 22/10/2023:

This week I have begun working on the machine vision for killer Sudoku. I have implemented code which isolates the puzzle within an image and then extract each cell within the puzzle. I have also finished the section in my report on the killer sudoku solver and on the data structures I have used to create the solvers.

Next, I will work on a machine learning model which will classify images containing numbers.

Week 5 - 29/10/2023:

This week I have finished the implementation of the machine vision including the machine learning code and I have started work on integrating the machine vision code in the frontend.

Next, I will work on the frontend to integrate the machine vision into the frontend and write a section in the report for machine vision and the frontend design.

Week 6 - 02/11/2023:

This week I have integrating the machine vision code into the frontend. I have also started to test the frontend using selenium. So far, I have fully tested the current play sudoku and the home pages. I have partially tested the killer sudoku page.

Next, I will work on finishing of the frontend testing, work on adding a section on machine vision and try to fix the machine vision code to give better results.

Week 7 - 08/11/2023:

This week I have been working on the report to get it ready for the submission. I have added a section on machine vision, software engineering, introduction, and conclusion. However, there are a few sections which need to be improved.

Next, I will work on adding all my references and generally getting the report in a good state.

Week 8 - 14/11/23:

This week I worked on finishing off the report to add the final sections. I have also created a new model for my digit classification code which has improved the accuracy of the machine vision.

Next, I will work on applying these changes for the killer sudoku machine vision. I will also try to improve my solver for the killer sudoku by trying a different way to get the variable domains.

Week 9 - 20/11/23

This week I worked on improving my killer sudoku machine vision, as it was struggling my obtaining the correct cages and the cage sums. I have made the algorithm more flexible at identifying these things and incorporated the new machine learning model. Through testing I have seen the model's accuracy improve a lot however its struggling to identify the number 1.

Next, I will work on trying to improve the recursive algorithm for the killer Sudoku, as it currently takes too long.

Week 10 - 29/11/23

This week I have been working on getting my project in a good state to prepare for the interim submission. I have been preparing for the presentation and recorded the demo video as well. In the report I have added an appendix which contains the user manual, links to my UML diagrams and demo video.

Next, I will work on trying to add reasoning techniques into my solver to provide the user with better hints.

## User manual

### Specification and how to run.

The project is developed using Python 3.9.1, so please make sure that is the version you are running the code with. To run the application, first make sure to install all the packages required for the project. The requirements.txt file contains all the required packages. Use this command "pip install -r requirements". Once the environment is established, to run the website, first make sure you are in the top level FinalYearProject directory, then run the command "py manage.py runserver". Now to view the website, go to this link "http://127.0.0.1:8000/". To run the TTD tests, use this command "py -m unittest discover -s "./test" -p "test\*.py"". Finally, to run the frontend testing, use this command py manage.py tests, note to run the frontend testing the website should already be running first.

### How the website works

The play Sudoku and play killer Sudoku pages work the same way, they allow users to select cells, place numbers by clicking the numbers below the puzzle. They also have the features to undo any actions made in the puzzles, give hints for the selected puzzle, and solve the entire puzzle. The upload page needs some necessary actions to get the best performance. First, both puzzles require an image to be uploaded, which needs to be of a specific form. For examples the MachineVisionImage directory contains images which work well. To be specific, the puzzle within the image must take up most of the image and there must not be any other box which is greater than the puzzle. After processing the image, you will be displayed with the puzzle the algorithm has found. You can then manually correct any errors made by it.

## Video Link

This is the link to the video demonstration of my project: <https://youtu.be/-IoJFmg8xu0>

## UML diagrams links

This the link to my GitLab account: <https://gitlab.cim.rhul.ac.uk/zkac166/PROJECT>. The document UML diagrams.png is where the diagrams are.

## Code snippets

### Snippet 1

    def removeNumbers(self, difficulty):

*'''*

*Given a range of cells to be removed, It generates a unique Sudoku*

*puzzle.*

*parameters:*

*difficulty: A 2D array containing the number of cells to be removed.*

*'''*

        random.shuffle(difficulty)

        emptyCells = difficulty[0]

        cells = [(i, j) *for* i *in* range(9) *for* j *in* range(9)]

*for* i *in* range(emptyCells):

*while* True:

*# if the current grid cannot be reduced any further then*

*a reset is needed.*

*if* len(cells) == 0:

*self*.grid = copy.deepcopy(*self*.solution)

*return* False

                x, y = random.choice(cells)

                cells.remove((x,y))

                removed\_value = *self*.grid[x][y]

*self*.grid[x][y] = 0

*# checking for unique solution after each removal*

                SolutionFinder = SudokuSolver2(Sudoku(

copy.deepcopy(*self*.grid)))

                solutions = SolutionFinder.SolutionFinder()

*if* solutions == 1:

*break*

*else*:

*# if not a unique solution, revert the change*

*self*.grid[x][y] = removed\_value

*return* True

### Snippet 2

    def findConnectCells(self, num, cell, visited):

*'''*

*Tries to find cells to form a cage of specific lengths.*

*parameters:*

*num - the lwngth of the cage*

*cell - the starting cell*

*visited - an empty array.*

*returns:*

*a set of cells or -1*

*'''*

*if* num == 0:

*return* set()

*if* cell[0] < 0 or cell[0] > 8 or cell[1] < 0 or cell[1] > 8 or

*self*.grid[cell[0]][cell[1]] == -1 or

*self*.grid[cell[0]][cell[1]] in visited:

*return* -1

        current\_cells = set([cell])

        visited.append(*self*.grid[cell[0]][cell[1]])

*# creating cages using randomness to avoid similar puzzles.*

        nextCell = random.choice([0, 1, 2, 3])

        response = None

*for* i *in* range(4):

*if* nextCell == 0:

                response =  *self*.findConnectCells(num-1, (cell[0]+1,

cell[1]), visited)

*if* nextCell == 1:

                response = *self*.findConnectCells(num-1, (cell[0],

cell[1]+1), visited)

*if* nextCell == 2:

                response = *self*.findConnectCells(num-1, (cell[0]-1,

cell[1]), visited)

*if* nextCell == 3:

                response = *self*.findConnectCells(num-1, (cell[0], cell[1]-

1), visited)

*if* response == -1:

                nextCell = (nextCell + 1) % 4

*elif* len(response) != num - 1:

                num = num - len(response)

                current\_cells = current\_cells.union(response)

                nextCell = (nextCell + 1) % 4

*else*:

*break*

*if* response != -1:

*return* current\_cells.union(response)

*else*:

*return* current\_cells

### Obvious pair in a row Code

    def checkPairsInRow(self, row, domain):

        singleCells = []

*for* i *in* range(9):

*if* domain[row][i] != -1 and len(domain[row][i]) == 2:

*if* domain[row][i] in singleCells:

                    cells = []

*for* j *in* range(9):

*if* domain[row][j] != -1 and domain[row][j] !=

domain[row][i]:

                            domain[row][j] = domain[row][j] - domain[row][i]

*if* domain[row][j] != -1 and domain[row][j] ==

domain[row][i]:

                            cells.append((row,j))

                    message = f"{domain[row][i]} are obvious pairs row

{row} in cells {cells[0], cells[1]}"

*return* message, domain

*else*:

                    singleCells.append(domain[row][i])

*return* None, None

### Hidden pair in column

    def hiddenPairColumn(self, col, domain):

        double = {}

*for* num *in* range(1, 10):

            at = []

*for* i *in* range(0, 9):

*if* domain[i][col] != -1 and num in domain[i][col]:

                    at.append((i, col))

*if* len(at) == 2:

*if* tuple(at) in double:

*for* a *in* at:

                        domain[a[0]][a[1]] = {double[tuple(at)], num}

                    message = f"{double[tuple(at)], num} are hidden pairs

in column {col} in cells {at[0], at[1]}"

*return* message, domain

*else*:

                    double[tuple(at)] = num

*return* None, None

### Pointing cells in row code

    def pointingCellsRow(self, row, col, domain):

*for* num *in* range(1, 10):

            occur = []

*for* i *in* range(row\*3, (row\*3)+3):

*for* j *in* range(col\*3, (col\*3)+3):

*if* domain[i][j] != -1 and num in domain[i][j]:

                        occur.append(i)

*if* len(set(occur)) == 1:

*for* i *in* range(9):

*if* domain[occur[0]][i] != -1 and not (i >= col\*3 and i < (col\*3) + 3):

                        domain[occur[0]][i].discard(num)

                message = f"{num} is a pointing cell in row {list(set(occur))}"

*return* message, domain

*return* None, None

### Rule of K for multiple rows

    def checkRuleOfKRow(self, row1, row2, domain):

        total = 45 \* ((row2 - row1) + 1)

        uncontainedCageNum = -1

        uncontained = []

        contained = set()

*for* row *in* range(row1, row2+1):

*for* i *in* range(0, 9):

                cageNum = *self*.cageLayout[row][i]

*if* *self*.killerSudoku.grid[row][i] != 0:

                    total -= *self*.killerSudoku.grid[row][i]

*continue*

*if* cageNum in contained:

*continue*

*if* cageNum == uncontainedCageNum:

                    uncontained.append((row,i))

*continue*

                cage = *self*.killerSudoku.cages[cageNum]

*for* cageSum, cells *in* cage.items():

                    width = set()

                    inAreaFilled = []

                    outArea = 0

                    outFilledArea = 0

                    outAmount = 0

*for* cell *in* cells:

*if* not (cell[0] >= row1 and cell[0] <= row2):

                            width.add(cell[0])

*if* *self*.killerSudoku.grid[cell[0]][cell[1]] != 0

and (cell[0] >= row1 and cell[0] <= row2):

                            total += *self*.killerSudoku.grid[cell[0]][cell[1]]

                            inAreaFilled.append(cell)

*if* not (cell[0] < row1 and cell[0] > row2):

                            outArea += 1

*if* *self*.killerSudoku.grid[cell[0]][cell[1]]

!= 0:

                                outFilledArea += 1

                                outAmount += *self*.killerSudoku.

grid[cell[0]][cell[1]]

*if* outArea != 0 and outArea == outFilledArea:

                        total -= cageSum - outAmount

                        contained.add(cageNum)

*elif* len(width) == 0:

                        total -= cageSum

                        contained.add(cageNum)

*else*:

                        uncontained += inAreaFilled

*if* uncontainedCageNum != -1:

*return* None, None

                        uncontained.append((row,i))

                        uncontainedCageNum = cageNum

*if* len(uncontained) == 0 or total == 0:

*return* None, None

        message = "rule of k in row", row

*return* *self*.reduceDomains(uncontained, uncontainedCageNum, total, domain), message