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

October 22, 2023

**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 Where other published and unpublished source materials have been used, these have been acknowledged.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:

Student Name: Manfred Sandy Alex Araujo

Date of Submission:

Signature:

Table of Contents

[Abstract 3](#_Toc148107407)

[Milestones 5](#_Toc148107408)

[Timeline 6](#_Toc148107409)

[Risks and Mitigations 8](#_Toc148107410)

[Chapter 1: Constraint solvers 11](#_Toc148107411)

[1.1 Introduction 11](#_Toc148107412)

[Chapter 2: Sudoku Solver 12](#_Toc148107413)

[2.1 Introduction 12](#_Toc148107414)

[2.2 First Sudoku Solver 12](#_Toc148107415)

[2.3 Improving the Solver 14](#_Toc148107416)

[Bibliography 15](#_Toc148107417)

Abstract

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 [3] shows that the mean time to solve a sudoku puzzle is between 8-23 minutes. This study was conducted using two web sources, the first was used by more expert puzzle solvers, and the second by a more varied skill range hence the difference in solving time. However, Helmut Simonis [4] describes sudoku as a constraint satisfaction problem using different propagation techniques to solve them within a few milliseconds. People are likely to find a sudoku or killer sudoku puzzle in places such as a newspaper or a magazine, where getting stuck on a solution usually means waiting until the next day to receive the answer. However, by using a solver, the puzzle could be solved in a few seconds and be shown to the user either directly or provide hints so the user can still get the satisfaction of solving the puzzle on their own. Along with the solver, using machine vision to upload puzzles and avoid the tedious nature of entering values manually is a strong motivation for working on this project. Additionally, the application will also have the feature to allow users to solve puzzles directly on the app meaning all the features will be in one convenient place.

Solving a Sudoku or Killer Sudoku puzzle involves using logic to fill cells as well as to remove possibilities from other cells. Both these types of puzzles use different deduction techniques such as Lone Ranger, elimination techniques, looking for twins and triplets in Sudoku as described by Lee Wei-Meng [1], and sum elimination, rule of k, and rule of necessity in killer Sudoku as described on killer sudoku online [2]. The techniques discussed in these sources are very important and they are essential to completing a Sudoku or killer Sudoku puzzle. However, the difficulty lies in the ability to spot where they can be used, this is where a solver can thrive and complete these types of puzzles incredibly quickly.

Since the rules of the game are so clear, the puzzle itself can be defined as a CSP (Constraint Satisfaction Problem). A CSP is a type of programming paradigm where the problem is represented as a model with the variables to be optimized, which in this case is each cell in the 9 by 9 2D array, the domain is defined which is between 1 to 9, and the constraint of the variable is that each number must appear only once in each 3 by 3 square, once in each column and once in each row. Constraint Solver tries to find a solution by assigning variables with values from their domain which satisfies all the constraints defined. If a value leads to a dead end, then backtracking is used to go back as described in [5]. One way to improve this approach would carefully choose which variable to try next, for example, using a priority queue to select a variable with the smallest number of possible domain values. This means that the search will fail earlier if the guess is not correct thus improving the speed at which a solution is found [6]. The main benefit of this approach is that it always guarantees that a solution is found as it tries all the valid numbers for each empty cell. Therefore, I am going to use a combination of backtracking, node consistency along with forward checking to reduce the search.

The difficulty of a puzzle can be measured in different ways such as the number of blank squares; if there are more empty squares the puzzle will be harder to solve. Another measure could be in the type of techniques required in solving the puzzle. If the puzzle requires more advanced reasoning, then it is considered more difficult. In addition, a Sudoku or killer Sudoku puzzle must only have one possible solution which makes generating puzzles tricky. For Sudoku a possible there are two main approaches, one is to start with an empty grid, and then add values that satisfy the Sudoku constraints until a matrix is reached. The other approach works in the opposite way which starts with a full grid and then removes values [7]. I will be using these methodologies to generate my puzzles for both types of puzzles with different difficulties.

I am also going to create a web application that allows access to the solver and all the other features. I will use the Django framework to create my back-end API and then connect it to my front-end. The project is set up on a website due to its portability as it can be used on all devices such as mobiles, laptops, computers, etc. Django is a great framework as it follows the MCV (Model-Controller-View) pattern, provides in-built security, fast and easy to use. I will be using the Django documentation [8] to set up the website as I am not proficient in it.

I will also extend my project to be able to use machine vision to allow users to be able to upload their puzzles into the app as images. Machine vision is effectively used to allow a computer to see using a camera. One approach [9] to extract a puzzle is 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 individual 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 recognize digits. This will improve the usability of the application as the user is not required to manually enter in 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.

Milestones

The first and main milestone of the project is to create my own 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. This will be used to write the report on what constraint solvers are and 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 grid. 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 also create an interface to allow the user to play the game on the website as well as a feature that allows them to upload images of puzzles. The purpose of setting up the application on a website is because they are very accessible and can be developed for any device with little change in code.

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 was able to generate my own 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 as well as 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.

Timeline

Term 1

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| task | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| Research machine vision to detect puzzles |  |  |  |  |  |  |  |  |  |  |  |
| Create a page for playing Sudoku and killer Sudoku |  |  |  |  |  |  |  |  |  |  |  |
| Research into constraint solvers |  |  |  |  |  |  |  |  |  |  |  |
| Implement a solver for sudoku and killer sudoku |  |  |  |  |  |  |  |  |  |  |  |
| Report: Section on Constraint solvers – backtracking, recursion |  |  |  |  |  |  |  |  |  |  |  |
| Implement machine vision for detecting sudoku puzzles |  |  |  |  |  |  |  |  |  |  |  |
| Train ML model for number recognition |  |  |  |  |  |  |  |  |  |  |  |
| Report: Section on machine vision |  |  |  |  |  |  |  |  |  |  |  |
| Research into generating puzzles |  |  |  |  |  |  |  |  |  |  |  |
| Report: Section on NP hardness and Big O notation |  |  |  |  |  |  |  |  |  |  |  |
| Research the Sudoku and killer Sudoku solving techniques |  |  |  |  |  |  |  |  |  |  |  |
| Report: Section on techniques and algorithm used by human solvers |  |  |  |  |  |  |  |  |  |  |  |
| Work on the presentation |  |  |  |  |  |  |  |  |  |  |  |

For term 1, I want to be able to research and implement the machine vision code which will allow Sudoku images to be converted into a 2D array. However, machine vision only identifies and extracts the puzzle, I then need machine learning to identify the numbers in the cells. I also want to research and implement a simple backtracking constraint-solving algorithm which can then be integrated into the website. In addition, I want to get started on research into generating my own puzzles, and the NP-hardness of sudoku. I will also generate a report on my research with some code snippets.

Term 2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| task | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| Optimize the solvers by adding real techniques |  |  |  |  |  |  |  |  |  |  |  |
| Report: Section on data structures used and how it impacted the solver |  |  |  |  |  |  |  |  |  |  |  |
| Implement an algorithm to generate my own puzzles |  |  |  |  |  |  |  |  |  |  |  |
| Report: Section on generating puzzles. |  |  |  |  |  |  |  |  |  |  |  |
| Machine learning to determine if a puzzle is present in an image |  |  |  |  |  |  |  |  |  |  |  |
| Report: Section on comparing a recursion and backtracking algorithm |  |  |  |  |  |  |  |  |  |  |  |
| Optimizing the machine vision code to improve accuracy |  |  |  |  |  |  |  |  |  |  |  |
| Report: Software engineering processes used |  |  |  |  |  |  |  |  |  |  |  |
| Additional time for any unresolved issues. |  |  |  |  |  |  |  |  |  |  |  |

For term 2, the focus is to implement code for generating my puzzles and optimizing my machine vision code. Since machine vision is difficult to get right, it must be up to a good standard otherwise it would force the user to constantly change the numbers manually. I will also add more sections to my report on the software process I have used and any data structures I have used.

Risks and Mitigations

Knowing the risks of a project is important for all projects as they can potentially lead to significant delays or cause the failure of a project. Therefore, in this section, I will list some of the risks and the mitigating actions I will take to reduce them.

**Task estimation risks**

Due to the varying difficulty of the tasks and my lack of complete understanding of all the tasks, there is a high probability that I have underestimated the time required for some tasks. I will try to ensure that this does not lead to significant time issues by re-adjusting time for tasks so that difficult tasks get more priority. The impact of this risk is very high as it could lead to the project being left incomplete at the end, however, the probability of it occurring is low.

**Machine Vision risks**

Machine vision can be a tricky task to work with due to its complicated nature. Therefore, there is a risk that I end up getting stuck on the problem and needing too much time to fix it. Therefore, I will try to mitigate this problem by doing sufficient research on the topic and gaining a good understanding of it. The impact and probability of this risk is medium as it is important that the puzzles are being properly recognized or this could be unused.

**Machine Learning risks**

As with all machine learning algorithms, accuracy is a major factor in how good the model is. Therefore, there is a risk that I spend too much time trying to maximize the accuracy and compromise time of other tasks. I will mitigate this by setting a base accuracy that would be satisfactory for the project, so I do not get caught up in over-optimizing the model. Due to the amount of research into number recognition, the probability of this risk is low but as it is important it has a medium impact.

**Hardware risks**

As with all things on a computer device, there is a risk that a hardware fault occurs which could lead to code, or the report being lost or maybe even accidentally deleted. Therefore, I will make sure this is not an issue by putting my code on the git-lab repository and backing up my report. The probability of this risk occurring is high but due to regular backups its impact is very low.

**Code Readability risks**

When working on projects like this which have many complicated pieces, there's a chance that the organization of the code becomes messy and unreadable. This in turn will make the code harder to maintain and debug, which will waste time. Therefore, I will make sure I always keep my code clean and organized by keeping UML diagrams and writing object-oriented code. The probability of this risk is high as it is difficult to maintain code, but the impact is low.

**Report and Code risks**

When working on code it is possible to leave out the report until the end of the project, but since the report makes up a significant percent of the report this should not happen. Therefore, I will make sure that at the end of each week, I will work on the report to keep it up to date. The probability of this risk is medium as it is easy to just code and leave the documentation for later, but I will try to keep a good balance. The impact of this risk is high as it could cause the report to be badly written if not kept regularly updated.

# 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 optimize, 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 possible values. It is usually implemented within a single function and is very easy and simple to build. In the case of 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 many times time until it finds the correct solution. For sudoku, there is a possible 6.6 × 1021 valid sudoku grid [14] 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 carried out before the search algorithm and its goal is to remove any inconsistent values from the domains [15]. In this case, 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 which is used to make the standard backtracking algorithm faster. When we have reached a dead end, where a variables domain is empty, we usually backtrack in chronological order, but in back jumping we backtrack directly to the variable assignment which caused the dead end to be reached [16]. By doing this we can avoid search with a variable which 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 which contains the variables that were involved in the conflict, the values assigned to those variables at the time and the specific constraint that was broken. The conflict set tells us the conflict variable which is most responsible for the conflict and hence backtrack to change the value assign 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 “minimize the size of the search tree” [17] and prune a branch as quickly as possible if it incorrect. This approach is also called Minimum Domain Size (MiD) which implements the “fail first” [5] 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 order. Then when we make a recursive call, we choose the next value by popping from the queue.

Another approach which is like the DiM 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 Smaller Value of the Domain (SVaL) and Greater Value of the Domain (GVaL) where the heuristic chooses the smallest and largest value in the domain respectively [17]. However, in the case of sudoku this doesn’t work but we can observe that each value 1-9 must appears 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 which is concerned with verifying and updating the domains of the related variables after assigning a value to a variable [18]. The main idea behind this algorithm is to reduce the domain of the variables and thus reduces the size of the search tree. For sudoku, after assigning a value to 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 variables domain is empty then we know that the current assignment was wrong so we can immediately backtrack and pruning the branch earlier. In addition, we can exclude a value which 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 remove 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 is working on a solver for sudoku which involves learning about constraint solvers and then onto the killer Sudoku solver. There are many libraries which are designed for the purpose of 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 Arc-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 constrain solving is to identify the domains for the variables within the problem. I started off by creating a getDomain method which takes a row and column and returns an array containing all the possible values it can contain. An obvious starting place for this would be to simply assign the number 1-9 to every cell and let the backtracking function deal with the incorrect values. However, I realised that this was not a good idea because some values would be guaranteed to not be correct so I could simply remove those values from the domain.

### Setting up a sudoku class

The first step to building the solver is to have a base class which will be used to represent the problem better. I started by defining a simple class called Sudoku and its constructor which simple 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 behaviours, 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 of 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. In the case of Sudoku an example of an inconsistent value would be any number which appears anywhere else in the cells 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 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 iterate through the given cells’ rows, columns, and box to get all the values which 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 which 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 of the core part of any constraint solver which is the backtracking algorithm. The algorithm works by first getting a cell which 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. In the case that 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 simply returns the first slot in the array which does not contain a 0 in it. If the method returns None it means that there 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 technique (explained in a future section) to do most of the heavy lifting and using 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 simply 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 which do not lead to a correct solution quicker. Also with a smaller domain we have a higher probabilty 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 smallest domains first. Since I wanted the heap to be efficient, I adapted the code provided by the python documentation [12] 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 why 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 set up I no longer require the getNextCell method as I simply need to just 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 cells domain and check if it contains the value assigned, if it does then I keep the original copy of the cell and its domain and push the updates 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 why I store the original copy is because if the cell was assigned the wrong value, then we need to put back 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])

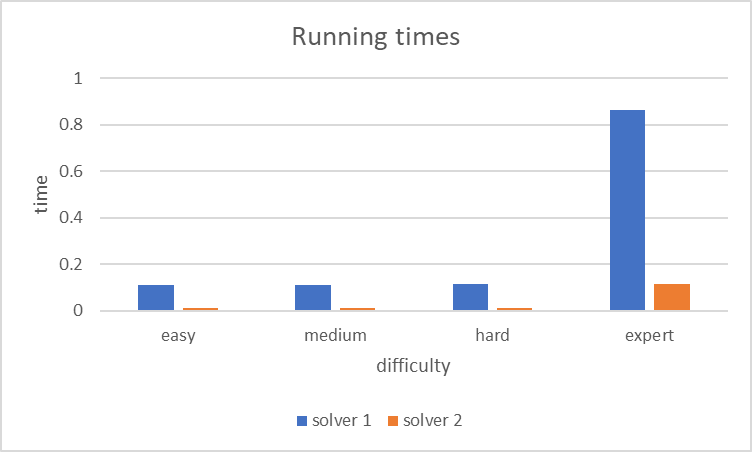
*self*.addToHeap((len(m\_set), updated[0], m\_set))

In the above code the updatedCells variable is the array from the previous code snippet. So, we just put back 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 simply mark them to be 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 compares the two different solvers have built and look at how they perform with different difficulties of puzzles.



1. Graph showing how the solvers perform on different puzzles.

In figure 1 the bars for the easy, medium, and hard puzzles represents 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 1 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 its roughly 10 times betters. 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, they expert puzzle shows that there is still room for improvement, this is because backtracking algorithms on their own are not very efficient.

## Killer Sudoku

My approach for the killer sudoku solver was very similar to the sudoku with some minor changes due to how the class was set up. 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 which had the cage number as keys and another dictionary and the value. The second dictionary had a key of 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 which 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 was able to get all information about a particular cell in O(1) time. I also created a method which checks whether the puzzle was valid, by simply 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 which 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

The above code first, I retrieve the cage number and then the cells in the cage, next the for loop adds 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 which returned an empty array if the remaining sum of the cage was 0 or below because it meant there’s 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 of the priority queue as described in section 2.3.1. While the concepts sued were the same, they were implemented slightly differently, the was difference was in the updating of the priority queue.

## Comparing the Killer Sudoku solvers

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

1. A graph showing the running time for the killer Sudoku solvers.

Figure 1 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 initial solver seems to perform better by about 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 about 12 minutes faster than the first. This clearly shows that the second solver performs significantly better for tougher puzzles while the first, does better on easier puzzles. One observation is that the killer Sudoku solvers are significantly slower than the Sudoku solver, that is because while Sudoku puzzle usually give a few prefilled cells, a killer Sudoku is completely empty. This means that the algorithm must fill the grid from scratch, as there 6.6 × 1021 valid sudoku grid [14] and more for killer sudoku since there are many different cage arrangements meaning this can take a long time. Figure 1 clearly demonstrates the need for a better solver because taking more than a minute would decrease interest in the application.

# Data Structures in Solvers

## Priority Queue

### Adding to the heap

The Constructor for the class initializes 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 simply 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 on the cell’s domain. The second element is the count which is a unique incrementing value. So, in cases 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 representing whether the elements are “active” or not. The reason why 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 simply use the store values.

### Removing and popping items

When talking about 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 simply 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, simply finds the entry using the key\_map dictionary and change the last element to the removed status. As for the pop method it simply 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 which 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 the 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 return 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 need to be updated. Also, in cases where we need to backtrack, the changes made previously need to be reverted so essentially re-update the affected cells. Since updating and reverting is going to be needed a lot, multiples 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 simply 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 cell 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 simply 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 cell which 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 which were updated in the decreaseKey we can implement a simple method which just puts those entries back into the queue.

    def increaseKey(self, updatedCells):

*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))

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

When updating cells which are part of the cage, we cannot simply just remove the values guessed as the case with non-cage cells. Instead, the domains change is 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 many 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 prioritised on 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 get all its other cage cells is very inefficient. Therefore, I implemented another dictionary which maps each cell to its cage number, now I can get all information about the cages in O(1) time. This is important because solver frequently needs to know which cells are groups together and if implemented inefficiently then the running time could grow by a lot.

Sets are also a useful data structure which 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 exists within a set which is O(1) in time complexity.

# Bibliography

|  |  |
| --- | --- |
| [1] | W.-M. Lee, “Basic Sudoku Solving Techniques / Intermediate Techniques / Advanced Techniques,” in *Programming Sudoku*, Apress, 2006, pp. 47-171. |
| [2] | “Tips,” Killer Sudoku, [Online]. Available: https://www.killersudokuonline.com/tips.html. [Accessed 24 9 2023]. |
| [3] | Simonis, Helmut, *Sudoku as a constraint problem,* CP Workshop on modeling and reformulating Constraint Satisfaction Problems. Vol. 12. Citeseer,, 2005. |
| [4] | Simha, Pramod J., K. V. Suraj, and Tejas Ahobala, “Recognition of numbers and position using image processing techniques for solving sudoku puzzles,” in *IEEE-international conference on advances in engineering, science and management (ICAESM-2012). IEEE*, 2012. |
| [5] | R. Haralick and G. Elliot, “Increasing tree search efficiency for constraint satisfaction problems.,” in *Artificial Intelligence*, 1980, pp. 14:263-313. |
| [6] | Pelánek, Radek, *Human problem solving: Sudoku case study.,* Faculty of Informatics, Masaryk University, 2011. |
| [7] | Ritik D, Rishika K, Samay P, “Handwritten Digit Recognition using Machine and Deep Learning Algorithms,” 2021. |
| [8] | “Django documentation,” Django, [Online]. Available: https://docs.djangoproject.com/en/4.2/. [Accessed 01 10 2023]. |
| [9] | 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. |
| [10] | 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. |
| [11] | Sun, B., Sun, X., Wu, Y., Yin, Y., & Yang, G., “A New Algorithm for Generating Unique-Solution Sudoku,” in *In 2008 Fourth International Conference on Natural Computation (Vol. 7, pp. 215-217).*, 2008. |
| [12] | “Heap queue algorithm,” Python, [Online]. Available: https://docs.python.org/2/library/heapq.html#priority-queue-implementation-notes. [Accessed 13 10 2023]. |
| [13] | Brailsford, Sally C., Chris N. Potts, and Barbara M. Smith., “European journal of operational research,” *"Constraint satisfaction problems: Algorithms and applications.",* vol. 119, no. 3, pp. 557-581, 1999. |
| [14] | Bertram Felgenhauer and Frazer Jarvis., “Enumerating possible Sudoku grids,” University of Sheffield, 2005. |
| [15] | Bessiere, Christian., “Arc-consistency in dynamic constraint satisfaction problems.,” in *AAAI Conference on Artificial Intelligence*, Anaheim, 1991. |
| [16] | Dechter, Rina., “Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition,” *Artificial Intelligence,* pp. 273-312, 1990. |
| [17] | Crawford, Broderick, Carlos Castro, and Eric Monfroy., “Solving sudoku with constraint programming.,” in *Cutting-Edge Research Topics on Multiple Criteria Decision Making: 20th International Conference*, Chengdu/Jiuzhaigou, 2009. |
| [18] | Cazenave, Tristan, and I. A. Labo., *A search based sudoku solver.,* Labo IA Dept. Informatique Universite Paris 8, 2006. |
| [19] | “killer,” EasyBrain, [Online]. Available: https://sudoku.com/killer. [Accessed 2023 10 22]. |