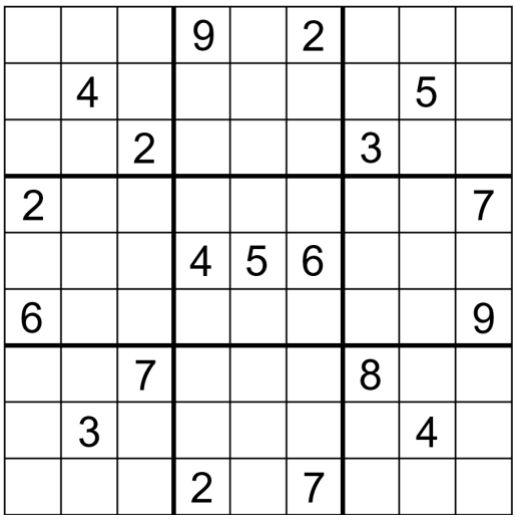
SUDOKU SOLVER



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

SHASHANK LAMA

AQA A-LEVEL COMPUTER SCIENCE

TABLE OF CONTENTS

Analysis

PROJECT OVERVIEW

I've decided to base my project on the game of Sudoku. Sudoku is a logic-based puzzle game that typically consists of a 9 x 9 grid. Within the main grid are nine 3 x 3 boxes. For each row, column and box, there are 9 spaces; the user must input a number into each empty space that doesn't already appear in the subsequent row, column or square. The grid is already partially filled, but less spaces can be occupied to make the game more difficult for the player.

There are many ways to solve a game of Sudoku; for the player, the best method is the process of elimination. This is where they look at what numbers have already been used and analysing which options they have. However, there are also computer algorithms that can solve a Sudoku puzzle in seconds. For my project, I am going to create a Sudoku game for my client which'll allow them to try to solve different puzzles themselves, with the option of also having the computer solve it.

I'm going to be creating this program to provide a solution to the problem for my client who currently cannot find a sudoku program that provides solutions offline, is visually appealing and one that he can later build on himself, if he pleases. I've made my system unique, such that, it can have the computer solve it using an algorithm and testing the limits to see how fast it can solve problems of varying difficulties.

THE INTENDED END USER

The end user for my project is my brother, who is an enthusiast of playing games. He often finds himself playing puzzle games when he doesn't have access to internet, for example when he's travelling. Sudoku is just one of the games he enjoys playing. However, sometimes he finds himself in a situation where he doesn't know how to solve a puzzle and would like to see the solution. Looking the puzzle up for the answer would not be a feasible option with no internet, and after observing different sudoku systems, we realised the majority do not have this feature.

Hence, came forth the idea of a Sudoku solver; using it, he can find a solution instantly. He's looking for a system that plays just like a typical Sudoku game, so nothing too overcomplicated. He expects the gameplay to be smooth, simplistic and bug-free. Moreover, he's asked me to design a system that he himself can reconfigure in the future where he would deem necessary.

Interview

I began my research by asking my client what they'd like for the Sudoku program. I've added a table of questions asked and his responses.

Question	Response
What are the main features you're expecting in this Sudoku game?	The ability to change cells with numbers only. The ability to check the puzzle at any point, and once finished, return an indication if it's correct or not. The ability to play with different difficulty boards.
What kind of unique features would you be looking for?	The program should be able to solve any puzzle that is presented.
What features are you looking for in the GUI?	I'd like it to be neat and colourful. There should be buttons which you can click that changes the difficulty of the board.
How important is the appearance of the GUI?	Fairly important. I would like a simplistic and aesthetically appealing design.
Would you like to attempt a puzzle already solved to see if you can beat your previous time?	Yes, that would be nice.
Would you like there to be a login system to save your specific times?	Not really, this isn't very important to me.
If you could change one thing, what would you change about the Sudoku game?	The playing grid/ square.

Analysis of Interview

After the interview, I got a better grasp for what my client was looking for. He wants a program that plays just like a typical Sudoku game with the added bonus of solving itself. Upon reviewing the features, I think it would be feasible for me to aim for most of these, such as a variable difficulty setting, however some I don't think would be necessary, such as a login system, which I've discussed with my client, and has stated that this isn't important, as it is designed for his own personal use.

Additionally, it seems the best thing is to not overcomplicate it and simply create a program that can function well and provide the purpose that was designed for it; in this case, I believe that the most significant parts of the program is getting it to produce a Sudoku board, produce a grid of numbers onto the board, allow the user to input numbers, check the numbers whilst you're playing and a button that will solve the subsequent puzzle.

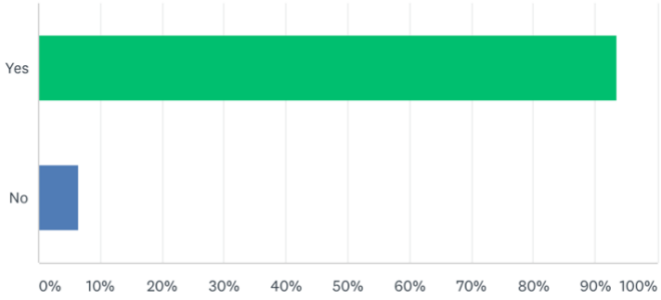
Survey

In addition, I've completed a survey using SurveyMonkey to a group of about 30 anonymous students in my school. Here are their subsequent responses. Note that not all the responses are being shown for open-ended questions.

Q1 Customize Save as

Have you ever played sudoku?

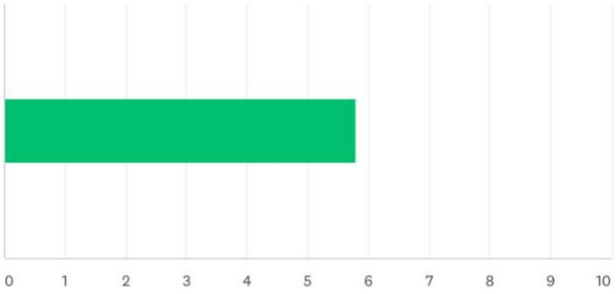
Answered: 31 Skipped: 0



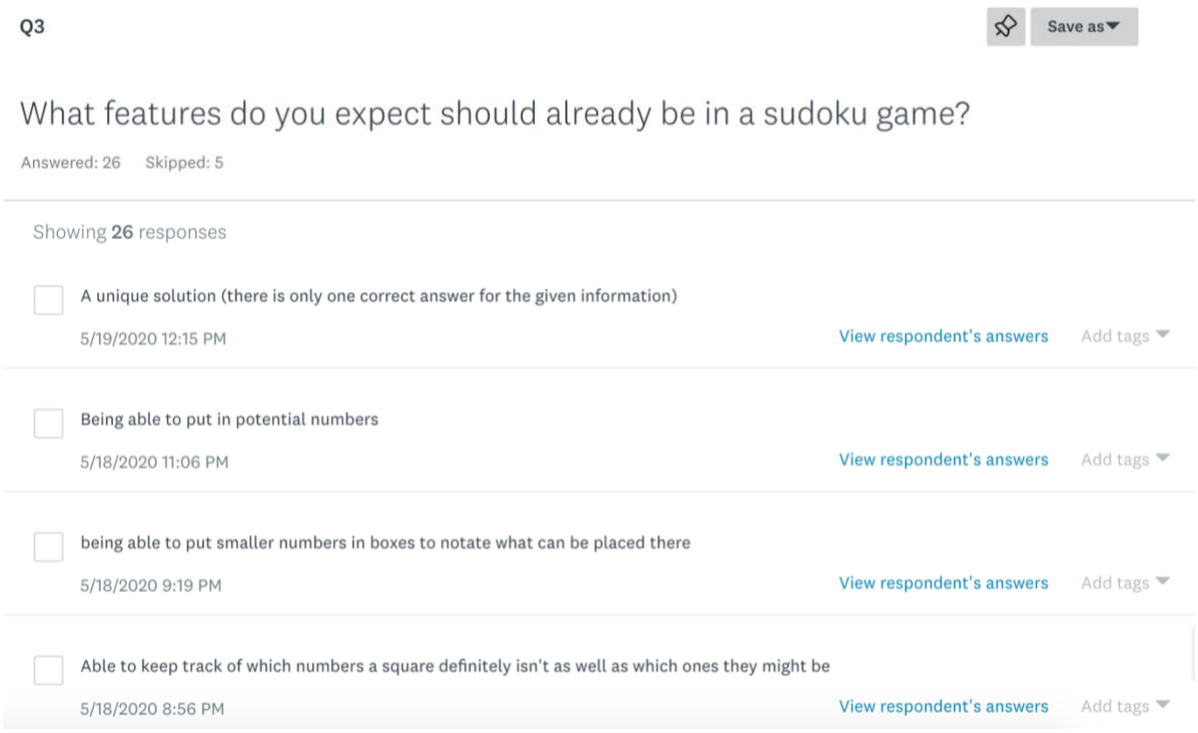
Q2 Customize Save as

If so, to what extent do you enjoy playing it?

Answered: 29 Skipped: 2



ANSWER CHOICES	AVERAGE NUMBER	TOTAL NUMBER	RESPONSES
Responses	6	168	29



Q3

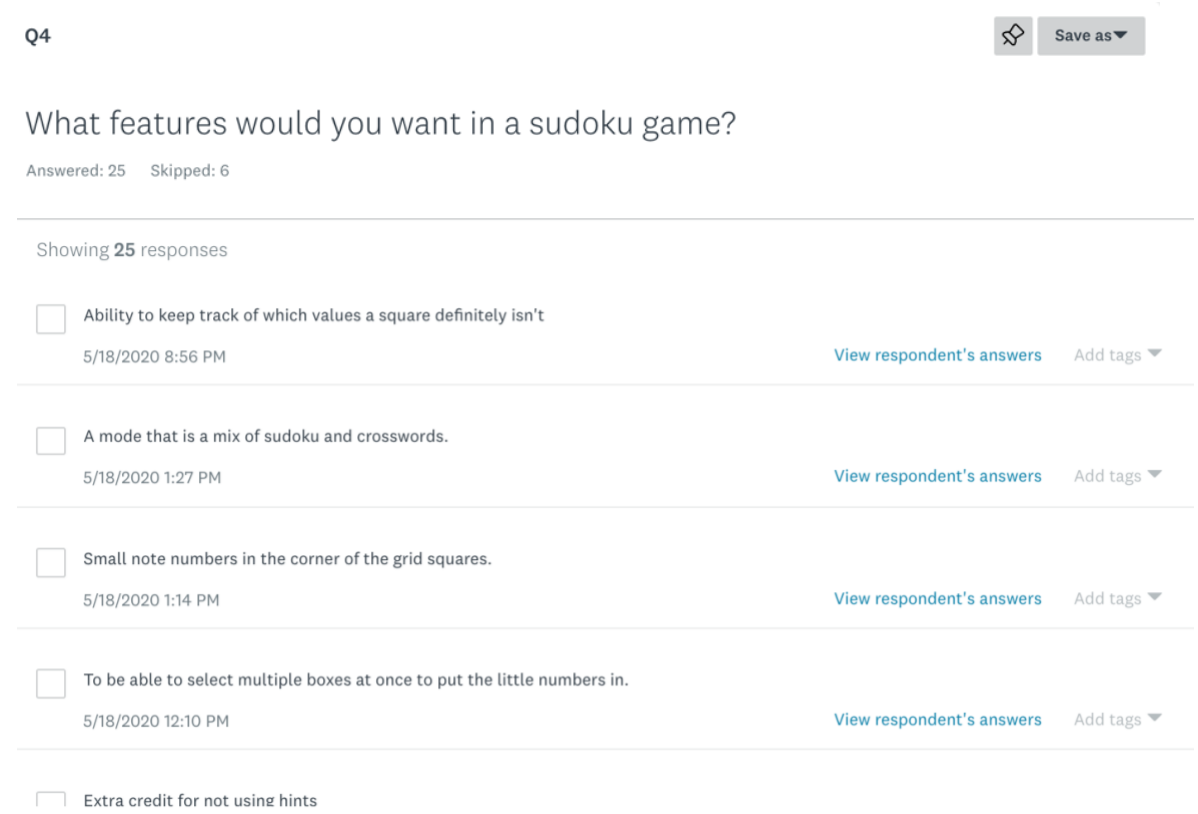
Save as

What features do you expect should already be in a sudoku game?

Answered: 26 Skipped: 5

Showing **26** responses

A unique solution (there is only one correct answer for the given information) 5/19/2020 12:15 PM	View respondent's answers	Add tags
Being able to put in potential numbers 5/18/2020 11:06 PM	View respondent's answers	Add tags
being able to put smaller numbers in boxes to notate what can be placed there 5/18/2020 9:19 PM	View respondent's answers	Add tags
Able to keep track of which numbers a square definitely isn't as well as which ones they might be 5/18/2020 8:56 PM	View respondent's answers	Add tags



Q4

Save as

What features would you want in a sudoku game?

Answered: 25 Skipped: 6

Showing **25** responses

Ability to keep track of which values a square definitely isn't 5/18/2020 8:56 PM	View respondent's answers	Add tags
A mode that is a mix of sudoku and crosswords. 5/18/2020 1:27 PM	View respondent's answers	Add tags
Small note numbers in the corner of the grid squares. 5/18/2020 1:14 PM	View respondent's answers	Add tags
To be able to select multiple boxes at once to put the little numbers in. 5/18/2020 12:10 PM	View respondent's answers	Add tags
Extra credit for not using hints		

Q5

Save as

What would you like in the GUI?

Answered: 25 Skipped: 6

Showing **25** responses

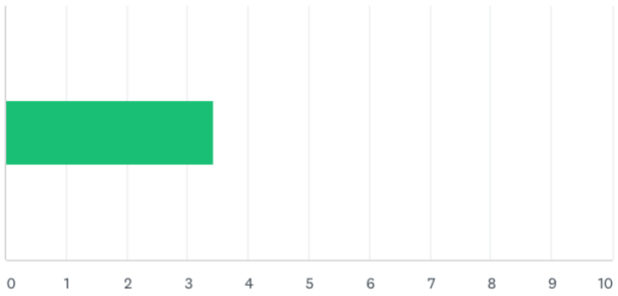
GUI should be easy to use and it should display the time taken. 5/18/2020 1:27 PM	View respondent's answers	Add tags
Clear entry boxes - simple interface. 5/18/2020 1:14 PM	View respondent's answers	Add tags
Functionality 5/18/2020 12:10 PM	View respondent's answers	Add tags
Clean, simplistic 5/18/2020 12:01 PM	View respondent's answers	Add tags
Colours, smooth buttons.		

Q6

Customize Save as

How important is the appearance of the GUI in your opinion?

Answered: 28 Skipped: 3



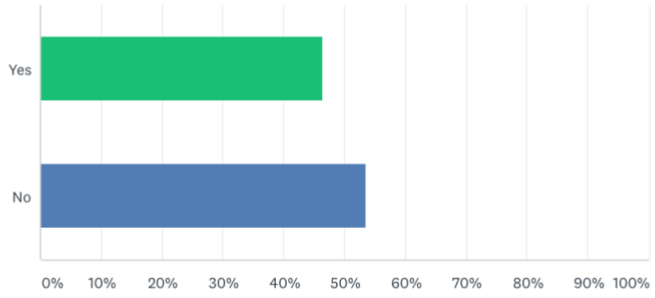
ANSWER CHOICES	AVERAGE NUMBER	TOTAL NUMBER	RESPONSES
Responses	3	96	28

Q7

Customize Save as

Would you like to attempt a puzzle already solved to see if you can beat your previous time?

Answered: 28 Skipped: 3



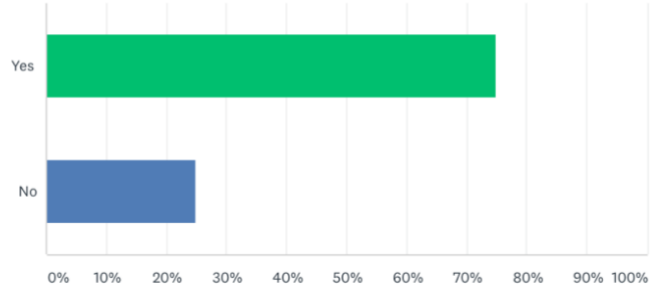
ANSWER CHOICES	RESPONSES	
Yes	46.43%	13
No	53.57%	15

Q8

Customize Save as

Would you like there to be a login system to save your specific times?

Answered: 28 Skipped: 3

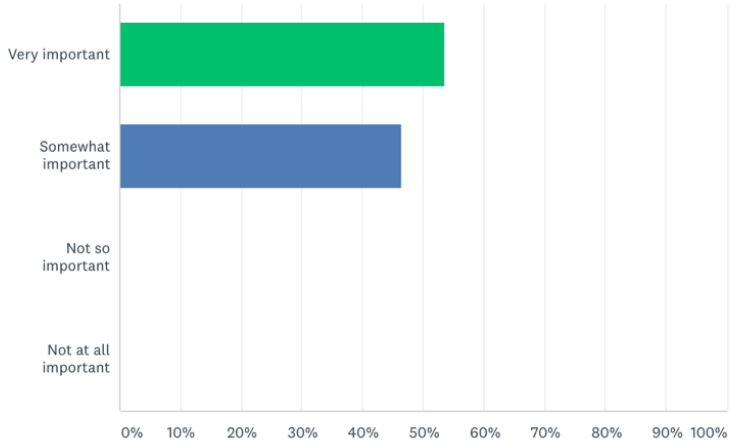


Q9

Customize Save as

How important is it for there to be a setting to change the difficulty of the puzzles?

Answered: 28 Skipped: 3



Q10

Save as

If you could change one thing, what would you change about the sudoku game?

Answered: 22 Skipped: 9

Showing **22** responses

There are a lot of sudoku variations, personally I prefer killer sudokus but any slight change that allows more complexity/variety can be good. If you want something very different you could do something with multiplayer - a 1v1 race would be easiest, but if there's a way to do something with more interaction between players (idk if this is possible with sudoku) then you might be able to do something like BTD Battles or Tetris 99.

5/18/2020 11:36 AM View respondent's answers Add tags

idk

5/18/2020 11:12 AM View respondent's answers Add tags

Multiplayer against friends

5/18/2020 10:42 AM View respondent's answers Add tags

Have a multiplayer mode where you compete against someone to see who can complete the sudoku game the fastest.

5/18/2020 10:33 AM View respondent's answers Add tags

Analysis of Survey

From the results gathered, I reviewed the survey and compared them to my client's responses. Most of the population from the survey had played Sudoku before, which meant that the survey would be more helpful/ reliable.

The group generally had a shared consensus for what should already be in a Sudoku puzzle; this includes things such as: there being a unique solution, the ability to undo a move and buttons used to select different difficulty settings. There were also some noteworthy features to include like highlighting which cells are incorrect. Most of this match my own client's point of view, thus I will be sure to try my best to implement most of these features.

Furthermore, a significant portion of responses stated the appearance of the GUI was not important which agreed with my client's own view. There were suggestions of the incorporation of colours too, which I agree to an extent, as adding unnecessary colours could make the board look improper and go against any functional use. Thus, I will put them in moderation.

Moreover, almost half of individuals, about 52% to be exact, were not concerned with beating their times in previous puzzles. However, roughly 75% did want a login system to save their previous times. These responses seem to contradict each other, as if the majority do not want to replay previous puzzles, what would be the need to save times? My client himself has stated that a login system isn't that important, so as mentioned above, I do not think I will be adding this feature. However, since I think I will be storing a collection of puzzles in a database, it would inevitably mean that some puzzles will be replayed.

Finally, when I asked people what features they'd want to change in Sudoku, I received some interesting responses. Some suggested a multiplayer option where they could challenge their friends and others proposed alternate Sudoku variations. I did consider applying multiplayer to my Sudoku game before, but after reflecting on my networking abilities as well as my own client's wishes, I didn't deem it to be necessary.

RESEARCH

Research Terminology

I've included a table of key terms that should help with understanding some of the specific terminology that'll be used later on in this section.

Key Terms	Definition
Population	This refers to the collection of individuals at the beginning of the natural selection process; each individual represents a solution for the problem to be solved.[1]
Genes	Variables that determine the unique characteristics of an individual.
Chromosomes	Genes joined together.
Fitness function	This determines the fitness of the individual and thus, how likely to survive against competitors by comparing the fitness score.
Fitness score	It is an indication of the probability that the individual will reproduce.
Parents	They are two pairs of individuals.
Selection phase	This is where a portion of the population are selected based on their fitness scores. Individuals with higher fitness scores are more likely to be selected, and thus pass on their genes to their offspring.
Crossover	This is where a crossover point is chosen at random for each pair of parents to be mated using their genes. Genes are exchanged among the parents until the crossover point is reached and the end result is new offspring. These new offspring are then added to the population.
Mutation	Random changes are applied to the individual parents to form children[2]. This occurs to maintain diversity within the population.
Termination	The algorithm terminates once the population stops producing offspring that is considerably different from the previous generation.

[1] "*Introduction to genetic algorithms"* – towardsdatascience, 08/06/18, accessed: 19/04/20. https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

[2] "*Genetic algorithms"* – computersciencewiki, accessed: 25/04/20. https://computersciencewiki.org/index.php/Genetic_algorithms#Mutation

Current Sudoku Systems

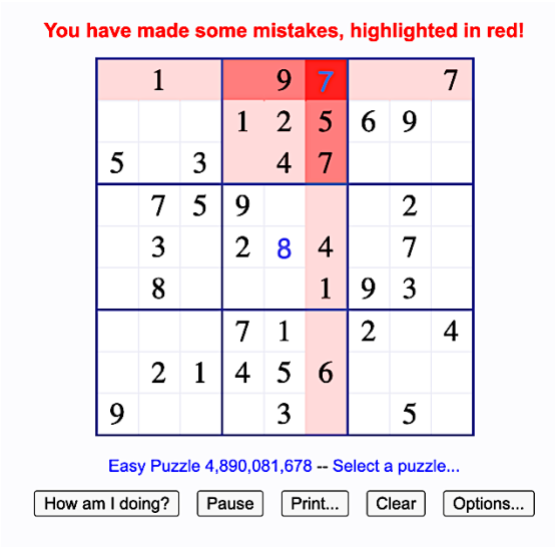
In this section, I explore different sudoku systems that have been created. There're are an abundant selection of platforms to play sudoku on, but here I've analysed just a few, looking at some of their unique features, what I like and could fit into my own program and things I don't like about each.

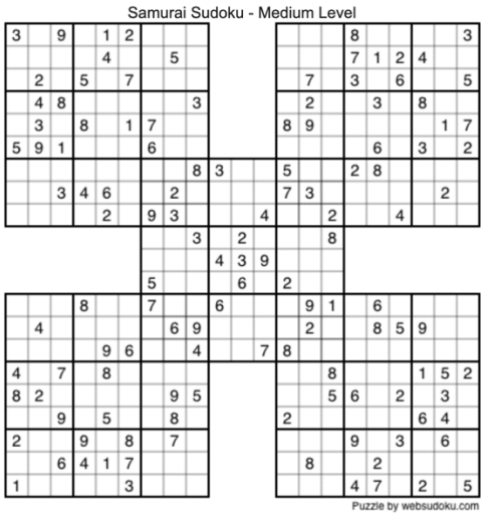
Web Sudoku

Web Sudoku is a popular website for playing Sudoku online[3]. Upon entering the website, you're invited to a puzzle at the centre of the screen with a message prompting the user to solve it. What makes Web Sudoku popular is that it boasts a grand selection of different puzzles, which include features that appeal to players of all skill levels.

For example, beginners can start off on at an 'Easy' difficulty setting and as they play, they can check for mistakes by utilising the 'How am I doing?' button. This reveals any areas of mistakes by highlighting the subsequent row, column and sub-grid; I will be adding a check button for my program which has a similar functionality but will be highlighting any specific occurrences of a number that has been repeated.

More experienced players can opt for harder difficulties, or even try variations of the standard grid, such as the 'Samurai Sudoku' one which is shown below. I do like the idea of having different types of sudoku puzzles, but my client hasn't indicated that he would be playing these kinds of puzzles in the first place. It also brings the question that, as the grid changes more and more, at what point is it still sudoku, or is it just a brand-new game? For these reasons, I think I will stick with the original untampered format for now, and perhaps if I decide to adjust my program in the future, I may try to create a solver program for the different variations as well.





[3] "*Web Sudoku*" – WebSudoku, accessed: 07/11/20. https://www.webSudoku.com/

Sudoku Kingdom

Sudoku Kingdom is another online platform to play sudoku, similar to Web Sudoku.

What makes Sudoku Kingdom unique to Web Sudoku is that it hosts a leader board system which saves the scores of registered users. These scores are calculated from the number of puzzles solved along with their difficulty level. They're updated frequently, as shown below by the 'Last played' section. This could be a great feature for multiplayer; however, I've already mentioned from my analysis of the survey that, upon reflecting of my networking capabilities as well as my client's own wishes, I didn't think that this kind of application would be worthy of the limited time I have for the project.

There's an auto checking system in place which prevents the user from making any wrong moves. I do not like this, as experienced/ more competent players, such as my client, would not appreciate the extra help and may feel they're cheating themselves by constantly being told when they're making a wrong move. In this regard, I do not think that Sudoku Kingdom does as well a job at appealing to players of all skill levels as Web Sudoku does. Hence forth, I will refrain from adding such a feature.

Like Web Sudoku, there's a variable difficulty setting, which I am planning on adding. Responses from the survey and the interview with my client has indicated that this is a crucial feature, thus I will be sure to attempt to implement this,

[4]

Player Rankings - November 2020 - Top 10

#	Login name	Solved Puzzles Easy	Med.	Hard	V.Hard	Score	Last played
1	wagdyss	0	0	0	128	1024	Nov 08, 2020
2	Simonez	0	0	0	128	1024	Nov 08, 2020
3	Phingme	0	0	0	128	1024	Nov 08, 2020
4	icecube98	0	0	0	128	1024	Nov 08, 2020
5	asparkkim	0	0	0	128	1024	Nov 08, 2020
6	Fukchan	0	0	0	127	1016	Nov 08, 2020
7	slchew	2	1	1	124	1000	Nov 08, 2020
8	ronnin	4	2	2	122	992	Nov 08, 2020
9	ipgabriel	0	0	0	120	960	Nov 08, 2020
10	TrungCom	0	0	0	118	944	Nov 08, 2020

Every solved puzzle the score is incremented using the following key:
Easy=1 point Medium=2 points Hard=4 points Very hard=8 points

*** Show Full Player Ranking Table ***

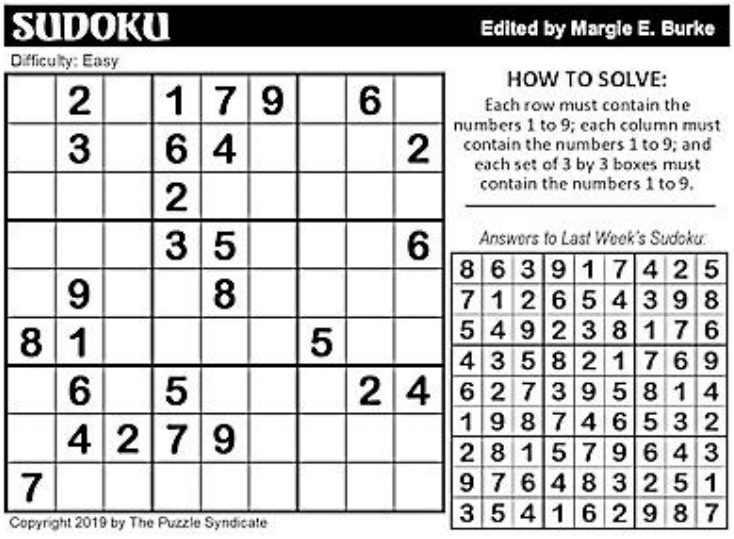
[4] "*Sudoku Kingdom*" – SudokuKingdom, accessed 10/11/20
https://www.sudokukingdom.com/

Newspaper Sudoku

Sudoku in newspaper in a way is Sudoku in its purest form. We're given nothing but a grid with numbers and a description of how to solve the puzzle; nothing more, nothing less. They're used as a fun activity you could do whilst you read your newspaper, usually containing the answers to a Sudoku from a previous paper.

Compared to the other examples mentioned above, the way Sudoku is played is very primitive and due to there being no real cues to help the player, e.g. such as hints, it is also the hardest form. However, there's no difficulty setting, so puzzles can range from being doable to extremely difficult. I am keen on this format nonetheless, as it like idea of having the game in its raw, unedited form and would like the same for my own program. Rather than putting hints or features that may assist the user, I would like the user to play the Sudoku game how it was originally designed to be played; even though, my client probably wouldn't need them, as he is already proficient at puzzle games.

In addition, using a Sudoku solver could be perfect for a version like this. With no access to internet, simply plugging the numbers into the grid can give you a solution instantly, rather than waiting a set number of days for a solution to be given as commonly done by newspapers.[5]


SUDOKU

Edited by Margie E. Burke

Difficulty: Easy

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

HOW TO SOLVE:

Each row must contain the numbers 1 to 9; each column must contain the numbers 1 to 9; and each set of 3 by 3 boxes must contain the numbers 1 to 9.

Answers to Last Week's Sudoku:

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

Figure 1 A common layout of Sudoku found in newspapers

[5] "*Weekly and Monthly Crosswords & Sudoku Puzzles for Newspapers & other Publications"* – ThePuzzleSyndicate, accessed 07/11/20. https://www.puzzlesyndicate.com/

Current Sudoku Solving Algorithms

In this section, I explore different algorithms that have been used to solve sudoku puzzles. Some algorithms suit the style of Sudoku more and thus implement better when solving, making them ideal. I've analysed a few algorithms, looking into how they work and whether they could be appropriate for my program.

Backtracking

[6]A common algorithm used to solve a Sudoku puzzle is backtracking. Backtracking is a technique used in a brute force search, known as depth-first; depth-first search digs deeper into the problem than its counterpart, breadth-first search, and can be used recursively[7]. It works for tree and graph data structures and works by following edges and ensuring vertices are not met twice. [8]

```python
def depth_first_search_recursive(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    for next in graph[start] - visited:
        depth_first_search_recursive(graph, next, visited)
    return visited
```

Figure 1 Depth first search using recursion on python

Searching thoroughly means a solution is guaranteed in the end, as long as the puzzle is solvable. This means that backtracking is a general heuristic method, where it may not always arrive at the solution in the most optimal method but will still guarantee a solution at the end. The time taken to solve the puzzle is generally consistent as well, regardless of the difficulty of the puzzle. Moreover, it is a simple algorithm, which in turn means the program code would be very simple to produce. Backtracking is gone through more detail in the design stage.

[6] "*Depth-first search" – Brilliant, accessed 24/04/20.* https://brilliant.org/wiki/depth-first-search-dfs/

[7] "*Brute-force Approach"* – The World of Computing, 14/12/2009, accessed 24/04/20. http://intelligence.worldofcomputing.net/ai-search/brute-force-search.html#.XqKvrtNKjBJ.

[8] "*Depth-first search"* – Computer Science Toronto, 16/12/2002, accessed 24/04/20. http://www.cs.toronto.edu/~heap/270F02/node36.html

Stochastic Optimization

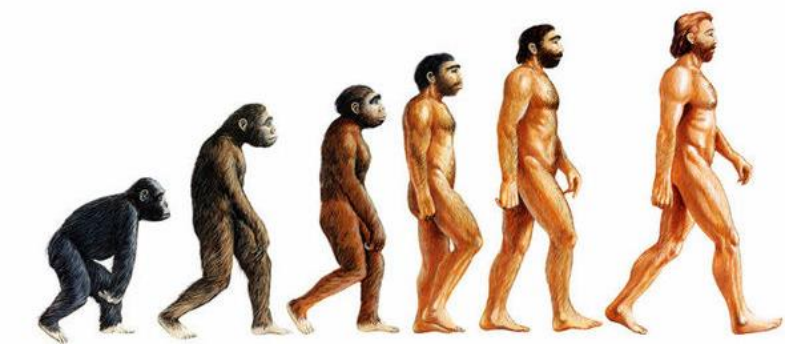
Another algorithm that can be used to solve Sudoku puzzles is by using stochastic optimization methods that generate and use random variables. An example of how it could be used is by:

I. Assigning random numbers to the empty cells in the grid.
II. Calculating the number of errors that occur
III. 'Shuffling' the numbers around until there're no errors and the solution is found.[9]

There're different ways to shuffle the numbers, but in this documentation, I looked specifically at genetic algorithms. Stochastic algorithms are known to be fast, which means they could be used as a solution for the lack of speed in backtracking.

Genetic Algorithms

Genetic algorithms are a type of evolutionary algorithm which imitates the process of natural selection, theorized by Charles Darwin[10]. This is where nature favours animals that have more desirable characteristics, which then consequently, causes them to survive and reproduce, thus passing on their genes on their offspring. This causes the species to adapt slowly over time in response to changes in the environment, or competition.[11] They are used across computer science to help solve optimization and search problems.



As genetic algorithms fall under the category of an evolutionary algorithm, it behaves like so:

[9] "*Sudoku-solving algorithms"* – Wikipedia, accessed 24/04/20.
https://en.wikipedia.org/wiki/Sudoku_solving_algorithms

[10] "*Human evolution"* – Express, 01/09/18, accessed 24/04/20.
https://www.express.co.uk/news/science/1011341/human-evolution-origin-ancient-footprints-crete

[11] "*What is natural selection?"* – BBC Bitesize, accessed 17/04/20.
https://www.bbc.co.uk/bitesize/topics/zpffr82/articles/z7hj2nb

I. Generate the initial population of individuals randomly. (First generation)
II. Evaluate the fitness of each individual in that population
III. Select the fittest individuals for reproduction. (Parents)
IV. Breed new individuals through crossover and mutation which creates new offspring.
V. Evaluate the fitness of the new individuals.
VI. Replace the least-fit individuals of the population with new individuals.[12]

This process is repeated until termination. This is important so that the new generation is better than the previous. A pseudocode for the genetic algorithm can be described like so:

```
START
Generate the initial population
Compute fitness
REPEAT
    Selection
    Crossover
    Mutation
    Compute fitness
UNTIL population has converged
STOP
```

Figure 2 pseudocode for genetic algorithm (Introduction to genetic algorithms, 2018)

The solution spaces

Empty spaces in a Sudoku grid refer to the solution space. A paper written by members of the University of the Witwatersrand discusses how they use stochastic methods to solve Sudoku puzzles.[13] In their research, they use a solution space which consists of one population; each individual has a select number of genes that correspond to the number of empty spaces in the grid. They've chosen this approach so that the individuals interact more amongst themselves, as the arithmetic operations are carried out between possible solutions. In addition, it is less tasking on the computer, since they've only used one population.

Fitness function

As for the fitness function, they've decided to use it to so that it determines whether an integer is repeated or not in a column, row or grid. The fitness value is assigned based on integers that

[12] "*Evolutionary algorithm"* – Wikipedia, accessed: 25/04/20. https://en.wikipedia.org/wiki/Evolutionary_algorithm

[13] "*Stochastic optimisation approaches for solving Sudoku"* – University of the Witwatersrand, 06/05/08, accessed: 25/04/20. https://arxiv.org/pdf/0805.0697.pdf

have already been repeated or are not present. It is also used as a guide to how close an individual is to the optimal solution. Applications of the fitness function and how it is implemented in context to Sudoku is delved into in design.

Initialization

For initialization, they've considered two different formats to encode the chromosomes. The first was to have each individual represented by binary. While it does create a more diverse population, I don't think it would be practical to use it, as most populations aren't naturally represented by binary due to its length. Instead, I think a better approach would be to use floating point encoding; it is more efficient than binary and allows for characters and commands to represent an individual.

Selection

The next stage being selection uses the fitness function previously discussed. What's important to note is that the selection function should not use only the fitness individuals, as this would result in the algorithm converging too quickly. Instead, they've described two different methods to implement this correctly:

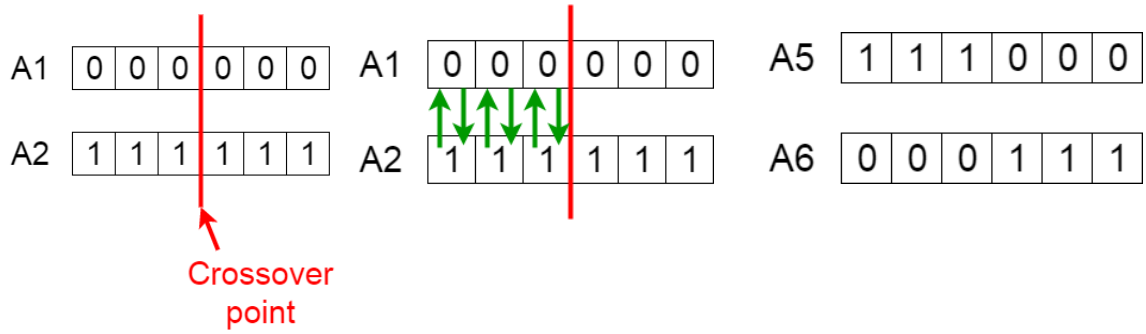
- **Roulette technique** – involves summing the fitness's of all individuals and selecting a random number between 0 and the sum.
- **Tournament method** – involves selecting a random number of individuals from the population and the fittest individual is selected. The more individuals selected, the better the chance of selecting the fittest.

I think I will be using the tournament method as in my opinion this is more likely to result in acquiring the solution faster.

Crossover

Once the mates have been selected, the genes go through crossover. If binary encoding is used, a point is chosen and genes up to that point come from one parent and after, from the other. Whereas, arithmetic crossover involves adding a percentage of the floating point of one individual to another. For instance, an individual has value 5.5 and another 10.7. If you select 30% from the one and 70% from the other the child will be 9.1. [14]

Termination



Finally, the genetic algorithm can terminate once a certain number of generations have been reached, and a suitable solution is reached. It is important in determining when a GA run would end. From observations, it has been concluded that the progression of GA is very fast initially, but this tends to saturate in later stages with only miniature improvements on each stage. Thus, you would ideally want a termination condition that is close to the optimal solution at the end of the run. There needs to be condition for termination, for example:

- When there has been no improvement in the population for X iterations.
- When an absolute number of generations is reached.
- When the objective function value has reached a certain pre-defined value.[15]

An example of a pre-defined value for termination is achieved using a counter. The counter increments each time off-spring is produced that isn't better than the individuals in the population already; this keeps track of the generations for which there has been no improvement in the population. But, if the fitness of the off-springs is better, you reset the counter back to zero. Consequently, if a predetermined value is reached, the algorithm terminates.

[14] *Introduction to genetic algorithms"* – towardsdatascience, 08/06/18, accessed: 25/04/20. https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3
[15] "*Genetic Algorithms – Termination Condition"* - tutorialspoint, accessed: 23/11/20. https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_termination_condition.htm

Graph Colouring

Sudoku can also be solved by algorithms designed for graph colouring. Graph colouring is where no two adjacent vertices have the same colour.[16] One graph can have more than one way to colour it, so the aim is to minimize the number of colours used.

For Sudoku, each square equates to one vertex, thus there'll be 81 vertices in the game. Two separate vertices are termed adjacent only if they're in the same grid, row or box. The ideal solution for a Sudoku puzzle is to find a colouring that uses 9 colours. However, at the moment not all puzzles can be solved using this method, so it wouldn't be very reliable.

[17]

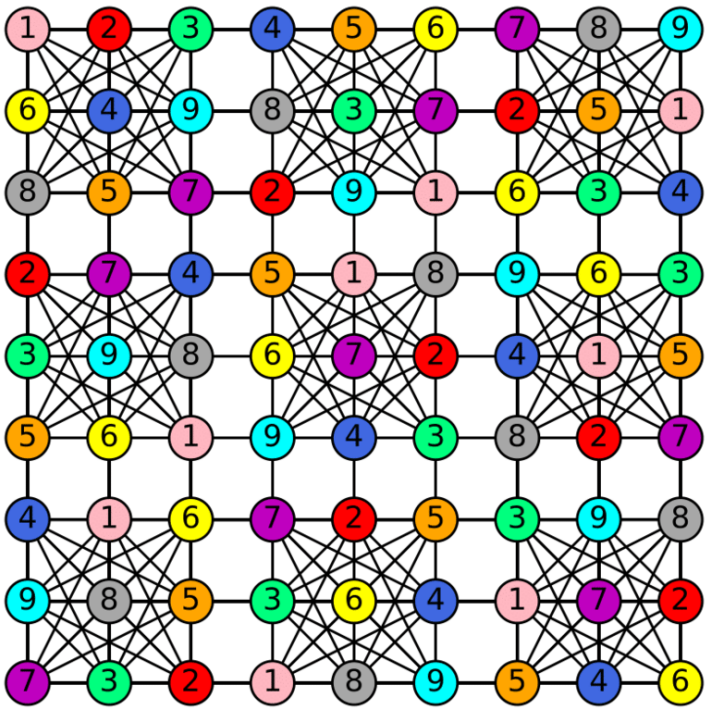


Figure 3 Graph colouring diagram for Sudoku puzzle

[16] "*Sudoku solver using graph coloring*" – codeproject, 28/07/14, accessed: 25/04/20. https://www.codeproject.com/Articles/801268/A-Sudoku-Solver-using-Graph-Coloring

[17] "*Graph coloring of Sudoku*" – ResearchGate, accessed: 30/09/20: https://www.researchgate.net/figure/b-Graph-coloring-of-Sudoku_fig9_311668725

POSSIBLE RESOURCES TO BE USED

Programming Languages

For the NEA, AQA have listed Java, C# and Python as options for the programming language. In the case of **Java**, it's main advantage in the context of my project is the ability to create a good GUI with far more ease than the other two. However, when considering that my client is looking for a "simple but elegant" GUI, it becomes clear that this is not the most significant worry, as any of the other languages stated can produce a respectable looking GUI. Personally, I don't think it is worth learning Java for this reason, especially when I would have to tackle the problem of learning a new language.

As for **C#**, it is that the program would run significantly faster compared to if it was written in Python. While this would definitely be a good feature to have, as it may mean that the computer can solve puzzles faster, I'd still stumble into the same problem with Java, which is that I'd have to invest more time learning how to code in these languages, which would be an issue given the time constraint.

Python is a high-level interpreted and general-purpose language that is renowned for its clarity and ease-of-use. A few advantages for me include:

- More efficient – due to simpler syntax when compared to other high-level languages
- Easier to debug – since it is an interpreter language so returns the first error
- Already familiar with the language – thus, less time spent trying to understand the programming language and more time spent programming

There are some notable drawbacks as well though. First of all, Python's library is said to be more restrictive when compared to the other languages, but personally, I have not had any issues. Secondly, Python receives runtime errors, meaning errors that only appear during runtime. Consequently, I may have to go through more testing to ensure the program is well-built. Finally, another problem that may raise issue is that Python has speed limitations, since it executes code line by line; this may result in puzzles being solved by the computer to take longer. While this isn't ideal, for me the benefits still make Python more preferable.

Therefore, after weighing the pros and cons, I decided I would be using Python as my programming language for this project.

Programming Style

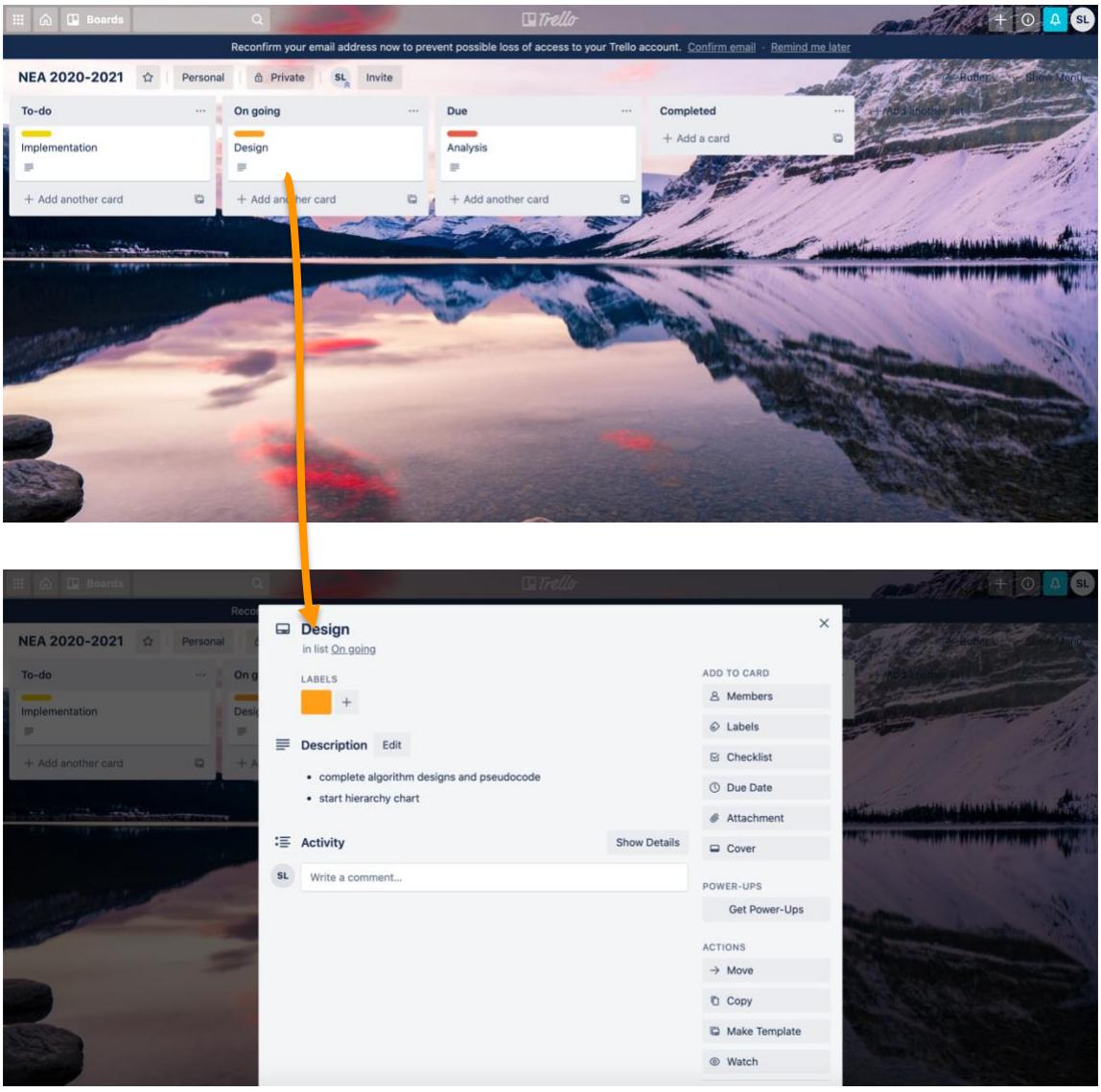
As for the style of programming I will be adopting, I will be using **object-oriented programming** in order to create my program. OOP is a type of **modular programming** which improves readability and makes debugging and maintaining easier. It uses the principles of decomposition, meaning the tasks are broken down into sub-tasks which simplify the process of writing the program, as well as improving the overall clarity. **Encapsulation** where possible is a design principle of OOP and is something I'm going to be incorporating. This means allows me to use methods which belong to other classes without needing to know how a specific method works. For instance, if I created two classes, a main class and a supplementary class, the additional class could have methods that are used in the main class.

Subroutines will have a mix of **modular** and **loosely coupled style**, meaning that they will be interacting with other parts of the program and will be **cohesive**, so will perform a single task. The vast majority of variables will be **local variables** I will be minimising global variables. Exception handling is a must, to prevent errors such as incorrect data entry or trying to convert a non-numeric string into an integer, for example ensuring only integers are inputted into the cells – characters like letters should be ignored. Avoiding the use of global variables and prioritising local variables will reduce the chance of error, so it's wise to do this. Furthermore, using meaningful identifiers, appropriate indentation and commenting where necessary helps to make the program more understandable, so I will be doing this where I deem appropriate.

Other Resources

Finally, I've adopted a project management tool known as Trello. Trello is based on Kanban – a method of workflow that helps visualise work, maximise efficiency and be agile. The benefits of using it as opposed to other project management tools is that: it is flexible, so there is no fixed duration for each task; it promotes productivity, as you can focus on work that's most important, which also prevents procrastination; and it is simple and has a familiar workflow.

However, the fact that there are no time frames for tasks can be a weakness if not handled properly. Thus, I need to try setting myself small deadlines for each task.



PROTOTYPE

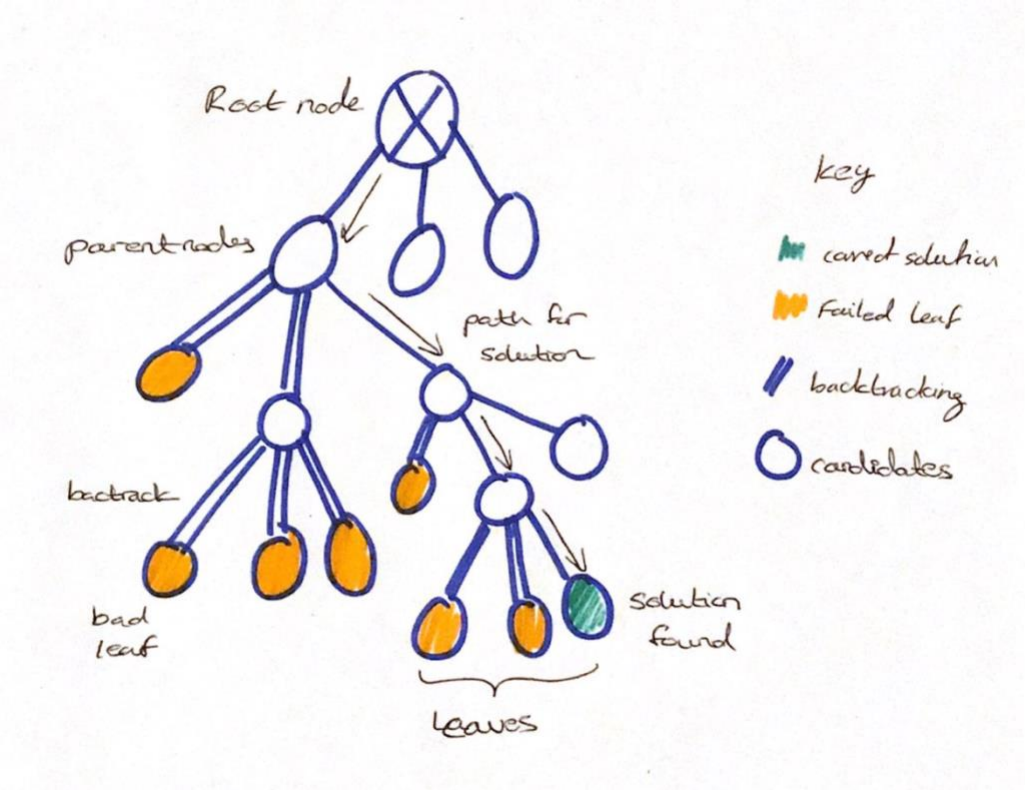
Backtracking is a fundamental algorithm in my project, as it will be responsible for the ‘solving’ feature, which will be achieved by using recursion. It works like so:

1) Find an empty space and choose an option from 1-9, depending on what’s available in the subsequent row/ column/ sub-grid.
2) If the option doesn’t get you to the solution, i.e. a finished Sudoku, you backtrack from it.
3) Repeat the steps until the solution is reached – where there are no empty spaces left.

Backtracking is an example of a selective **graph/ tree traversal method**. The **parent node** can represent the initial starting positions and the **leaves** are the final solutions. As mentioned in the research of the algorithm, the brute-force nature of backtracking means that you can refine the choices by eliminating the ones that are obviously not possible and then proceed to recursively check the potential choices. This way, at each depth of the tree, you can reduce the number of choices that can be considered in the future.

When a mistake is made and you go to a bad leaf, you then backtrack and continue to search for a good leaf. If you end up at the root with no more options available, this means that there are no good leaves to be found.

I’ve created a tree traversal diagram demonstrating this, as shown below



Moreover, I've included a pseudocode for a version of Sudoku, shown in figure 5, known as Mini Sudoku, which is a 3 x 3 grid – or simply a sub-grid in the typical Sudoku. Looking at this you can appreciate how backtracking works for Sudoku puzzles, which will help to play a role when designing the solver for my program.

```
function solve( board )
    if the board contains no invalid cells, ie.cells that violate the rules:
        if it is also completely filled out then
            return true
    for each cell in the board
        if the cell is empty
            for each number in {1,2,3}
                replace the current cell with number
                if solve(board) and the board is valid
                    return true
                else
                    backtrack
        return false
```

Figure 4 Pseudocode for Mini Sudoku

OBJECTIVES

After reviewing all my research in analysis, I have created a list of SMART objectives for my program, which is broken down further into different sub-headings. These objectives will be tested again at the end, once my program is created to see if I have met them.

1. GUI

1.1. Display the main playing board using Pygame
1.2. Display buttons which have a range of uses, including checking the board and changing the difficulty of the board
1.3. Create a simplistic and visually appealing GUI
1.4. Make use of colour in the GUI

2. Gameplay

2.1 Should be able to play the game like a normal Sudoku
2.2 Should include buttons which have unique features, which are checking the puzzle, choosing a difficulty setting for the puzzle and solving
2.3 Shouldn't be able to enter non-numeric values into the cell
2.4 Shouldn't be able to change cells that are in the grid before user starts

3. Backtracking Solver

3.1 Use recursion to solve the puzzle
3.2 It should be able to solve any puzzle
3.3 The time to solve each puzzle should be consistent

4. Database

4.1 Successfully store puzzles in a database
4.2 Access the database when a difficulty is selected and produce a board from that subsequent difficulty

Design

In this section, I will be explaining the design and further planning aspects for the implementation of my code. Initially I researched a range of different sudoku systems for inspiration, to get a better idea of how I would design a program best suited for my client's needs[18]. I will be including some diagrams which summarises key parts of the program, as well as explanations of different algorithms in context of Sudoku. I will also speak about data structures, libraries, the user interface and a test plan used for testing.

TABLE OF SKILLS TO BE USED

This table includes all the technical skills I'm planning on using.

Skill used	Application	Objective Achieved
Complex user defined use of Object-Oriented Programming (OOP)	Programming style	1.1 Display the main playing board using Pygame 1.2 Display buttons which have a range of uses, including checking the board and changing the difficulty of the board 2.1 Should be able to play the game like a normal Sudoku
Exception handling	Preventing errors	2.1 Should be able to play the game like a normal Sudoku 2. 3 Shouldn't be able to enter non-numeric values into the cell 2.4 Shouldn't be able to change cells that are in the grid before user starts
Files organised for direct access	Organise files well to make the program run and allow database to be created properly	4.1 Successfully store puzzles in a database 1.1 Display the main playing board using Pygame 1.2 Display buttons which have a range of uses, including checking the board and changing the difficulty of the board

[18] "Let's make Sudoku" – A Plus Coding, YouTube, 21/12/19 https://www.youtube.com/watch?v=r_cmJBgrq5k&list=PLryDJVmh-ww0J6BDQrdLtRyNOv4Lym_xo&index=1

Recursive algorithms Graph/ Tree Traversal	Solving the board	3.1 Use recursion to solve the puzzle 3.2 It should be able to solve any puzzle 3.3 The time to solve each puzzle should be consistent
Multi-dimensional arrays Lists List operations	Storing the sudoku puzzle	1.1 Display the main playing board using Pygame 4.1 Successfully store puzzles in a database 1.2 Display buttons which have a range of uses, including checking the board and changing the difficulty of the board
Simple mathematical modelling	Checking grids consist of one occurrence of each number	1.1 Should be able to play the game like a normal Sudoku 1.2 Display buttons which have a range of uses, including checking the board and changing the difficulty of the board
Simple data model in database Normalisation	Storing puzzles Removing repetition of data in tables	4.1 Successfully store puzzles in a database 4.2 Access the database when a difficulty is selected and produce a board from that subsequent difficulty
Text files Writing and reading from files	Reading data	4.1 Successfully store puzzles in a database 4.2 Access the database when a difficulty is selected and produce a board from that subsequent difficulty 1.1 Display the main playing board using Pygame

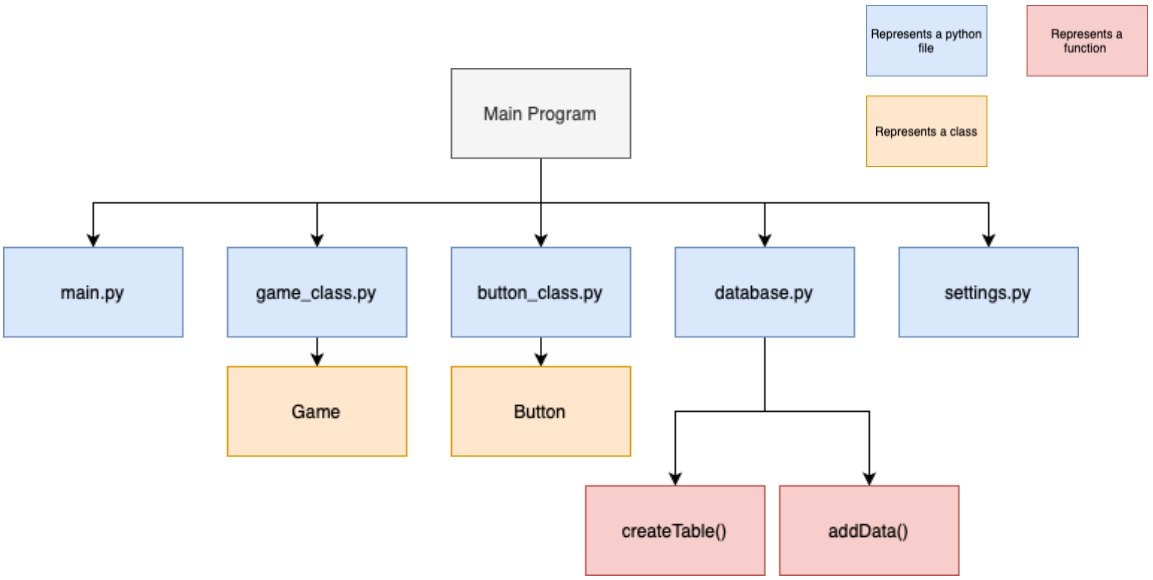
SYSTEM OVERVIEW

This section includes a general system overview for how the system will function. I've done this using different diagrams to demonstrate how data will flow and how different parts of the program interact and communicate with each other.

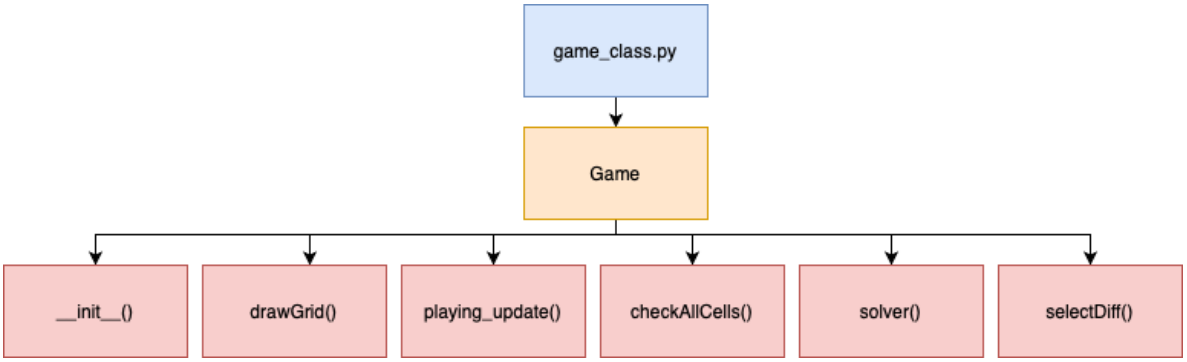
Hierarchy Charts

I've created different hierarchy chart to show a brief overview of the program and how the system interact with each other. There's a hierarchy chart for each python file I work with. I've gone into further detail in certain functions I felt was necessary.

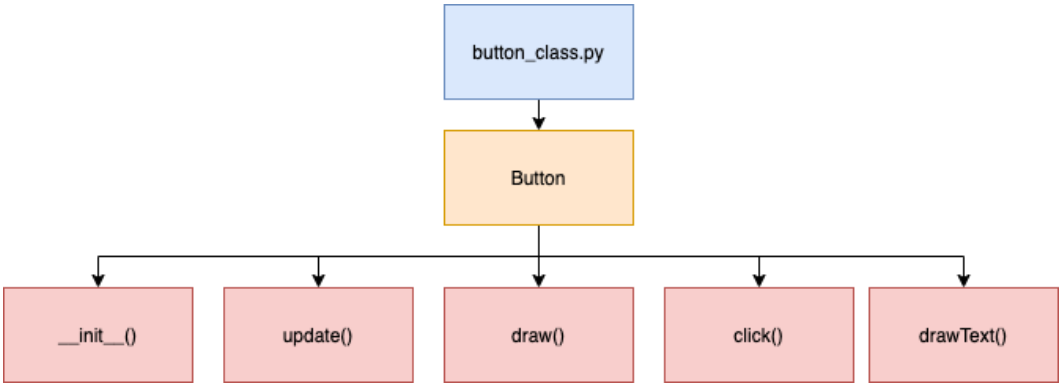
Here's the hierarchy chart for the main program. Below is a key which'll represent what each colour of the rectangle correlates to. I've created a file for different sections of the code, which is divided further into classes and functions. The game and button classes consist of more functions which will be shown in more detail below. You may notice that the database file will include functions for creating the table and adding data. It's important to note that this is just a simplified model of how the final code will be.



The game_class file can be split into 5 sections: playing state functions, board checking functions, helper functions, solver functions and database functions. Playing state functions will be functions used for the gameplay, board checking functions are functions used to check for incorrect cells and helper functions will be those that include tasks which work alongside the other functions that have been previously mentioned, for example a drawGrid() function will draw the playing board, a playing_update() function which updates the GUI during gameplay and a checking function to see if cells are correct. The hierarchy chart below shows a simplified version of how the game class may actually be.

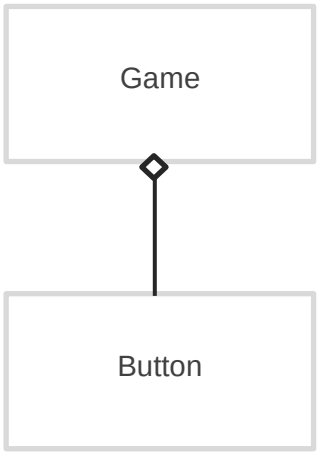


The button_class file includes the Button class which is divided into functions that'll contribute to making the buttons in charge of changing difficulty, solving and checking the puzzle at the end. Thus, the main functions of the button class will include drawing the buttons onto the GUI, drawing text which will appear on the buttons, being aware of when the user clicks on the button and updating the system, when a button is clicked.

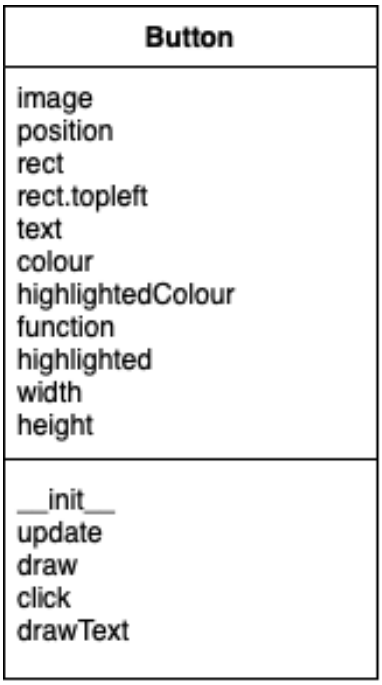


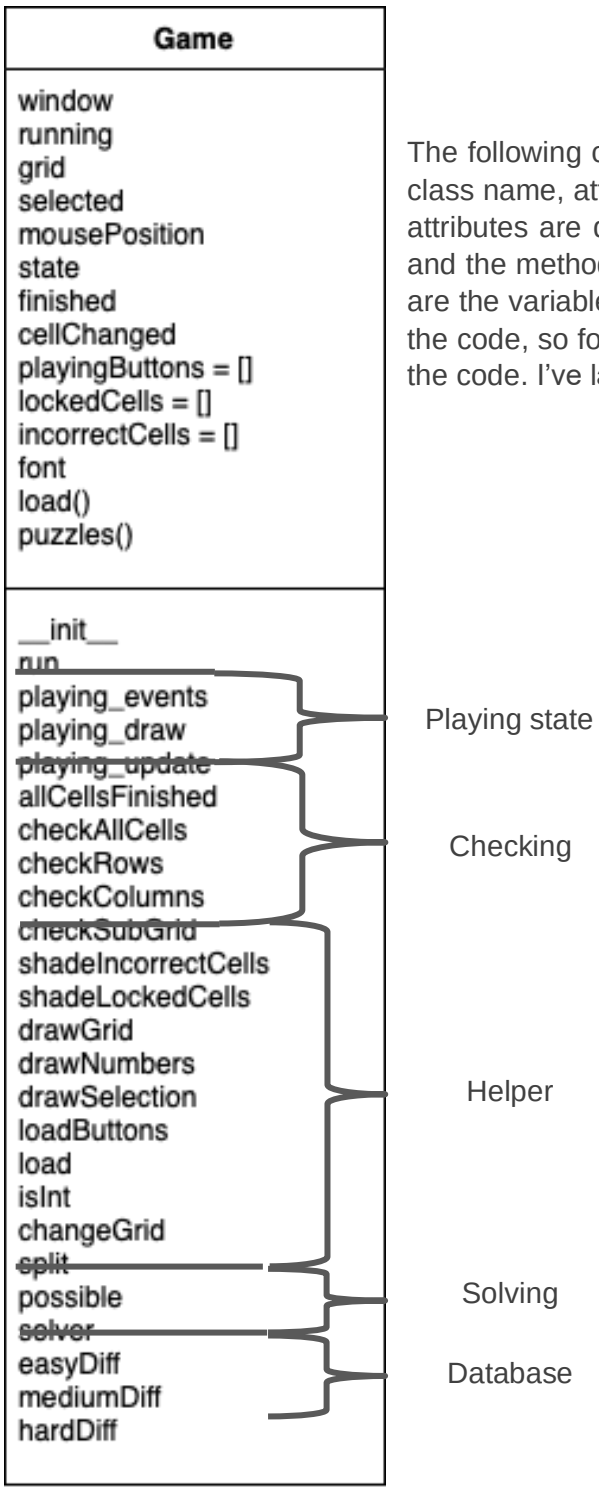
Class Diagrams

There'll be two classes in the program: the game class and the button class. The two classes are associated by aggregation, which is represented by the diamond headed arrow.



The class diagram below is for the button class. It shows the class name, attributes and methods belonging to the class. The attributes are displayed in the upper part of the class diagram and the methods are displayed in the lower part.





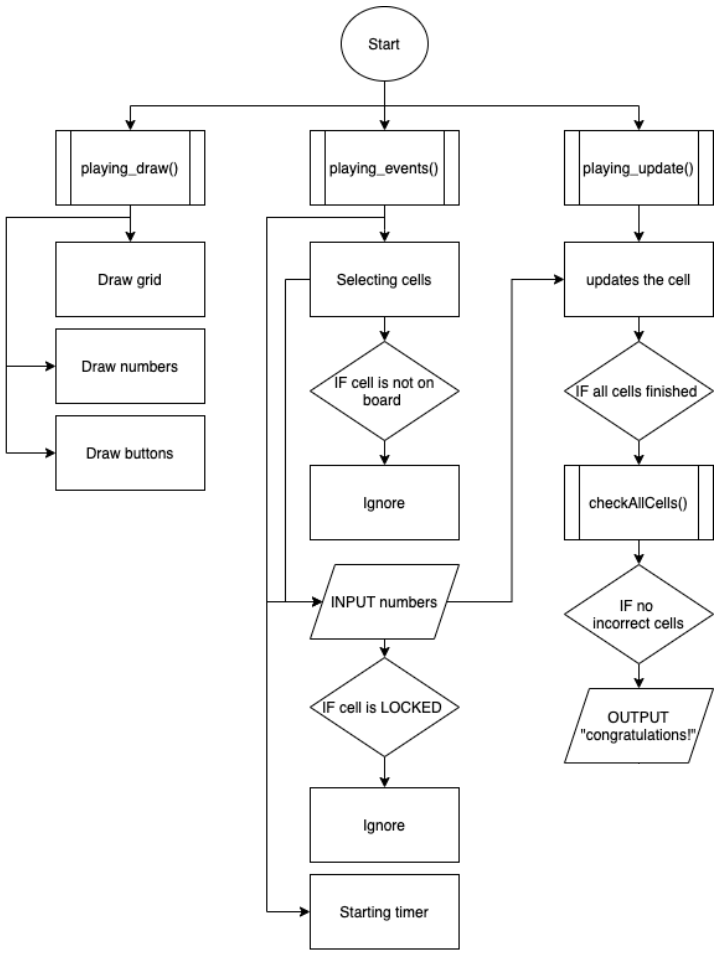
The following class diagram is for the game class. It shows the class name, attributes and methods belonging to the class. The attributes are displayed in the upper part of the class diagram and the methods are displayed in the lower part. The attributes are the variables used in the button class that begin with self in the code, so for example the attribute running is self.running in the code. I've labelled the methods into sections shown below.

Program Flowcharts

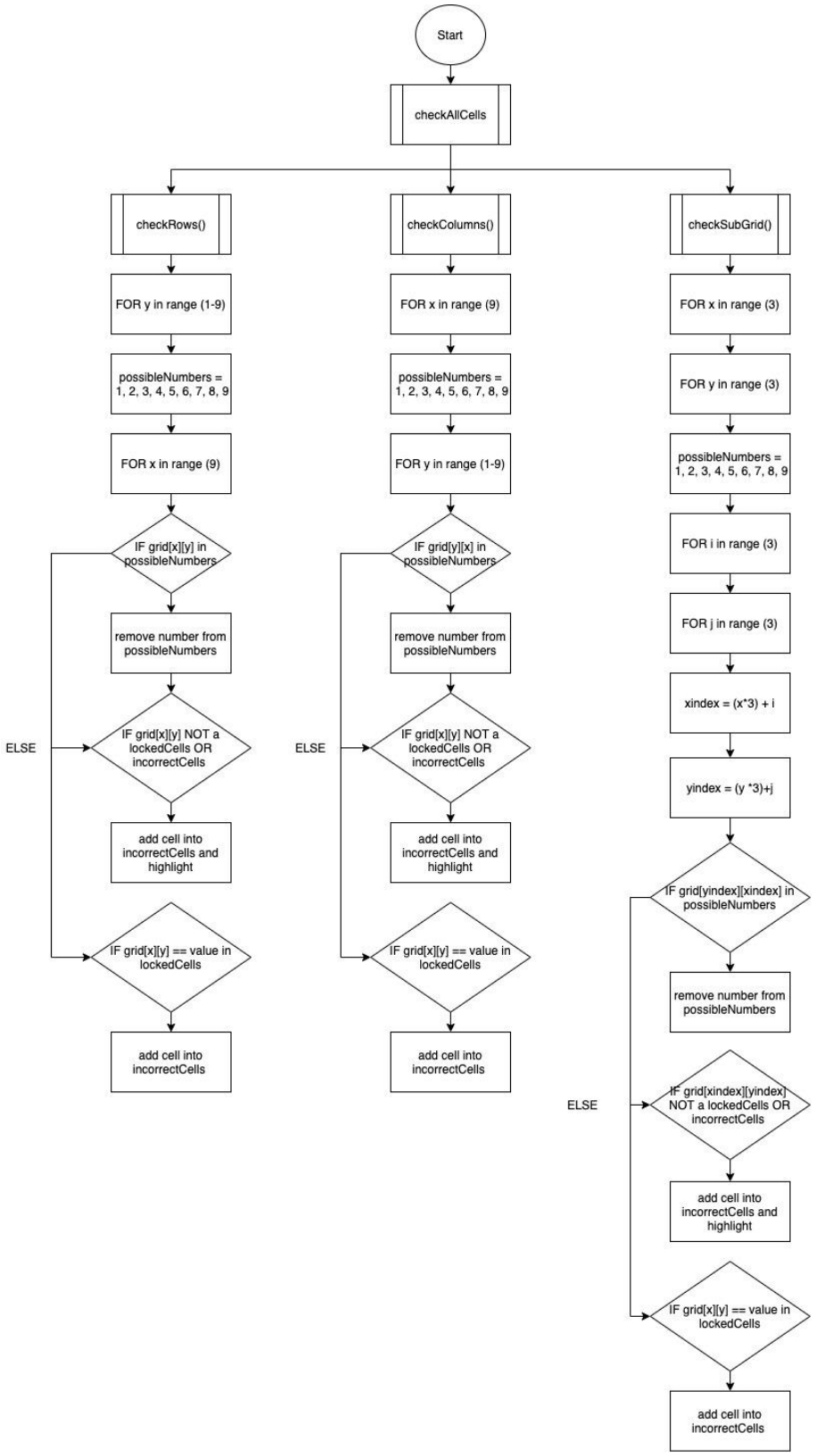
I've created a few flowcharts to break a few different processes for my Sudoku programme. This should help with understanding the logic behind a few functions that will be implemented.

Playing State Functions

The first flowchart above shows what happens when the game is loaded up and as it is played. As we can see, there are 3 main subroutines which will be in charge of these operations. These can be seen as the main elements responsible for the gameplay. Following the logic will help understand how the game will run.



Board Checking Function

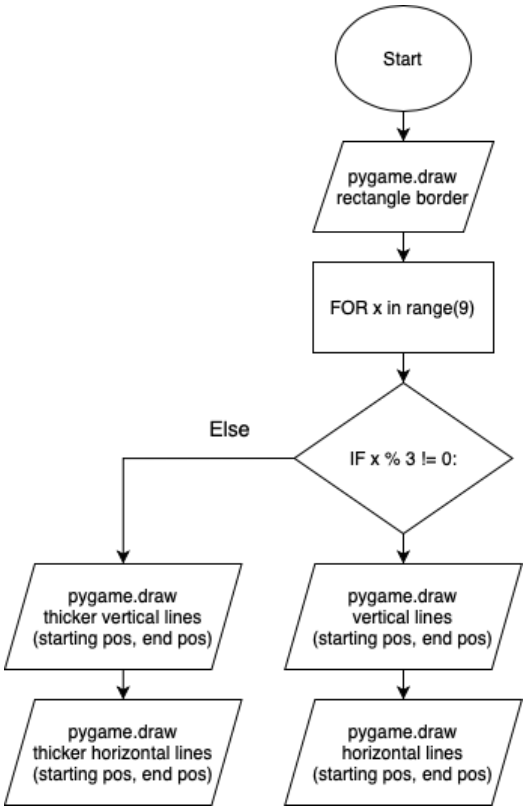


The flowchart above shows how grids will be checked using the function ‘checkAllCells’. When it is called, it will utilise functions, ‘checkRows’, checkColumns’ and ‘checkSubGrid’. For each function, it will make assign a variable, ‘possibleNumbers’ with numbers 1-9, as each subsequent row, column and subgrid will have only one occurence of the numbers.

The functions check each cell for numbers to ensure that they’re not repeated by removing numbers that appear from the possibleNumbers list; if a number appears that isn’t in the possibleNumbers list, it would label the corresponding cell as an ‘incorrectCell’ and upon using the checking button, would highlight it as incorrect. Furthermore, as cells that have already appeared in the grid will appear in array ‘lockedCells’, by checking if a value in a cell has the same value as a locked cell, it can identify it as another incorrect cell.

Draw Grid Function

To begin with, I’ll need to draw the square border of the grid. Then, I will draw nine verticle and horizontal lines, in their correct positions, using the for loop as an index for the coordinates, in order to produce the grid properly. For each third line I then draw over these lines again, but this time with a thicker line, so that you can distinguish between each sub-grid better, and to create a more visually appealing grid, satsifying the objective, “*1.3. Create a simplistic and visually appealing GUI*”.



LIBRARIES TO BE USED

Here are the list of Python libraries I will be using for my program.

Pygame

I will be using Pygame in order to produce the GUI for the game, so this includes displaying the main grid, buttons for variable difficulties and user interactions within the board, e.g., inputting numbers. Since this isn't a built-in library for Python, I must import it first. On the other hand, Pygame has its own built-in functions which allows users to perform certain tasks, therefore it must be initialised before getting access to it. Some of these include: 'pygame.draw', 'pygame.display' and 'pygame.colour', all of which I'm likely to use for my program.

SQLite3

SQLite is used to perform operations on databases; it is a pre-built library within Python. Tasks I will be doing will include creating databases, reading different tables, writing new records and tables, adding data to tables, etc. To use SQLite, you must first establish a connection to the database, using a variable that will represent the database. You can then create a cursor object and use its execute method to perform commands in SQL. You can still use databases created using SQLite with other software, like Microsoft Access, since it produces a file extension that ends with '.db', which is used widely for databases.

System

Another pre-built library in Python, the sys module provides us with new functions and variables which can be used to manipulate different parts of the Python runtime environment. In my use, it will mainly be to use module 'sys.exit', which is a Python exit command that allows the script to exit back to either the Python console or the command prompt. It is generally a safe way to exit the program in case of an exception being thrown.

ALGORITHM TO BE USED

Backtracking

As previously mentioned, backtracking is a type of recursive algorithm, which will test each possible path (refer to figure 4) until a solution is found. As it is uses recursions, this means that the function will keep calling itself until a condition has been satisfied; the condition for Sudoku will be until each row, column and sub-grid is filled with a single occurrence of numbers 1-9.

After assessing other algorithms and considering the reliability and consistency of backtracking, I believe backtracking is the most suitable algorithm for my project. Even the primary disadvantage of speed is not as drastic as initially thought, upon researching past implementations of backtracking on YouTube. Because of this, I see no reason not to include it.

I've created a design for the pseudocode for a function which is designed specifically to solve the Sudoku puzzle using backtracking. Note that 'grid' is a global variable which stores the grid to be solved. Also, note that if the grid has no feasible solution, the function will return false.

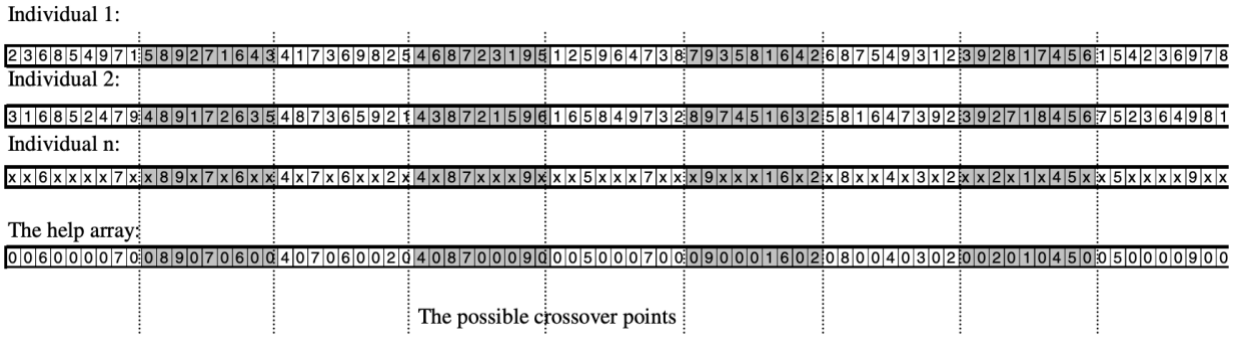
```
FUNCTION solve_backtrack()
FOR column in range (9):
        FOR row in range (9):
                IF grid[column][row] is empty:
                        FOR each number in range (1-10):
                                WHILE the board is solvable:
                                        grid[column][row] = number
                                        solve_backtrack()
                                        IF solve_backtrack() and board is valid:
                                                RETURN TRUE
                                        ELSE:
                                                BACKTRACK
                        IF there're no solutions:
                                RETURN FALSE
PRINT grid
```

Stochastic Optimization

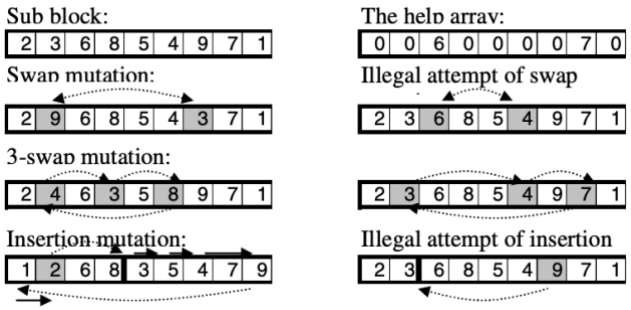
As mentioned in analysis, Sudoku can also be solved used stochastic optimization methods, namely using a genetic algorithm. Genetic algorithms are inspired by biology, where the computer replicates nature by using the concept of 'survival of the fittest'.

While it is possible to implement them as a Sudoku solver, a research paper[19] by professors from the University of Vassa have demonstrated that it isn't the most effective method to solve Sudoku puzzles. In their investigation, they studied how efficient the genetic algorithm is for solving Sudoku puzzles, and also if it can be used to create Sudoku puzzles and test their difficulty levels.

The Sudoku puzzle is stored as an integer array of 81 numbers. The array is then divided into nine sub-blocks, which contain nine numbers – corresponds to the sub-grids in Sudoku. Crossover is applied so that it exchanges whole sub blocks of nine numbers between individuals. Therefore, the crossover point cannot be inside a building block. For each array, there will be static numbers already present in the Sudoku puzzle, therefore changing them is considered an 'illegal attempt'. These static numbers are stored in a 'help array'.[16]



They implement three mutation strategies: swap mutation, 3 swap mutation and insertion mutation. Mutations are applied to randomly select positions inside each sub block, and then compare the positions to the help array to see if the attempt is 'legal'. If it is illegal, the process is repeated until legal positions are found. [16]



The following checking functions are used:

[19] "*Solving, rating and generating Sudoku puzzles with GA"* – ResearchGate, Timo Mantere, Janne Koljonen, October 2007, accessed: 18/10/20. https://seco.cs.aalto.fi/publications/2006/hyvonen-et-al-developments-in-artificial-intelligence-and-the-semantic-web-step-2006.pdf#page=91

As each row, column and sub-grid must contain only one occurrence of numbers 1-9, the equations 1-2 are used to check whether this condition is met. Equation 1 is ensuring that the sum of the numbers for each row/ column is equal to 45, and equation 2 is comparing the values to 9! A 3rd equation is used which compares each row x_i and column x_j to check numbers equal to set A. There're two lists: g_i and g_j; if a number appears it will remove It from set A. As long as number doesn't appear more than once in the list, then the sudoku has been solved correctly. When the board is solved correctly, for all three equations should be equal to zero.

$$g_{i1}(x) = \left|45 - \sum_{j=1}^{9} x_{i,j}\right| \qquad g_{i2}(x) = \left|9! - \prod_{j=1}^{9} x_{i,j}\right| \qquad A = \{1,2,3,4,5,6,7,8,9\}$$

$$g_{i3}(x) = \left|A - x_i\right|$$

$$g_{j1}(x) = \left|45 - \sum_{i=1}^{9} x_{i,j}\right| \qquad g_{j2}(x) = \left|9! - \prod_{=1}^{9} x_{i,j}\right| \qquad g_{j3}(x) = \left|A - x_j\right|$$

(1) (2) (3)

These equations are unanimous to Sudoku, and I'm planning on implementing at least one of these myself, as a means for checking that the board is completed correctly.

During their implementation, they used a population size of 100, elitism 40 and overall mutation probability of 0.12. The stopping condition was that the optimal solution had been found, or a maximum of 100,000 generations had been created.

Difficulty rating	*Givens*	*Count*	*Min*	*Max*	*Average*	*Median*	*Stdev*
New	0	100	206	3824	1390.4	1089	944.67
1 star	33	100	184	23993	2466.6	917	3500.98
2 stars	30	69	733	56484	11226.8	7034	11834.68
3 stars	28	46	678	94792	22346.4	14827	24846.46
4 stars	28	26	381	68253	22611.3	22297	22429.12
5 stars	30	23	756	68991	23288.0	17365	22732.25
Easy	36	100	101	6035	768.6	417	942.23
Challenging	25	30	1771	89070	25333.3	17755	23058.94
Difficult	23	4	18999	46814	20534.3	26162	12506.72
Super difficult	22	6	3022	47352	14392	6722	17053.33

Upon testing their model on several difficulty puzzles, they discovered the average and median GA generations needed to find solution increased. Therefore, there was a positive correlation between time taken for a genetic algorithm to solve a puzzle and the difficulty of the puzzle. This means that genetic algorithms can be used as a method of measuring the difficulty of Sudoku puzzles. Looking at figure 5, you can observe this trend.

The results of the test showed that the GA could only solve difficult puzzles a limited number of times. For instance, 4 stars was solved 26/ 100 test runs, 5 stars was solved 23/ 100 test runs and both difficult and super difficult were “so difficult that GA found the solution only a few times”. Since the primary function of the genetic algorithm in my project would be to solve puzzles, it doesn’t seem to be feasible option, as I want the solver to be able to solve all puzzles and consistently as well, regardless of difficulty.

Nonetheless, the paper concludes that Sudoku can be a viable option for a puzzle generator and can also measure difficulty of each puzzle, thus, in the future, if I wanted to develop my program further, I may implement genetic algorithms in this manner.

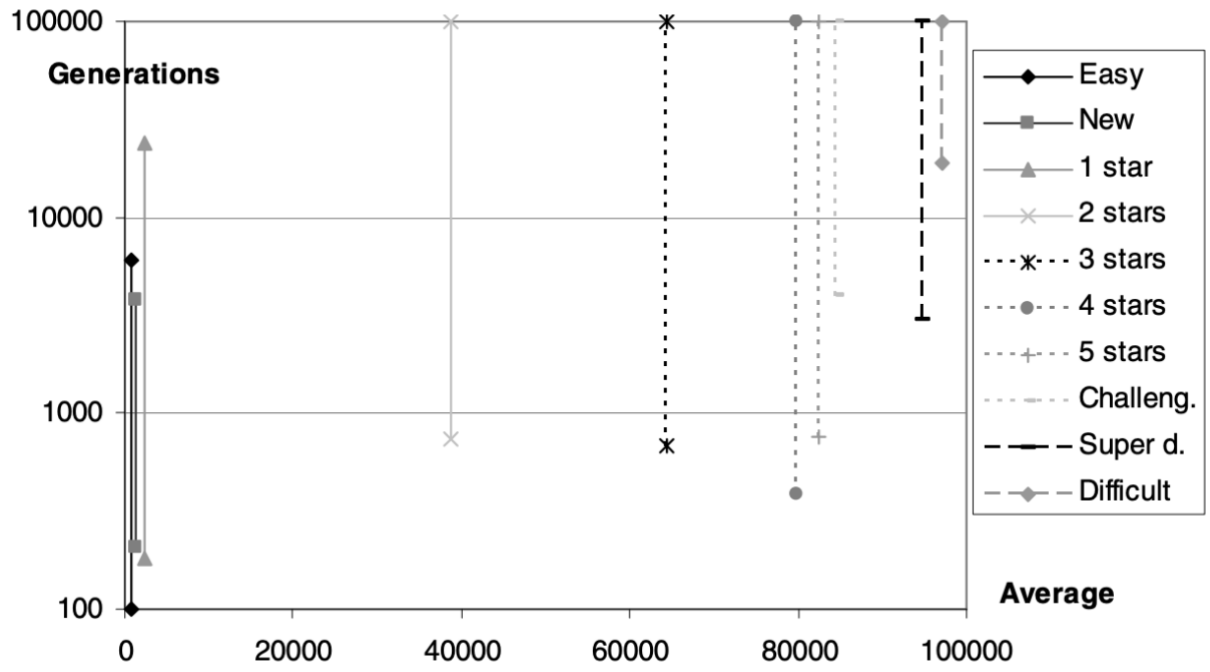


Figure 5 A comparison of the maximum, minimum and average generations needed to solve each difficulty level[16]

DATA STRUCTURES

In this section, I'm going to discuss the data structures I will be incorporating in my program, as well as going into the structure of the database and how I've organised my files.

A 2-dimensional array is going to be used to store the sudoku puzzle. The format of this is shown below, where each 1-dimensional array inside represents a row in the square. Therefore, each number in the 1-D array will have a unique occurrence of numbers 1-9. Furthermore, for every third number marks a sub-grid. You can indent the 2-D array in a way that looks fairly similar to the actual grid, in which you can notice that each ordinal number in the array marks a column; thus, for each 3rd number in each 1-D array, the numbers should have a single occurrence of each number between 1-9. It's necessary for it to be an array, since arrays are a type of static data structure, so are fixed in size. As the grid is a 9x9 matrix, I don't want the size of the data structure to be changing.

[[5,0,0,3,0,0,0,0,0], [0,2,0,0,8,0,0,0,0], [0,0,0,0,9,0,7,0,2], [4,0,0,6,0,0,0,0,0], [0,5,0,0,2,0,0,0,0], [0,1,0,0,0,0,0,8,6], [2,6,0,4,7,0,5,9,8], [0,4,0,9,0,0,1,0,3], [3,0,0,0,5,0,0,6,7]]

However, since I'm going to be using SQLite, I will be running into a problem, where SQLite does not support array as a data type, since arrays are a composite data type. The list of data types that it supports include:

- **NULL** – a blank value
- **TEXT** – a string stored using database encoding, e.g. UTF-8
- **INTEGER** – a signed integer
- **REAL** – a floating point value
- **BLOB** – a blob of data stored exactly as it was inputted

After carefully evaluating my options, I decided what would be best is to create a split/ join function that converts the array into a string which SQLite can process, and then when accessing the puzzles from the database to use it in the main program, converting back into its original 2-D array form, which my program can process better.

Pseudocode for the **join function** would be the following:

```
grid = 2-D array where grid is stored
string = “”
FUNCTION join(string):
        FOR each listNumber in range (9):
            FOR index in range (9):
                string = string + str(grid[listNumber][index])
```

Pseudocode for the **split function** would be the following:

```
splitString = []
newGrid = []
FUNCTION split():
    FOR each Character in range len(string):
        APPEND to splitString(int(string[Character]))
    FOR index in range(0, 81, 9):
        APPEND to newGrid(splitString[index: index+9])
```

splitString and newGrid are both empty lists to begin with, but the first for loop adds each character which represents a number in the original string into splitString, and the second for loop creates lists within the list in steps of 9 to ensure that it’s formatted at the end correctly.

Database Structure

Before creating the table in SQLite, I will create the table using a spreadsheet software, like Excel, and save it as a csv file. A csv file is a list of data that's separated by commas. This makes it easy to import and create the subsequent table in SQLite.

The appearance of the table is demonstrated below. There's a column for the ID of the puzzle, the level and the data itself. The ID column is the unique identifier, also known as the primary key. I will be using 30 puzzles; 10 of them are easy, 10 are medium difficulty and 10 are hard difficulty. This is also indicated by the level column where the character represents the subsequent difficulty. The data column stores the puzzle in its string format. For instance, take the 2-D array above, it would be stored like so.

"540060078306180020092004500609000705710496000408003100000030900180000257950027000"

Then, to be able to use the string to create a grid in my program, I would use the split function which returns it back into a 2-D array. Note that ID is stored as an integer and Level is a string.

SudokuTable		
ID	**Level**	**Data**
1-10	e	-
11-20	m	-
21-30	h	-

SQL Commands

Some of the SQLite commands I will be using include the following:

CREATE

I'm going to be using the CREATE command in SQLite to create the table called 'SudokuTable' in the database. The command requires us to specify the name of the table, its key attributes (ID, Level and Data) and their data types (integer and string). Pseudocode for this is shown.

```
CREATE TABLE SudokuTable (ID INT(225), Level STRING,
Data STRING, PRIMARY KEY (ID))
```

INSERT

Once the table is created, it's initially an empty table, thus I will need to insert the data to the database accordingly. To add records I will use the INSERT command in SQLite. For example, the pseudocode shows what it may look like, where values 'id', 'level' and 'data' are inserted into the subsequent columns and are extracted from the csv file.

```
INSERT INTO SudokuTable (ID, Level, Data) VALUES (id, level, data)
```

SELECT

When I want to choose a random puzzle that's of easy difficulty, I can achieve this by considering that, since easy puzzles have IDs within the range of 1-10, if I generate a random number from 1-10 and select the data for that from the table, I can complete this task. This will play a role in selecting puzzles of varying difficulties later on. Pseudocode for this is shown.

```
num = random.randint(1, 10)

SELECT Data FROM SudokuTable WHERE ID = num
```

Normalisation

After reviewing my table created, I noticed that I could normalise my data. Normalising my database would mean that it would allow for faster searching and sorting that an unnormalized database thanks to the smaller tables created in the normalisation process.

Normalisation typically carries with it three levels, and in particular, I noticed I could normalise my data to second normal form. In order to meet the second normal form requirements, a database must also satisfy the first normal form and have partial key dependencies removed. Partial key dependencies in databases are essentially where a non-key attribute doesn't depend on the whole of the composite key.

For my table I noticed that there was a repetition of data in the level column. Thus, I created separate tables for each difficulty. Level would be the **non-key attribute**, as it cannot determine what ID and data is. Moreover, each ID has its own unique data, thus, they're dependent on each other. Below I've created tables under second normal form.

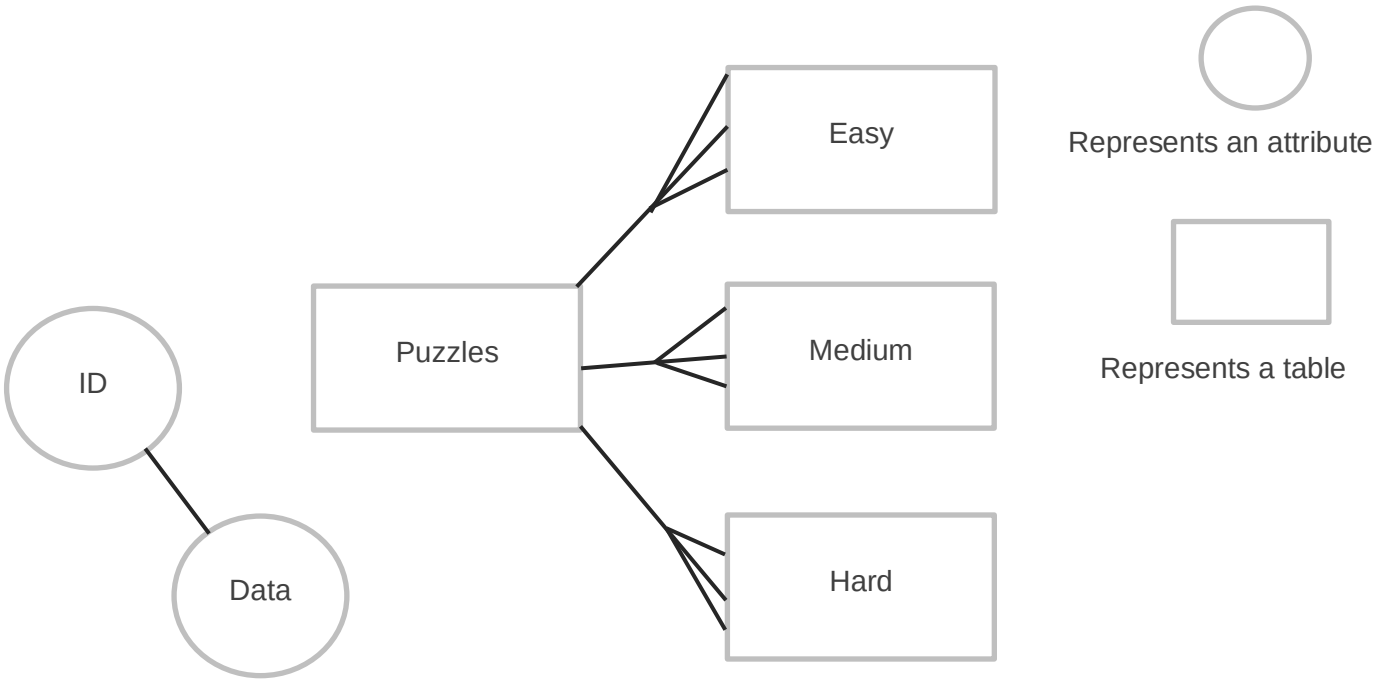
Easy	
ID	**Data**
1	-
2	-
…	-
10	-

Medium	
ID	**Data**
11	-
12	-
…	-
20	-

Hard	
ID	**Data**
21	-
22	-
…	-
30	-

Puzzles	
ID	**Data**
1	-
2	-
3	-
…	-
30	-

I've created entity relationship diagrams for you to visualise the relationship between each table in the database. Puzzles table can have many easy, medium and hard difficulty puzzles. But each ID has its own data.



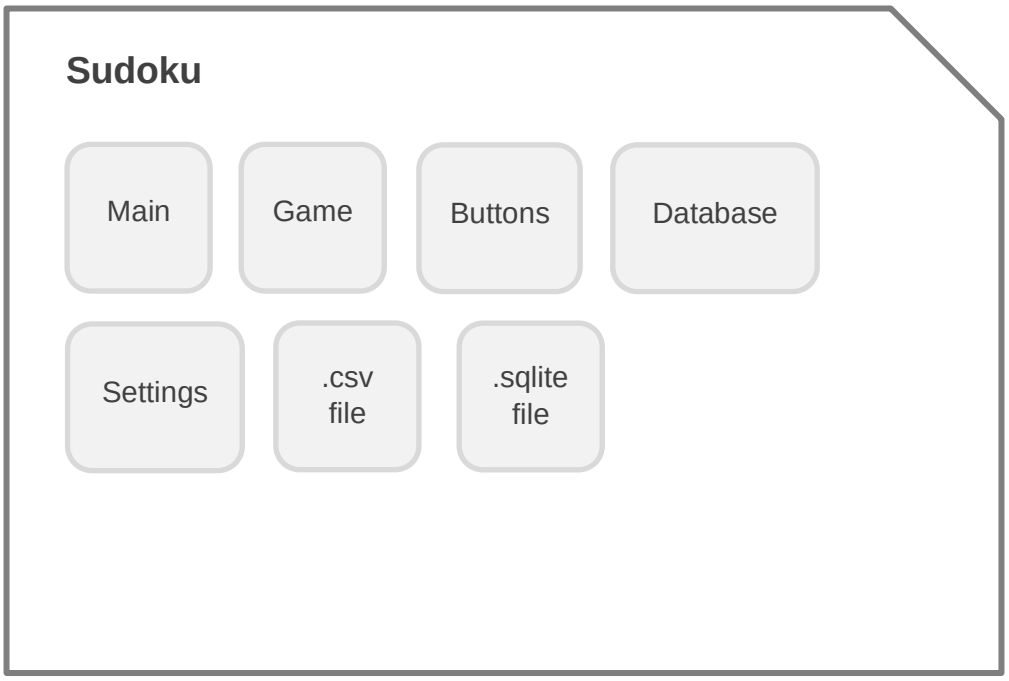
I've created pseudocode for how I plan to program how to read files from each table. Below I've included pseudocode for choosing a puzzle from the easy table, but all difficulties will follow the same format. First you choose from the data column of easy a value which is randomised. Then assign this to a variable, e.g. string. You then split the string back into a 2-D array and assign this value to the grid and update it, so that the new grid is displayed.

```
FUNCTION easyDifficulty():
    Try:
        SELECT Data from 'Easy'
        ORDER TABLE RANDOMLY
        LIMIT 1
    Except:
        OUTPUT "Error occurred selecting data"
    string = DATA FETCHED
    split(string)
    UPDATE GRID
```

File Structure and Organisation

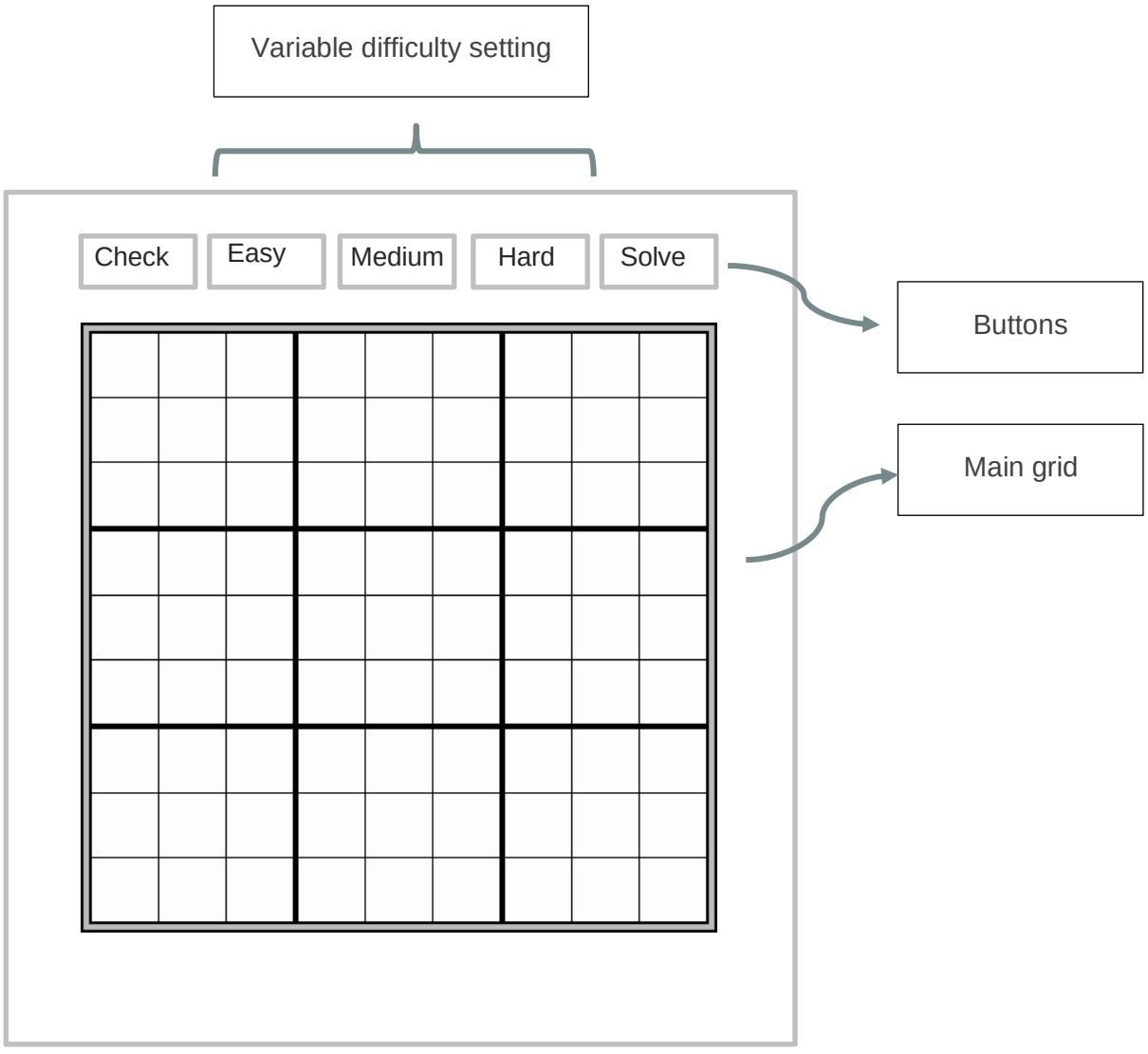
I'm going to be creating my project in a single folder called 'Sudoku'. The program will be split into a couple of different python files which each have their own purposes, but in the end all function work together to create the program. There's also going to be the database file and the csv file which is going to be used for the database aspect of my program.

It's important to keep all the files in a single folder to make sure that all files work properly together, since they depend on one another.



GUI DESIGN

As my client has said to stick with a simple design, I've done exactly that. When I asked how important the appearance of the GUI was in the survey, the average response came out at a 3/10. Thus, I've decided to emphasise functionality and ease of use over unnecessary flashy features. Nonetheless, this doesn't mean I won't be incorporating colour to boost the overall appearance of the GUI. The GUI will be created using Pygame. Pygame is good at creating a working GUI with a minimal amount of code as it contains built in functions, which'll also help with creating the GUI. Below, I've labelled key features of the GUI.



TESTING

This section shows how I will be planning to conduct tests to check that my program is functioning properly. Video tests will be done at the end to record my program in action. This is to demonstrate whether I have achieved my objectives, like trying to play the game normally and capturing the solver function for a range of puzzles.

Test 1

The first test will be on the **GUI**. The GUI will be where most of the user interaction will be, thus as a result, a lot of my testing will be focused on the GUI. I'm going to test the following:

Test Number	Test	Objectives Tested
1.1	Testing if the board is produced correctly	1.1 Display the main playing board using Pygame
1.2	Testing if the 'locked cells' are shaded	1.3 Create a simplistic and visually appealing GUI 1.4 Make use of colour in the GUI
1.3	Testing if buttons appear	1.2 Display buttons which have a range of uses, including checking the board and changing the difficulty of the board 1.3 Create a simplistic and visually appealing GUI
1.4	Testing if incorrect cells are shaded	1.3 Create a simplistic and visually appealing GUI 1.4 Make use of colour in the GUI
1.5	Testing if selected cells are shaded	1.3 Create a simplistic and visually appealing GUI 1.4 Make use of colour in the GUI
1.6	Testing if numbers are on the board	1.1 Display the main playing board using Pygame
1.7	Testing if buttons are highlighted when mouse is above it	1.2 Display buttons which have a range of uses, including checking the board and changing the difficulty of the board 1.3 Create a simplistic and visually appealing GUI 1.4 Make use of colour in the GUI

Test 2

The second set of tests relate to the **gameplay** of the program. I'm going to test the following:

Test Number	Test	Objective
2.1	Testing if users can input numbers	2.1 Should be able to play the game like a normal Sudoku
2.2	Testing if users cannot input non-numeric characters	2.3 Shouldn't be able to enter non-numeric values into the cell
2.3	Testing that user cannot change a locked cell	2.4 Shouldn't be able to change cells that are in the grid before user starts
2.4	Testing that buttons work	2.2 Should include buttons which have unique features, which are checking the puzzle, choosing a difficulty setting for the puzzle and solving
2.5	Testing what happens when a user completes the game	2.1 Should be able to play the game like a normal Sudoku

Test 3

I will also be conducting tests on the **backtracking recursive algorithm**, ensuring that it works at all times. A range of different puzzles will be tested from all kinds of difficulties. I will be recording the time for each puzzle to see if difficulty has an effect on the time taken to solve; however, if it's been implemented correctly, this should not have a significant effect.

Test Number	Test	Objective
3.1	Testing if backtracking algorithm works	3.1 Use recursion to solve the puzzle
3.2	Testing if backtracking algorithm works on puzzles of all difficulties	3.2 It should be able to solve any puzzle
3.3	Testing backtracking algorithm run-time is consistent	3.3 The time to solve each puzzle should be consistent

Test 4

The final test will be on the **database** which stores the puzzles being used by the program.

Test Number	Test	Objective
4.1	Testing boards are stored properly	4.1 Successfully store puzzles in a database
4.2	Testing if users can choose different difficulty boards	4.2 Access the database when a difficulty is selected and produce a board from that subsequent difficulty

Technical Solution

In this section, I will be exploring and breaking down the code I've written. The code has been split into different files, of which I will be showing the game class file, the button class file, the database file and the settings file. Do note that the database functions are split across the game class file and the database file. Each section includes screenshots of different snippets of code, with explanations underneath each bit explaining what is going on.

TABLE OF SKILLS USED

This table includes some of the technical skills I've used.

Skill used	Objective Achieved	Line number in code
Complex user defined use of Object-Oriented Programming (OOP)	1.1 Display the main playing board using Pygame 1.2 Display buttons which have a range of uses, including checking the board and changing the difficulty of the board 2.1 Should be able to play the game like a normal Sudoku	gameClass.py line 14 buttonClass.py line 7 main.py line 4
Exception handling	2.1 Should be able to play the game like a normal Sudoku 2. 3 Shouldn't be able to enter non-numeric values into the cell 2.4 Shouldn't be able to change cells that are in the grid before user starts	gameClass.py line 299, 375, 391, 408, 424 database.py line 28, 44, 59, 74
Files organised for direct access	4.2 Successfully store puzzles in a database 1.1 Display the main playing board using Pygame 1.2 Display buttons which have a range of uses, including checking the board and changing the difficulty of the board	main.py line 1 gameClass.py line 4 database.py line 3, 10
Recursive algorithms Graph/ Tree Traversal	3.2 Use recursion to solve the puzzle 3.2 It should be able to solve any puzzle 3.3 The time to solve each puzzle should be consistent	gameClass.py line 334, 350

Multi-dimensional arrays Lists List operations	1.1 Display the main playing board using Pygame 4.2 Successfully store puzzles in a database 1.2 Display buttons which have a range of uses, including checking the board and changing the difficulty of the board	gameClass.py line 74, 133, 151, 169, 284, 312, 375, 391, 407, 423 database.py line 13, 17, 18, 19, 20 settings.py line 10, 28
Simple mathematical modelling	1.1 Should be able to play the game like a normal Sudoku 1.2 Display buttons which have a range of uses, including checking the board and changing the difficulty of the board	gameClass.py line 133, 151, 169
Simple data model in database Normalisation	4.2 Successfully store puzzles in a database 4.2 Access the database when a difficulty is selected and produce a board from that subsequent difficulty	database.py line 26, 42, 57, 72,
Text files Writing and reading from files	4.2 Successfully store puzzles in a database 4.2 Access the database when a difficulty is selected and produce a board from that subsequent difficulty 1.1 Display the main playing board using Pygame	gameClass.py line 375, 391, 407, 423 database.py line 91, 107, 122, 137

GAME CLASS

The game class is where most of the action in the program occurs. It contains several methods that contribute to a range of different functions in the program, from producing the board to applying the backtracking algorithm. To organise all the methods being used, you can split the different functions in the game class into five sections: playing state functions, board checking functions, helper functions, solver functions and database functions. Each are described in more detail in their subsequent sections.

Explanation of __init__()

```
class Game():

    def __init__(self):
        pygame.init()
        self.window = pygame.display.set_mode((WIDTH, HEIGHT)) # Pygame window
        self.running = True
        self.grid = grid # default grid
        self.splitString = [] # variables for split
        self.newGrid = []  # variables for split
        self.selected = None # indicates if cell is selected
        self.mousePosition = None # returns position of mouse
        self.state = "playing" # default state
        self.finished = False
        self.cellChanged = False
        self.playingButtons = [] # list that holds all buttons for playing state
        self.lockedCells = [] # list that holds all locked cells
        self.incorrectCells = [] # list that holds all incorrectly played cells
        self.font = pygame.font.SysFont("arial", cellSize // 2) # font to be used
        self.load() # initialises the game
        self.puzzles() # creates a fresh new board every time
```

The __init__() function is a method in python which is used as the constructor for the game class. It initialises the attributes of the class. As I'm using Pygame to create the GUI, I'll be utilising many of Pygame's functions.

To begin with, window refers to the GUI window created by Pygame; this is going to be used throughout the program to draw the board and its numbers. The game is set as running by default, and will be changed to false, if the game ends. Grid is simply the grid variable which is the board currently being played on. The properties splitString and newGrid are used by the split function and is initiated here to fix a bug. Splitting the Selected is used later on to indicate when a user clicks on a cell. Mouse position is used so that the game knows when the mouse is on

the board or not – shown in mouseOnGrid. When the game is running, the game's state is defined as playing. The finished variable is initially false by default, but it is changed to true, when all cells have been finished correcty. Initially, cellChanged is set to false, but once the user starts to play, it is changed to true. The playingButtons, lockedCells and incorrectCells variables are lists which are set as empty by default every time a new game starts. Font describes what kind of font is going to be used. The load variable is responsible for creating a 'fresh new' game by resetting the board. I call the puzzles function which loads a fresh new board each time the program is run. These variables are described in greater detail later on.

Explanation of run()

```
def run(self):
    # the game loop
    while self.running == True:
        if self.state == "playing":
            # calls the following functions which allows gameplay to work
            self.playingEvents()
            self.playingUpdate()
            self.playingDraw()
    # when game is no longer running
    pygame.quit()
    sys.exit()
```

When the program is run, the game is setup by performing the run function. It works while 'self.running' is true the game's state is defined as "playing" and calls the playing state functions which are described in detail below. When the game finishes self.running is no longer true and thus the game quits.

PLAYING STATE FUNCTIONS

The playing state functions include methods which are responsible for the gameplay aspect of the program. This includes normal features like entering a number into a square. Note that looking back at the flowcharts of the playing state functions in design will help understanding how the code has been written easier.

Explanation of playingEvent()

```
def playingEvents(self):
    # contains code which is in charge of general user gameplay
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            self.running = False

        # if the user clicks the mouse
        if event.type == pygame.MOUSEBUTTONDOWN:
            selected = self.mouseOnGrid()
            if selected:
                self.selected = (selected)  # returns position of mouse
            else:
                self.selected = None
                for button in self.playingButtons:
                    if button.highlighted:
                        # indicates that mouse is hovering over button
                        button.click()

        # if the user types a key
        if event.type == pygame.KEYDOWN:
            if self.selected != None and list(self.selected) not in self.lockedCells:
                if self.checkInt(event.unicode):
                    # changes the cell
                    self.grid[self.selected[1]][self.selected[0]] = int(event.unicode)
                    # indexes are in the self.selected attribute
                    self.cellChanged = True
```

The playing events functions contains code for the tasks that occur during the player's gameplay. Extending from the run function, if the event type was pygame.QUIT, then 'self.running' would be false and the game would end. If the user clicks the mouse, the area is indicated as being selected. If it's a cell, using the playingDraw function, it would highlight the cell. Otherwise, if it's a button and the user clicks on it, it too would be highlighted a colour. Moreover, if the user selects an empty cell and enters an integer then the cell is changed to that number. It must be an integer and goes through exception handling in function checkInt.

Explanation of playingUpdate()

```
def playingUpdate(self):
    # contains code which updates the board during the user's gameplay
    self.mousePosition = pygame.mouse.get_pos()
    for button in self.playingButtons:
        button.update(self.mousePosition)

    if self.cellChanged:
        # when a player makes a move
        self.incorrectCells = []
        if self.allCellsFilled():
            # checks if board is correct
            self.checkAllCells()
            if len(self.incorrectCells) == 0:
                print("Congratulations! \nYou've completed this sudoku!")
```

The role of the playing update is self-explanatory, it is responsible for updating the grid as the user plays. The position of the mouse is found using built-in functions from Pygame and is used in the button class, explained in further detail below, to highlight the buttons when the mouse is directly above it. Each time a cell is changed, the incorrectCells is reverted back to an empty list. If all cells have been finished (explanation of allCellsFinished below), then checkAllCells function is called, which checks each row, column and sub-grid and if there’re no incorrect cells, then a congratulations message is outputted to the user.

Explanation of playingDraw()

```
def playingDraw(self):
    # contains which sets up the GUI
    self.window.fill(WHITE)

    # adding buttons
    for button in self.playingButtons:
        button.draw(self.window)

    # when the user selects
    if self.selected:
        self.drawSelection(self.window, self.selected)

    # shading cells
    self.shadeLockedCells(self.window, self.lockedCells)
    self.shadeIncorrectCells(self.window, self.incorrectCells)

    # drawing the GUI
    self.drawNumbers(self.window)
    self.drawGrid(self.window)
    pygame.display.update()
    self.cellChanged = False
```

It essentially setups the GUI where the user will play. It displays the buttons, described later on in the button class, by calling the draw function from it. If the user clicks on a cell it's described as being selected so calls the drawSelection function, which highlights the subsequent cell. Moreover, to prevent confusion, it also highlights cells that were present before the user starts to make any moves, which are referred to as "locked cells". In addition, it also shades any cells that are incorrect. Furthermore, it calls the drawGrid function which actually produces the sudoku board and then draws the numbers on top of the grid using the drawNumbers function. Initially the cellChanged function is set to false, as at first no cells will be changed. Note that all these funcitons are described in greater depth in the helper functions section.

BOARD CHECKING FUNCTIONS

As the name suggests, these group of functions are responsible for checking that the sudoku board has been solved correctly. It's used by the button 'check'. The board is checked to see if it's correct by checking the rows, columns and then the sub-grid.

Explanation of allCellsFilled()

```
def allCellsFilled(self):
    # used to check to see if all cells in the grid have been filled
    for row in self.grid:
        for number in row:
            if number == 0:
                return False
    return True
```

This function is used to check that all cells have been filled. If a number in a cell states 0, it represents a blank space which means that not all cells have been filled. If it doesn't it reports true. When all cells are filled, the checkAllCells function is called in playingUpdate. If it has not been completed correctly, it will return false and highlight where the incorrect cells are.

Explanation of checkAllCells()

```
def checkAllCells(self):
    # used to check the grid
    self.checkRows()
    self.checkColumns()
    self.checkSubGrid()
```

To check all cells, I'm going to be using the maths mentioned in design. I will be checking each row, column and sub-grid to ensure there's only a single occurrence of each number from 1-9. For my program, I did this by checking that the sum of the numbers for each row/ column was equal to 45. This function is used in the button 'check'.

Explanation of checkRows()

```
def checkRows(self):  # used to check each row
    for yindex, row in enumerate(self.grid):
        possibleNumbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
        # the list of possible numbers that can be inputted
        for xindex in range(9):
            if self.grid[yindex][xindex] in possibleNumbers:
                possibleNumbers.remove(self.grid[yindex][xindex])
                # if a number already appears in the grid, it removes it from the list
            else:
                if [xindex, yindex] not in self.lockedCells and [xindex, yindex] not in self.incorrectCells:
                    self.incorrectCells.append([xindex, yindex])
                    # if the user makes an incorrect move, it's added to the incorrectCells list
                if [xindex, yindex] in self.lockedCells:
                    for i in range(9):
                        if self.grid[yindex][i] == self.grid[yindex][xindex]:
                            self.incorrectCells.append([i, yindex])
                            # if the user enters a locked cell, it's added to the incorrectCells list
```

Implementing the flowchart for the checkRows function in design, you can create the following piece of code. The index for the variable yindex is enumerated from self.grid, which is simply refers to the variable storing the puzzle. A list of the possible inputs is stored. If a number in the grid, prior to the user playing, appears within the list, it is removed. Thus, if a number in a cell is not in the possibleNumbers list, then it is appended to the list of incorrectCells. You also don't want to add an incorrect cell to the list twice, hence why I check that the cell is not already in self.incorrectCells. In addition, an extra checking step is included, due to an error which occurred that would cause some incorrect cells to not be detected. This is caused by a value in a locked cell not appearing in the possibleNumbers list, and since I cannot return an error for locked cells, I perform extra checks by looking at each row to see which cell is responsible for this and adding it to incorrectCells.

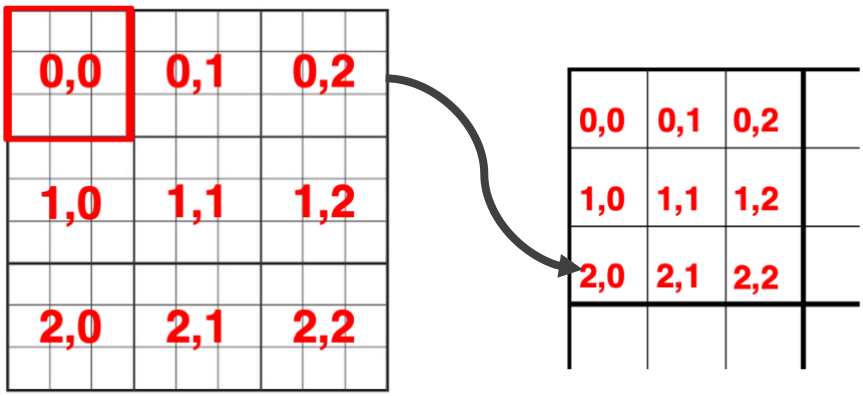
Explanation of checkColumns()

```
def checkColumns(self):  # used to check each column
    for xindex in range(9):
        possibleNumbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
        # the list of possible numbers that can be inputted
        for yindex, row in enumerate(self.grid):
            if self.grid[yindex][xindex] in possibleNumbers:
                possibleNumbers.remove(self.grid[yindex][xindex])
                # if a number already appears in the grid, it removes it from the list
            else:
                if [xindex, yindex] not in self.lockedCells and [xindex, yindex] not in self.incorrectCells:
                    self.incorrectCells.append([xindex, yindex])
                    # if the user makes an incorrect move, it's added to the incorrectCells list
                if [xindex, yindex] in self.lockedCells:
                    for i, row in enumerate(self.grid):
                        if self.grid[i][xindex] == self.grid[yindex][xindex]:
                            self.incorrectCells.append([xindex, i])
                            # if the user enters a locked cell, it's added to the incorrectCells list
```

Following the same principle as the checkRows() function, checkColumns is applied in a very similar manner. The only difference this time is the order in which the xindex and yindex is placed; so, this time I can see xindex comes before the list of possibleNumbers, ensuring that each column is checked correctly. The same extra checks are done at the end, as shown, to avoid any potential errors, but this time checking the columns.

Explanation of checkSubGrid()

Before moving onto the code, I'd like to discuss the logic behind checking sub-grids. Each grid can be broken down into 3 sub-grids for each row and column, which can then be broken down further into each cell. The diagram below shows the first sub-grid can be referred to as (0,0), and the one next to it is (0,1). Within each sub-grid, coordinates are (0,0), (0,1), (0,2), etc.



Once again, the code for checkSubGrid is similar to the checkRows and checkColumns functions, but with an extra step. At the beginning, the for loops are to check each of the sub-grids. The possible numbers are stored in a list. Within each for loop, there are another set of for loops (think of the coordinates of the cells within each sub-grid).

The indexes are calculated using (x * 3) + i, which a allows us to find the coordinates of each cell in a sub-grid, before moving on to the next sub-grid. If a number appears within the possibleNumbers list, it is removed; if it does not, this means that the number has already appeared, hence why it is no longer a 'possible number'. It will then add it to the list of incorrectCells, given it is not a locked cell and isn't already in the list. Once more, I perform extra checks at the end, but this time, for each cell in a sub-grid.

```
def checkSubGrid(self):  # used to check each subgrid in the board
    for x in range(3):
        for y in range(3):
            possibleNumbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
            # the list of possible numbers that can be inputted
            for i in range(3):
                for j in range(3):
                    # used to check each cell in each subgrid
                    xindex = x * 3 + i
                    yindex = y * 3 + j
                    if self.grid[yindex][xindex] in possibleNumbers:
                        possibleNumbers.remove(self.grid[yindex][xindex])
                        # if a number already appears in the grid, it removes it from the list
                    else:
                        if [xindex, yindex] not in self.incorrectCells and [xindex, yindex] not in self.lockedCells:
                            self.incorrectCells.append([xindex, yindex])
                            # if the user makes an incorrect move, it's added to the incorrectCells list
                        if [xindex, yindex] in self.lockedCells:
                            for k in range(3):
                                for l in range(3):
                                    xindex2 = (x * 3) + k
                                    yindex2 = (y * 3) + l
                                    if self.grid[yindex2][xindex2] == self.grid[yindex][xindex]:
                                        # extra measures caused by bug which wouldn't register some locked cells
                                        self.incorrectCells.append([xindex2, yindex2])
                                        # if the user enters a locked cell, it's added to the incorrectCells list
```

HELPER FUNCTIONS

These functions help support the other functions in the program and consolidate the loosely coupled style of programming I'm using. They each perform single operations given by the name of the function and make use of functions in the game class and button class as well.

Explanation of shadeIncorrectCells() and shadeLockedCells()

```
def shadeIncorrectCells(self, window, incorrect):
    # draws a coloured square to indicate the cell is incorrect
    for cell in incorrect:
        pygame.draw.rect(window, INCORRECTCELLCOLOUR, (
        cell[0] * cellSize + gridPosition[0], cell[1] * cellSize + gridPosition[1], cellSize, cellSize))

def shadeLockedCells(self, window, locked):
    # draws a coloured square to indicate the cell is locked
    for cell in locked:
        pygame.draw.rect(window, LOCKEDCELLCOLOUR, (
        cell[0] * cellSize + gridPosition[0], cell[1] * cellSize + gridPosition[1], cellSize, cellSize))
```

As one of my objectives is to provide a simple and elegant GUI, I've decided to improve the aesthetics of the GUI whilst also providing a useful feature by shading the locked cells and incorrect cells in the sudoku board. This can prove to be useful, as it will show which cells are wrong for the user and as the game nears its end, it will also highlight specifically what numbers need to be changed; I've shaded the locked cells so players can keep track of which cells they've inputted, and which ones were there to begin with. This is achieved using Pygame's library and the colours used in the program are in the settings file, which I've listed below.

```
# colours
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
GREEN = (0, 200, 0)
TEAL = (174, 227, 245)
LOCKEDCELLCOLOUR = (245, 245, 245)
INCORRECTCELLCOLOUR = (255, 150, 150)
```

Explanation of drawNumbers() and drawSelection()

```
def drawNumbers(self, window):
    for yindex, row in enumerate(self.grid):  # for each row in the grid
        for xindex, num in enumerate(row):  # for each cell in each row
            if num != 0:  # while cell is not blank
                position = [(xindex * cellSize) + gridPosition[0], (yindex * cellSize) + gridPosition[1]]
                # draws the numbers in relation to position of grid on the window
                self.textOnScreen(window, str(num), position)
                # calls textOnScreen to output numbers

def drawSelection(self, window, position):
    # draws a coloured square to indicate the cell has been selected
    pygame.draw.rect(window, TEAL, (
    (position[0] * cellSize) + gridPosition[0], (position[1] * cellSize) + gridPosition[1], cellSize, cellSize))
```

In the drawNumbers function I want to make sure I'm only adding numbers already given and not blank spaces, indicated by a 0, which are left ignored. In drawSelection, it shades a cell when the user clicks on it – this is referred to as selected cell. The colour is teal as indicated and comes from the settings file.

Explanation of drawGrid()

```
def drawGrid(self, window):
    # draws border square
    pygame.draw.rect(window, BLACK, (gridPosition[0], gridPosition[1], WIDTH - 150, HEIGHT - 150), 2)
    for x in range(9):
        if x % 3 != 0:  # for every line it draws lines, thickness 1
            # vertical lines
            pygame.draw.line(window, BLACK, (gridPosition[0] + (x * cellSize), gridPosition[1]),
                             (gridPosition[0] + (x * cellSize), gridPosition[1] + 450))
            # horizontal lines
            pygame.draw.line(window, BLACK, (gridPosition[0], gridPosition[1] + (x * cellSize)),
                             (gridPosition[0] + 450, gridPosition[1] + +(x * cellSize)))
        else: # for every third line it draws lines, thickness 2
            # vertical lines
            pygame.draw.line(window, BLACK, (gridPosition[0] + (x * cellSize), gridPosition[1]),
                             (gridPosition[0] + (x * cellSize), gridPosition[1] + 450), 2)
            # horizontal lines
            pygame.draw.line(window, BLACK, (gridPosition[0], gridPosition[1] + (x * cellSize)),
                             (gridPosition[0] + 450, gridPosition[1] + +(x * cellSize)), 2)
```

start position

end position

You may remember the flowchart of the drawGrid function, which will help with understanding the logic of the code here. In order to produce the main board for the GUI, the function above is used. Pygame's extensive library is made use of to draw each shape, as seen by 'pygame.draw'. You can observe that the code is mainly figuring out how to produce a square and putting it at the centre of the GUI. You may notice that some variables like 'WIDTH' and 'gridPosition' is used, which can be found in the settings file shown below. I draw nine verticle and horizontal lines, in their correct positions, using the for loop as an index for the coordinates, in order to produce the grid properly.

The first use of 'pygame.draw' produces the main square and the following is used to draw the lines which then forms the sub-grids and cells. When drawing the vertical and horizontal lines you can think of the brackets inside being the coordinates for where to draw each line. Moreover, every third line, I draw again but with a thicker line so that you can distinguish between sub-grids; the 2 at the end of each line indicates the thickness of the line.

```
WIDTH = 600
HEIGHT = 600

# positions and sizes
gridPosition = (75, 100)  # the top left of the grid
cellSize = 50
gridSize = cellSize * 9
```

Explanation of mouseOnGrid()

```
def mouseOnGrid(self):
    # if we click off the grid, it returns false
    if self.mousePosition[0] < gridPosition[0] or self.mousePosition[1] < gridPosition[1]:
        return False

    if self.mousePosition[0] > gridPosition[0] + gridSize or self.mousePosition[1] > gridPosition[1] + gridSize:
        return False

    # if we click on the grid, it returns the position of the mouse
    return (
    (self.mousePosition[0] - gridPosition[0]) // cellSize, (self.mousePosition[1] - gridPosition[1]) // cellSize)
```

If the mouse position is higher or more to the left than the grid, it returns false, to indicate that the mouse is not on the grid. Otherwise, if you click on the grid, it will return where the position of the mouse is. This is used to indicate if a user clicks on a cell in the grid in playingEvents.

Explanation of loadButtons()

```
def loadButtons(self):
    # initialising buttons for the GUI
    self.playingButtons.append(Button(75, 40, WIDTH // 8, 40, function=self.checkAllCells, colour=(210, 140, 220),
                                      text="Check"))  # check button

    self.playingButtons.append(
        Button(170, 40, WIDTH // 8, 40, function=self.easyDiff, colour=(100, 230, 110),
               text="Easy"))  # easy button

    self.playingButtons.append(
        Button(264, 40, WIDTH // 8, 40, function=self.mediumDiff, colour=(255, 180, 50),
               text="Medium"))  # medium button

    self.playingButtons.append(
        Button(356, 40, WIDTH // 8, 40, function=self.hardDiff, colour=(240, 130, 130),
               text="Hard"))  # hard button

    self.playingButtons.append(
        Button(450, 40, WIDTH // 8, 40, function=self.solver, colour=(110, 200, 250),
               text="Solve"))  # solve button
```

In loadButtons, I am setting up the buttons for the GUI. I'm using the structure specified by the button class which is discussed below. Each button has its own use, specified by the 'function' parameter. I have a check button, three buttons with variable difficulty and a solve button. They each have their own colour and text to specify what each button does. The game class and button class are associated by aggregation. It also satisfies objective 1.2. and 2.2. which is to "display the buttons which have a range of uses, including checking the board and changing the difficulty of the board" and "should include buttons which have unique features, which are checking the puzzle, choosing a difficulty setting for the puzzle and solving".

Explanation of textOnScreen()

```
def textOnScreen(self, window, text, position):
    font = self.font.render(text, False, BLACK)
    fontWidth = font.get_width()  # gets the width of the font object
    fontHeight = font.get_height()  # gets the height of the font object
    position[0] += (cellSize - fontWidth) // 2  # finds the centre of the cell width
    position[1] += (cellSize - fontHeight) // 2  # finds the centre of the cell height
    window.blit(font, position)  # outputs onto screen
```

This function is used to output the numbers onto the board for when a user enters a number. Its width and height is calculated to produce the font at the right size and works out the position so that the font is added to the centre of the button. It's used in the drawNumbers function.

Explanation of load()

```
def load(self):
    # contains code to setup a new game
    # sets lists empty to initialise board
    self.playingButtons = []
    self.loadButtons()
    self.lockedCells = []
    self.incorrectCells = []
    self.finished = False

    # setting up locked cells from original board
    for yindex, row in enumerate(self.grid):
        for xindex, num in enumerate(row):
            if num != 0:
                self.lockedCells.append([xindex, yindex])
```

The function initiates the program when it is run. It resets all lists to being empty, such as the incorrectCells list, as these cannot have any values to begin with. The game's buttons are loaded and displayed on the GUI. The game has just started so self.finished is set to false. I will also need to declare which cells are 'locked cells'; these are cells already present in the grid before the game starts and are appended to list lockedCells. For the puzzles in my program, any value that is not equal to zero initially, isn't a locked cell. This is done to stop users from changing the values of pre-set cells as well when checking the program.

Explanation of checkInt()

```
def checkInt(self, string):
    # contains code to stop users passing in non-numeric values into cells
    try:
        int(string)  # tries to convert value in variable string into an integer
        return True
    except:
        return False  # if a non-numeric string is entered, it returns False and returns an error
```

This section of code refers to the exception handling to prevent errors caused by the user entering a non-numeric value. You can see it tries to convert the value in the variable string into an integer, and if it is one, it'll return true, else it will return false and be ignored. You can't have characters in a sudoku puzzle, so making sure that only integer inputs are passed into the grid.

Explanation of changeGrid

```
def changeGrid(self, puzzle):
    # contains code in charge of changing the grid
    self.grid = puzzle
    self.load()
```

This function changes the grid when a difficulty button is clicked so that the new board is displayed. The self.grid property is changed to the parameter puzzle.

Explanation of split

This function is used by the database functions to restore the grid from a string into a 2D array

```
def split(self, var):
    # used by database functions to restore grid into 2D array format
    # sets lists empty each time to stop duplication of data
    self.splitString = []
    self.newGrid = []

    # appending numbers as 1-D array
    for char in range(len(var)):
        self.splitString.append(int(var[char]))
    # appending numbers as 2-D array
    for index in range(0, 81, 9):
        self.newGrid.append(self.splitString[index:index + 9])
```

format, so that it can produce the grid. Each time the split function is called, it sets both splitString and newGrid empty, due to there being an error which would cause the lists to not be refreshed and produce the same grid in the same list, causing an error when displaying the board. I then append the numbers into a 1-D array format in splitString and then into a 2-D array in newGrid. This function is used by the database functions.

SOLVER FUNCTIONS

The process to backtrack a puzzle can be split into two parts: first checking that the puzzle is solvable and second applying the backtracking algorithm, using the logic of the flowchart from the design stage. Applying the '*Python Sudoku Solver*' tutorial by Computerphile[20], I wrote the following code. Note that the 'grid' variable is where the default puzzle is stored.

Explanation of updateSolved

```
def updateSolved(self, window):
    # updating solved board
    self.drawNumbers(self.window)
    pygame.display.update()
```

Once the board has been solved, I use updateSolved to draw the numbers again and refresh the GUI so that they're shown on the screen.

Explanation of possible()

```
def possible(self, y, x, n):
    # checking to see if it's possible to solve the board
    for i in range(0, 9):
        if self.grid[y][i] == n:
            return False
    for i in range(0, 9):
        if self.grid[i][x] == n:
            return False
    x0 = (x // 3) * 3
    y0 = (y // 3) * 3
    for i in range(0, 3):
        for j in range(0, 3):
            if self.grid[y0 + i][x0 + j] == n:
                return False
    return True
```

Checking column

Checking row

Checking sub-grid

The first part as mentioned will involve us checking to see it's possible to put a certain number in a cell. Parameters are y and x, which refer to the coordinates within the grid, and n is the number from 1-9 that could be added. It returns true if the number is a possible option, otherwise if the number already exists, it will return false.

[20] "*Python Sudoku Solver – Computerphile*", YouTube, 12/02/20
https://www.youtube.com/watch?v=G_UYXzGuqvM

From the order of the code, you can observe that the function checks the rows first, then the columns and then then each sub-grid. Checking each row and column follows the same format, but checking each sub-grid requires us to find the quotient of the x and y coordinates and multiplying it by 3 to find the grid coordinates of the sub-grid it belongs to, and checking if the number, n, appears in that subsequent sub-grid.

Explanation of solver()

```
def solver(self):
    for yindex, row in enumerate(self.grid):  # for each row in the grid
        for xindex, num in enumerate(row):  # for each cell in each row
            if self.grid[yindex][xindex] == 0:
                for n in range(1, 10):  # counts to n-1, so 10 is never reached
                    if self.possible(yindex, xindex, n) and self.finished == False:
                        self.grid[yindex][xindex] = n
                        self.solver()  # recursion
                        self.grid[yindex][xindex] = 0  # backtrack if not possible so make space empty again
                return

    # updating the solved board
    self.updateSolved(self.window)

    # freezing the screen and getting an input
    input("Enter to continue: ")
```

When solving you would want to look at every cell in the grid, hence why I use for y and for x in a range of 9. If a grid coordinate is marked 0, it indicates that the cell is empty, in which case a for loop is called in the range of 1-10 (as in Python, it counts to n-1, thus 10 is never actually reached). The possible function is then applied to see what numbers are possible. If it's possible the cell is changed to the number. I then use the principles of recursion by calling the solver function again inside itself, so that I can keep checking each cell. However, if a cell is no longer possible due to a mistake earlier, the space is made empty again by setting the cell equal to 0. The process is repeated until all the cells are filled and the puzzle is solved – think back to the tree traversal graph in the prototype. The solved board is then updated and using the input statement, the screen is frozen until we get the user's input to continue gameplay.

DATABASE FUNCTIONS

These set of functions are anything to do with manipulating the database. The functions can be found in the file database and in the gameClass. The database is stored in ‘database.sqlite’. I’m going to be creating the database, creating the tables, storing each puzzle in the right tables and accessing them when using buttons.

```
import csv
import sqlite3
from settings import *

connection = sqlite3.connect('database.sqlite') #establishing a connection to the database
cursor = connection.cursor()

with open('SudokuData.csv', 'r') as csvfile: #reading the csv file
    reader = csv.reader(csvfile)
    puzzles = []
    for line in reader:
        puzzles.append(line)

puzzles = puzzles[2:] #data for puzzles table
easy = puzzles[0:10] #data for easy table
medium = puzzles[10:20] #data for medium table
hard = puzzles[20: 30] #data for hard table
```

To begin with I have the code above which is used to establish a connection the database file and open the csv file which has the string format for each puzzle. You can see how this csv table would look on the next page. The file is read and then each line is appended into the puzzles list. Next by slicing the list, you can get the string needed to append for the easy table, medium table, hard table and puzzles table, which includes all the puzzles of that difficulty.

SudokuData

SudokuData		
SodukuID	SodukuLevel	SodukuData
1	e	540060078306180020092004500609000705710496000408003100000030900180000257950027000
2	e	015000000938200570000030016802079360000002700170058420000720050386091007057000000
3	e	700100300090020710210040009029400100008050002050031900002300076507090800400705003
4	e	020500038600240000159038000000000000840190052290704083000061379000902004960300020
5	e	036087000002030076900062030010020050600005003590000041340009800000058904100046020
6	e	094250060000090000028016090071502039040080102009034600000000300085300720032800904
7	e	309600500570100040084003700107200300032009600690400050025904800000070005860300470
8	e	000000000901250840047380290005830020402000600380000159610973000003020960029400000
9	e	000062000091030040650900130000000089045307200310006004000000005002706980026834 05
10	e	193000002068070013000100080014820060000040008857600000029300007080702609001906804
11	m	500300000020080000000907024006000005002000001000008626047059804090010330005006 7
12	m	003402609000700800000030007040003000970806030500090000850040000060307200000020904
13	m	000700300708009061050060000010500000000603004089007050400078000000209000060052 40
14	m	690700300000620810000100076960000047030570008070000620300200000400300007000400 30
15	m	006000000090000063503004100060200041000000902050006004071009868310060540004870 00
16	m	607100908030004600009300005100540360900060050050009200000600120020007000700000503
17	m	018070002000000000740020008400200000063104020000800601820400950005000010000059080
18	m	000000000030049060908570002002064105000000000050320040200783600700051000000690800
19	m	001927005000080020007006841008053000350600000000804053009000072410000600000062504
20	m	002067000006005300701004000409000031280000590000000000000210870000000000000870104
21	h	380410000000000000021803000000000000710000092906000304000580040007060050000020018
22	h	000060003068001007000035600890000050500290001030000006017000304200006000000310068

Explanation of createPuzzle()

```
def createPuzzles(): #creating the puzzles table
    try:
        cursor.execute('''
            CREATE TABLE Puzzles (
                ID int NOT NULL,
                Level STRING,
                Data STRING
            )
        ''')
        connection.commit()
    except sqlite3.OperationalError:
        print("Puzzles has already been created")
        return
```

I'm going to create the puzzles table using this function which uses the SQL statement 'create'. Its attributes include ID which is the unique identifier that's an integer, cannot be empty and is the primary key for the table; level which is a string and specifies the difficulty of the puzzles; and data which is also a string and contains the actual puzzle. I use try and except so that if the function is accidentally called again, you get a message saying that the table has already been created. Note that these attributes are empty initially. This satisfies the objective 4.1 Successfully store puzzles in a database.

Explanation of createEasy()

```
def createEasy(): #creating the easy table
    try:
        cursor.execute('''
            CREATE TABLE Easy (
                ID int NOT NULL,
                Data STRING
            )
        ''')
        connection.commit()
    except sqlite3.OperationalError:
        print("Easy has already been created")
        return
```

Similar to puzzles, I'm going to create the easy table using this function which uses the SQL statement 'create'. Its attributes include ID is which is the unique identifier that's an integer, cannot be empty and is the primary key for the table, and data which is also a string and

contains the actual puzzle that's easy difficulty. I use try and except so that if the function is accidentally called again, you get a message saying that the table has already been created.

This function will have the same format for createMedium and createHard, but instead, creating the tables for those corresponding difficulties. Note that these attributes are empty initially.

Explanation of addPuzzles()

```
def addPuzzles(): #adding data into the puzzles table
    for puzzle in puzzles:
        #setting values to insert
        id = int(puzzle[0])
        level = str(puzzle[1])
        data = str(puzzle[2])
        #executing command in sql
        cursor.execute(f'''
            INSERT INTO Puzzles (ID, Level, Data)
            VALUES ('{id}', '{level}', '${data}');
        ''')
        connection.commit()
    cursor.close()
    connection.close()
```

This procedure is in charge of adding the data for puzzles from the csv file into the newly formed puzzles table. I'm going to use the SQL statement 'insert' and be specifying that the data I'm going to be inserting will be into attributes ID, level and data. The puzzles table will have all 30 puzzles of varying difficulties and will be used so that each time the program is run, a random puzzle is produced on the board. At the end the database is closed.

Explanation of addEasy()

```
def addEasy(): #adding data into the easy table
    for puzzle in easy:
        # setting values to insert
        id = int(puzzle[0])
        data = str(puzzle[2])
        # executing command in sql
        cursor.execute(f'''
            INSERT INTO Easy (ID, Data)
            VALUES ('{id}', '${data}');
        ''')
        connection.commit()
    cursor.close()
    connection.close()
```

This procedure is in charge of adding the data for puzzles from the csv file into the newly formed easy table. I'm going to use the SQL statement 'insert' and by specifying that the data I'm going to be inserting will be into attributes ID and data. The easy table will have the first 10 puzzles of the puzzles list, which include puzzles of easy difficulty. This procedure will have the same format for addMedium and addHard, but instead, puzzles 11-20 are medium and 21-30 are hard. At the end the database is closed.

Demonstration of Puzzles Table

Puzzles (30 rows)

```
SELECT * FROM 'Puzzles' LIMIT 0,30
```

Execute

ID	Level	Data
1	e	$5400600783061800200920045006090007057104960004080031000000309001800002579500270000
2	e	$0150000009382005700000300168020793600000027001700584200007200503860910070570000000
3	e	$7001003000900207102100400090294001000080500020500319000023000765070908004007050003
4	e	$0205000386002400001590380000000000008401900522907040830000613790009020049603000200
5	e	$0360870000020300769000620300100200506000050035900000413400098000000589041000460200
6	e	$0942500600000900000280160900715020390400801020090346000000030008530072003280090 4
7	e	$3096005005701000400840037001072003000320096006904000500259048000000700058603004 70
8	e	$0000000009012508400473802900058300204020006003800001596109730000302096002940000 0
9	e	$0000620000910300406509001300000000890453072003100060040000000050027069800268340 5
10	e	$1930000020680700130001000800148200600000400088576000000293000070807026090019068 04
11	m	$5003000000200800000000907024006000000500200000100000862604705980409001033000500 67
12	m	$0034026090007008000000300070400030009708060305000900008500400000603072000000209 04

This is the puzzles table which I created shown on the website '*SQLite Viewer'* [21]. As you can see I've stored all the puzzles from the csv file. There're attributes ID, level and data. The first 10 puzzles are easy, the next 10 are medium and the 10 after that are hard.

The data is the string format of the puzzles. You may notice that there's a dollar sign and this is because each time we added the data, it would compress the string. This means that there may be an error when reading the data.

Thus to ensure that this wouldn't occur, I've added the dollar sign. When reading the data, I simply splice the dollar sign off, so it leaves just the string.

Demonstration of Easy Table

Easy (10 rows)

```
SELECT * FROM 'Easy' LIMIT 0,30
```

Execute

ID	Data
1	$5400600783061800200920045006090007057104960004080031000000309001800002579500270000
2	$0150000009382005700000300168020793600000027001700584200007200503860910070570000000
3	$7001003000900207102100400090294001000080500020500319000023000765070908004007050003
4	$0205000386002400001590380000000000008401900522907040830000613790009020049603000020
5	$0360870000020300769000620300100200506000050035900000413400098000000589041000460020
6	$0942500600000900000280160900715020390400801020090346000000030008530072003280090 4
7	$3096005005701000400840037001072003000320096006904000500259048000000700058603004 70
8	$0000000009012508400473802900058300204020006003800001596109730000030209600294000000
9	$0000620000910300406509001300000000890453072003100060040000000050027069800268340 5
10	$1930000020680700130001000800148200600000400088576000000293000070807026090019068 04

This is the easy table which I created. As you can see I've stored all the puzzles from the csv file which are easy difficulty. There're attributes ID and data.

[21] SQLite Viewer
https://inloop.github.io/sqlite-viewer/

Demonstration of Medium Table

Medium (10 rows)

```
SELECT * FROM 'Medium' LIMIT 0,30
```

Execute

ID	Data
11	$5003000000200800000000907024006000005002000001000008626047059804090010330005006 7
12	$0034026090007008000000300070400030009708060305000900008500400000603072000000209 04
13	$0007003007080090610500600000105000000006030040890070504000780000000209000060052 40
14	$6907003000006208100001000769600000470305700080700006203002000000400300007000400 30
15	$0060000009000006350300410006020004100000009020500060040710098683100605400048700 0
16	$6071009080300046000093000051005403609000600500500092000006001200200070007000005 03
17	$0180700020000000007400200084002000000631040200008006018204009500050000100000590 80
18	$0000000000300490609085700020020641050000000000503200402007836007000510000006908 00
19	$0019270050000800200070068410080530003506000000008040530090000724100006000000625 04
20	$0020670000060053007010040004090000312800005900000000000000210870000000000008701 04

This is the medium table which I created. As you can see I've stored all the puzzles from the csv file which are medium difficulty. There're attributes ID and data.

Demonstration of Hard Table

Hard (10 rows)

```
SELECT * FROM 'Hard' LIMIT 0,30
```

Execute

ID	Data
21	$380410000000000000021803000000000007100000929060003040005800400070600500000200 18
22	$0000600030680010070000356008900000505002900010300000060170003042000060000003100 68
23	$0000006000030000154028000000405300009500000400000060000100905030080570040000200 07
24	$0090000430070082006800200009007000010000903200000435008050009000902060000100700 80
25	$0061000809800030000400091000020080700107000080009503400040600031080000040508000 90
26	$8000000030300002107000005900200700006005080000401030000049050800006000300058704 60
27	$0050001409010008008002009040030050038001000007064000000000700800056027000000400 9
28	$0000024598000100000549070000000008000000003004000090620030060001902800006000002 81
29	$6000000900920007040580001060102900070000600000681030007048000200009000005630400
30	$0000000002800000594090001030000000000012070800007841000650003000276000070140000 0

This is the hard table which I created. As you can see I've stored all the puzzles from the csv file which are hard difficulty. There're attributes ID and data.

Explanation of easyDiff()

```
def easyDiff(self):
    try:
        # fetches a puzzle from easy table
        cursor.execute(f'''
        SELECT Data FROM 'Easy'
        ORDER BY RANDOM()
        LIMIT 1
        ''')
    except sqlite3.OperationalError:
        print("Error occurred selecting data")
        return
    data = cursor.fetchall() # assigns this data to a variable
    string = str(data)[4:85] # gets the appropriate string
    self.split(string) # reverts it back into a 2D array
    self.changeGrid(self.newGrid)
```

This function is stored in the game class. The puzzle is fetched from the easy table using the SQL statement, ‘select’. I use order by random to randomise the selection and limit by 1 to select a specific puzzle. I then store the value fetched from the table into a variable ‘data’ and then get the appropriate string value which I convert back into a 2D array using the split function. I then apply changeGrid to update the grid to the newly fetched puzzle. Note that mediumDiff(), hardDiff() and puzzles() all follow this same format.

These functions are used by the buttons, discussed in loadButtons and satisfies the objective 2.1, “should include buttons which have unique features, which are checking the puzzle, choosing a difficulty setting for the puzzle and solving” and 4.2, “Access the database when a difficulty is selected and produce a board from that subsequent difficulty”.

BUTTON CLASS

The button class is a separate class to the main game class, however the two are associated by aggregation. Therefore, if the game class is destroyed, the button class can still function, however the functionality of the buttons in the program will not be shown unless instantiated somewhere. It has a relationship where objects of the classes Game and Button could be associated as “a game has buttons”.

Explanation of __init__()

```
class Button:

    def __init__(self, x, y, width, height, text=None, colour=(180, 180, 180), highlightedColour=(102, 102, 102),
                 function=None):

        self.image = pygame.Surface((width, height))
        self.position = (x, y) # coordinates used for drawing buttons
        self.rect = self.image.get_rect() # shape of buttons
        self.rect.topleft = self.position # top left of buttons
        self.text = text
        self.colour = colour
        self.highlightedColour = highlightedColour
        self.function = function # function to be called
        self.highlighted = False # indicates if mouse is hovering over button
        self.width = width
        self.height = height
```

The __init__() function is a method in python which is used as the constructor for the button class. It initialises the attributes of the class. The parameters x and y are coordinates for where to put the buttons; width and height is the width and height of the buttons itself; text is used to create text, found in the drawText method shown below; colour stores the default colour for each rectangle before it is changed, but this colour is used as a test, as each button has their own separate colour, seen in loadButtons; highlightedColour is simply the colour of the button when the mouse is hovering over it; finally, function is used for when the button takes in a function to be used and calls it. This is because each button has their own purpose, for example checking the board, which satisfies objective 2.2. To create a surface for where the image is drawn, I pass in a tuple x and y. Method ‘rect’ is used for creating a rectangle. Top left of rectangle is to clarify the position of the rectangle, which will play a role in producing the GUI properly, such as the grid. The rest of the attributes are fairly self-explanatory, and you will see how they’re used in the functions of the class.

Explanation of update()

```
def update(self, mouse):
    # contains code used to indicate if mouse is hovering over button
    if self.rect.collidepoint(mouse):
        self.highlighted = True
    else:
        self.highlighted = False
```

The update function takes the position of the mouse; it is responsible for highlighting the button when the mouse is hovering above it. The Pygame function, ‘collidepoint’, allows us to check if the rectangle is colliding with a point, in which case this is the button, and if it is, it returns true; otherwise if it isn’t it returns false. This is going to be used by ‘self.highlighted’, to indicate when to shade the button at the right time.

Explanation of draw()

```
def draw(self, window):
    # draws the buttons
    if self.highlighted:
        # highlights button if mouse if hovering over it
        self.image.fill(self.highlightedColour)
    else:
        # highlights button with its corresponding colour
        self.image.fill(self.colour)
    # blits button onto screen in the right position
    window.blit(self.image, self.position)

    #if self.text:
    # draws text on buttons as well
    self.drawText(self.text)
    window.blit(self.image, self.position)
```

The draw function is responsible for producing the colour of the image, which targets the objective of creating an aesthetically pleasing GUI. It fills the image with the colour defined, If the mouse if hovering above it, it fills it with the highlighted colour, else, if not, then it'll use the default colour setting. The images are 'blitted' to the window parameter at the correct position. This is used by the playing state functions and can be seen in playingDraw, as shown, to actually display it on the GUI.

```
def playing_draw(self):
    self.window.fill(WHITE)

    for button in self.playingButtons:
        button.draw(self.window)
```

Explanation of click()

```
def click(self):
    self.function()  # calls the function when you click the button
```

When the player clicks on a button, the click function is called. As mentioned above, each button has their own purpose, thus when adding buttons to 'loadButtons' I want to describe the functions to be called, hence why it is an attribute of the class. So, when the click function is called, it will simply call the function that it specifies. You may remember that click is used by the playing state functions in playingEvents, after identifying that the user has clicked the mouse. I've included the snippet of code where it is used.

```
# if the user clicks the mouse
if event.type == pygame.MOUSEBUTTONDOWN:
    selected = self.mouseOnGrid()
    if selected:
        self.selected = (selected)  # returns position of mouse
    else:
        self.selected = None
        for button in self.playingButtons:
            if button.highlighted:
                button.click()
```

Explanation of drawText()

```
def drawText(self, text):
    # contains code for drawing text on buttons
    font = pygame.font.SysFont("system", 24, bold=1)
    text = font.render(text, False, (50, 50, 50))
    width, height = text.get_size()
    # returns tuple, first width and then height
    x = (self.width - width) // 2
    y = (self.height - height) // 2
    # working out the coordinates for positioning the text properly
    self.image.blit(text, (x, y))
    # blits text onto screen
```

Similar to the textOnScreen function in the game class, drawText performs the same action. It draws text on top of the button to identify what that button is doing. When comparing side by side, you may notice some differences though. First, to make the code more compact, I've used a tuple for width and height and the pygame function get_size to return both the width and the height, rather than using get_width and get_height. Consequently, this makes finding the centre of the cell a bit easier as well. I then blit the given text on top of the centre of the rectangle.

SETTINGS FILE

This file is part of my project which stores miscellaneous items of the program.

Explanation of Colours Used

```
# colours used in program
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
GREEN = (0, 200, 0)
TEAL = (174, 227, 245)
LOCKEDCELLCOLOUR = (245, 245, 245)
INCORRECTCELLCOLOUR = (255, 150, 150)
```

These are all the colours being used in the programme. They are constants which represent the different RGB codes for each colour.

Explanation of Board Structure

```
# default board
grid = [[0,0,0,0,0,0,0,0,0],
        [9,0,1,2,5,0,8,4,0],
        [0,4,7,3,8,0,2,9,0],
        [0,0,5,8,3,0,0,2,0],
        [4,0,2,0,0,0,6,0,0],
        [3,8,0,0,0,0,1,5,9],
        [6,1,0,9,7,3,0,0,0],
        [0,0,3,0,2,0,9,6,0],
        [0,2,9,4,0,0,0,0,0]]
```

This is the default board in the game. It's one of the puzzles from the puzzles table. It shows how the 2-D array for the game would look like.

Explanation of Positions and Sizes Used

```
# positions and sizes for creating the board
WIDTH = 600
HEIGHT = 600
gridPosition = (75, 100)  # the top left of the grid
cellSize = 50
gridSize = cellSize * 9
```

These are the positions and sizes used for drawing the GUI, for example in drawing the board, positioning the buttons appropriately and creating the correct size for each cell.

Explanation of join()

```
# join function joins numbers in a 2-D array format into a string
def join(string):
    string = ""
    for listNumber in range(9):
        for index in range(9):
            string = string + str(grid[listNumber][index])
    print(string)
    # used when adding numbers into csv file
```

This was the function I used to join the numbers in the original 2-D array into a string format, so that it could be added into a csv file, which stored it in the database.

Testing

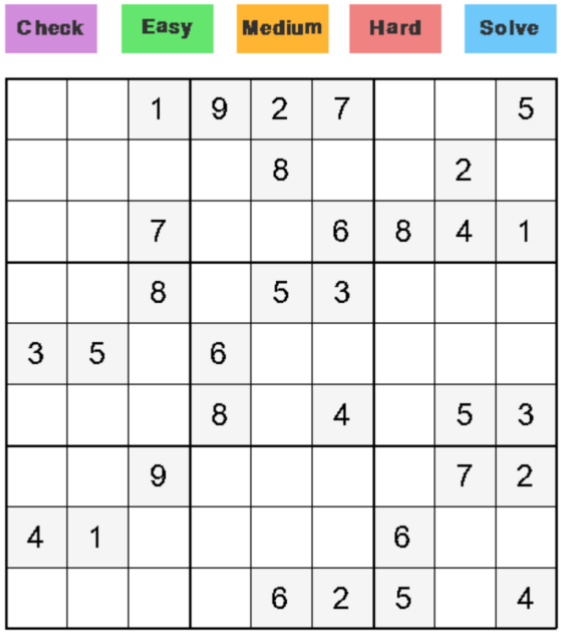
In this section, I am going to test my program with my original objectives. Testing will be split into the format specified in the test plan in design; this means the first test will be on the GUI, then the gameplay, then the solver and finally the database. The purpose of testing is to try and uncover any undetected errors in the code. To ensure that all of the core requirements have been met, I've designed the tests, such that each objective is tested. The link to the YouTube video where the video testing have been conducted is: https://youtu.be/N6s2gmz8iJU.

TESTING GUI

To begin with, I'm going to be testing my GUI to see if it produces the board and its buttons correctly. As I've mentioned already all my tests have been designed to ensure that all objectives are met. Objectives for the GUI that need to be satisfied include:

1.1. Display the main playing board using Pygame
1.2. Display buttons which have a range of uses, including checking the board and changing the difficulty of the board
1.3. Create a simplistic and visually appealing GUI
1.4. Make use of colour in the GUI

A screenshot of the GUI created is shown, which satisfies the above requirements.



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

The table below shows each test conducted on the GUI, the expected results and the actual results observed. I've included the timestamps for each test in the video.

Test Number	Test	Expected Result	Actual Result	Timestamp
1.1	Testing if the board is produced correctly	When program is run, GUI is loaded onto screen with the board	Same as expected	00:22
1.2	Testing if the 'locked cells' are shaded	When board is loaded, locked cells are different shade to normal cells	Same as expected	00:45
1.3	Testing if buttons appear	When program is run, GUI is loaded with buttons produced above the board	Same as expected	00:22
1.4	Testing if incorrect cells are shaded	When an incorrect answer is inputted, and the board is checked, it shades the cell red	Same as expected	01:14
1.5	Testing if selected cells are shaded	When a cell is clicked on, cell is shaded	Same as expected	01:02
1.6	Testing if numbers are on the board	When board is loaded, numbers from the 2-D array are outputted on the cells	Same as expected	00:22
1.7	Testing if buttons are highlighted when mouse is above it	When the mouse is hovered above the buttons, buttons should change to a different colour to indicate this	Same as expected	00:32

TESTING GAMEPLAY

To test the gameplay aspects, I used the PyCharm debugger tool extensively in order to resolve issues while testing. For instance, I faced an error during solving where after the board is solved, it would instantly disappear again. After using the debugger, I realised that due to the recursive nature of the solver, it would 'unwind' and unsolve itself afterwards. Thus, to prevent this, using an input statement and the python console, I managed to solve this issue.

In addition, each time I would change the board, it would fetch the numbers for the same puzzle and add onto the same list where the GUI displays the board, thus causing the board to be displayed incorrectly. To combat this, I had to make sure the list was empty each time, by initialising the variables responsible for fetching the grid as empty and creating a new function changeGrid, which made changing the board easier.

Moreover, I also had an issue where I had to specify self.selected, a selected cell, as being a list and use it to compare it with self.lockedCells, list of locked cells, in order to stop a problem where a few locked cells could be changed by the user when they shouldn't be able to.

Test Number	Test	Expected Result	Actual Result	Timestamp
2.1	Testing if users can input numbers	When user clicks on a cell and enters a number, 1-9, it's inputted into the cell. If it's a 0, it clears the cell.	Same as expected	02:11
2.2	Testing if users cannot input non-numeric characters	When user clicks on a cell, they should not be able to enter non-numeric characters, e.g., 'a' or '!'	Same as expected	02:18
2.3	Testing that user cannot change a locked cell	When user clicks on a locked cell and attempts to change the number, no changes should be made	Same as expected	02:45
2.4	Testing that buttons work	When user clicks on a button, they perform their respective functions	Same as expected	03:08
2.5	Testing what happens when a user completes the game	They should be greeted with a congratulations message	Same as expected	04:48

TESTING SOLVER

As mentioned above, when testing the solver, I encountered a problem where after the board becomes solved, it would disappear immediately. Once I managed to solve this issue, there were no further problems. I used a variety of different puzzles of varying difficulties, and in my tests it would get to the correct solution each time almost immediately. Therefore, the tests prove that I have successfully met the following objectives.

3.1 Use recursion to solve the puzzle
3.2 It should be able to solve any puzzle
3.3 The time to solve each puzzle should be consistent

The table used for testing and the timestamp for each test in the respective video is below.

Test Number	Test	Expected Result	Actual Result	Timestamp
3.1	Testing if backtracking algorithm works	When user clicks solve, board should be solved, where no row, column or sub grid contains more than one occurrence of a number	Same as expected	05:52
3.2	Testing if backtracking algorithm works on puzzles of all difficulties	Solver should work for easy, medium and hard difficulty puzzles	Same as expected	06:37
3.3	Testing backtracking algorithm run-time is consistent	Time to solve each puzzle of varying difficulties should be consistent	Same as expected	08:04

TESTING DATABASE

When testing the database functions, I needed to look at gameClass.py and database.py. The database file is different from the main game file, because I only needed to use the functions once; this was for importing the csv file which stores each puzzle into a database format, creating the normalised tables in the database file and adding the correct data extracted from the csv file. Note that the sudoku data is pre-validated, meaning I've checked to ensure that they're all solvable, thus making sure no puzzles appear in the program that can't be solved. As for the gameClass file, I will be using database functions for fetching data from the tables which will be used at the beginning of the program and every time the user changes difficulties, hence why it's better suited to be in the same file as the rest of the main game functions.

You may notice in the video, that the data has dollar signs, and as I mention, this was to correct an error where the puzzle data stored in the tables would be compressed and consequently, not read properly when fetching the data to display the new puzzle on the board.

Below are the original objectives to be satisfied for the database. To prove 4.1, I will be showing videos of the created tables in SQLite Viewer[22] and for 4.2, I will video the different boards each time a difficulty button is clicked and seeing if it's in their respective table.

4.1 Successfully store puzzles in a database
4.2 Access the database when a difficulty is selected and produce a board from that subsequent difficulty

The table used for testing and the timestamp for each test in the respective video is below.

Test Number	Test	Expected Result	Actual Result	Timestamp
4.1	Testing boards are stored properly	Boards should be fetched correctly from csv file, database file should be created, tables and their respective data should be imported correctly	Same as expected	08:50
4.2	Testing if users can choose different difficulty boards	When user clicks on a difficulty button, the correct tables should be accessed and should fetch a puzzle randomly	Same as expected	12:39

[22] SQLite Viewer – Juraj Novak, github
https://inloop.github.io/sqlite-viewer/

Evaluation

This is the evaluation of my project, where I discuss how successful my project has been as well as what my client thinks after using the system. I will be going through my initial objectives and discuss to what extent I believe I have completed them. To conclude, I will also be exploring things I could've done differently for my project.

EVALUATION OF OBJECTIVES

Evaluation of GUI

1.1. Display the main playing board using Pygame

1.2. Display buttons which have a range of uses, including checking the board and changing the difficulty of the board

1.3. Create a simplistic and visually appealing GUI

1.4. Make use of colour in the GUI

I've managed to achieve these objectives as shown in testing, where I managed to create the board, buttons and display numbers using Pygame. It displays the board, buttons, and according to my client, has a pleasant looking user interface.

Evaluation of Gameplay

2.1 Should be able to play the game like a normal Sudoku

2.2 Should include buttons which have unique features, which are checking the puzzle, choosing a difficulty setting for the puzzle and solving

2.3 Shouldn't be able to enter non-numeric values into the cell

2.4 Shouldn't be able to change cells that are in the grid before user starts

I believe I have completed these set of objectives too. My client had no issues playing the game like a typical sudoku and managed to complete it successfully with no errors. He utilised the buttons to change difficulties, solve a board, and check a puzzle that he attempted. He also tested to see if he could change any locked cells, which he could not, and if he could add letters and non-integer characters into the board and found these weren't passed in, which fulfils objectives 2.3 and 2.4. nicely.

Evaluation of Backtracking Solver

3.1 Use recursion to solve the puzzle

3.2 It should be able to solve any puzzle

3.3 The time to solve each puzzle should be consistent

I managed to get the solver working for different puzzles and on tests I found it solved puzzles of varying difficulties almost instantaneously, with little to no variation in times between each difficulty, thus satisfying the objectives for the solver. One thing to note though, is that solving the board, requires the user to use the python console to continue playing, which my client did not mind. Moreover, I made it so that the user cannot play on the same board after solving, which my client was also fine with.

Evaluation of Database

4.1 Successfully store puzzles in a database

4.2 Access the database when a difficulty is selected and produce a board from that subsequent difficulty

Storing and accessing the individual puzzles in the database was also a success, with objective 4.2 tying into objective 2.2., where buttons are being used to call functions that fetch data from tables in the database.

FEEDBACK FROM END USER

"I'm grateful for the program that's been created for me, which has provided me with a new alternative way to play Sudoku. I can now play Sudoku with all its normal features as well as be able to solve puzzles without using cellular data/ hotspot whilst I do not have access to internet. I'm satisfied with all the features that have been implemented and like the fact that the system has been built in such a way that easily allows further changes to be made in the future. I've found it straightforward to use and have had no experience with bugs during my time using it. Speed of operations were fine too, and the solver worked correctly every time I used it. Overall, I'm very thankful for this application." – Kelsung L (my client)

I'm very pleased that I've managed to create a fully-functioning system that has satisfied my client's needs. It brings me joy to hear that he's had a fun time using it. This isn't to say that there were some areas that I felt I could've done better, which I will discuss below.

CONCLUDING REMARKS

Overall, I believe my project has been a successful one, given that all of my objectives have been fulfilled. I've tried to keep a consistent coding style throughout my project, for example I have tried to maintain the use of the variable naming convention known as camelCase throughout my code. This is where the initial letter of each variable is lowercase and any words following within each name starts with a capital letter.

I believe my code has made use of meaningful identifiers, indentation and comments where appropriate to improve the readability and clarity aspect, so that if a programmer with no experience with my code were to look at it, they could understand what is going on. This is also with the addition of the documentation of my project with the design and implementation stages describing the flow of data and interaction between each of these functions,

The use of cohesive subroutines helped tremendously when developing the program, as each function was responsible for perform a single task. This meant that I could reduce the repetition of code and also made writing the code easier, as I could understand the logic behind what I was writing. For example when changing the difficulties of the grid, calling the changeGrid function would set self.grid equal to the new grid fetched by the database and load it appropriately on the grid.

However, the journey was not an easy one and there were some notable difficulties along the way. For example, changing the locked cells required me to specify that self.lockedCells was a list, otherwise, lockedCells would not retain the list of locked cells properly. Also, I had a difficult time displaying the board whenever it was changed, for example when a new board was selected or it was solved using a solver. It was only until I managed to create additional helper functions and variables that aided the process of changing the board that I managed to solve this issue, such as the function changeGrid and updateSolved.

What could have gotten better is displaying the solved board, as the recursive nature of the solver meant that after solving it, it would unsolve itself right after, which is what caused a great deal of confusion when trying to display the solved board. To combat this, an input statement could be placed which would produce the board frozen on the screen and the only way to unfreeze it would be to use the python console. This was not ideal, but luckily my client did not care about this. In the future, if I adjusted the code for the solver functions, I'm sure this issue could've been resolved better. Additionally, perhaps adding more boards for each difficulty, for example instead of 10 of each difficulty, ideally at least 50, however it took a long time to write each puzzle in its 2D array format and then convert into string and then add into the database.

Finally, as I have mentioned before, if I were to extend this project into the future, I could test genetic algorithms and their ability to produce sudoku puzzles just like the paper in design had examined. It would be interesting to see how it produces puzzles and I can test it using the solver and checker functions during gameplay. An even greater challenge would be applying all of the same principles as before but on different types of sudoku boards.

Code

MAIN

```
from settings import *
from gameClass import *

if __name__ == "__main__":
  #calling the game class
  game = Game()
  game.run()
```

GAMECLASS

```
import pygame
import sys

from settings import *
from buttonClass import *
from database import *
from main import *


# ------------------------------------------------------------------------------------------------------
# Game Class
# ------------------------------------------------------------------------------------------------------

class Game():

    def __init__(self):
        pygame.init()
        self.window = pygame.display.set_mode((WIDTH, HEIGHT)) # Pygame window
        self.running = True
        self.grid = grid # default grid
        self.splitString = [] # variables for split
        self.newGrid = []  # variables for split
        self.selected = None # indicates if cell is selected
        self.mousePosition = None # returns position of mouse
        self.state = "playing" # default state
        self.finished = False
        self.cellChanged = False
        self.playingButtons = [] # list that holds all buttons for playing state
        self.lockedCells = [] # list that holds all locked cells
        self.incorrectCells = [] # list that holds all incorrectly played cells
        self.font = pygame.font.SysFont("arial", cellSize // 2) # font to be used
        self.load() # initialises the game
        self.puzzles() # creates a fresh new board every time

    def run(self):
        # the game loop
        while self.running == True:
            if self.state == "playing":
                # calls the following functions which allows gameplay to work
                self.playingEvents()
                self.playingUpdate()
                self.playingDraw()
        # when game is no longer running
        pygame.quit()
        sys.exit()
```

```
# ------------------------------------------------------------------------------------------
# Playing State Functions
# ------------------------------------------------------------------------------------------

    def playingEvents(self):
        # contains code which is in charge of general user gameplay
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.running = False

            # if the user clicks the mouse
            if event.type == pygame.MOUSEBUTTONDOWN:
                selected = self.mouseOnGrid()
                if selected:
                    self.selected = (selected)  # returns position of mouse
                else:
                    self.selected = None
                    for button in self.playingButtons:
                        if button.highlighted:
                            # indicates that mouse is hovering over button
                            button.click()

            # if the user types a key
            if event.type == pygame.KEYDOWN:
                if self.selected != None and list(self.selected) not in self.lockedCells:
                    if self.checkInt(event.unicode):
                        # changes the cell
                        self.grid[self.selected[1]][self.selected[0]] = int(event.unicode)
                        # indexes are in the self.selected attribute
                        self.cellChanged = True

    def playingUpdate(self):
        # contains code which updates the board during the user's gameplay
        self.mousePosition = pygame.mouse.get_pos()
        for button in self.playingButtons:
            button.update(self.mousePosition)

        if self.cellChanged:
            # when a player makes a move
            self.incorrectCells = []
            if self.allCellsFilled():
                # checks if board is correct
                self.checkAllCells()
                if len(self.incorrectCells) == 0:
                    print("Congratulations! \nYou've completed this sudoku!")


     def playingDraw(self):
         # contains which sets up the GUI
         self.window.fill(WHITE)

         # adding buttons
         for button in self.playingButtons:
             button.draw(self.window)

         # when the user selects
         if self.selected:
             self.drawSelection(self.window, self.selected)

         # shading cells
         self.shadeLockedCells(self.window, self.lockedCells)
         self.shadeIncorrectCells(self.window, self.incorrectCells)

         # drawing the GUI
         self.drawNumbers(self.window)
         self.drawGrid(self.window)
         pygame.display.update()
         self.cellChanged = False
```

```
# ------------------------------------------------------------------------------------------------
# Board Checking Functions
# ------------------------------------------------------------------------------------------------

    def allCellsFilled(self):
        # used to check to see if all cells in the grid have been filled
        for row in self.grid:
            for number in row:
                if number == 0:
                    return False
        return True

    def checkAllCells(self):
        # used to check the grid
        self.checkRows()
        self.checkColumns()
        self.checkSubGrid()

    def checkRows(self):  # used to check each row
        for yindex, row in enumerate(self.grid):
            possibleNumbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
            # the list of possible numbers that can be inputted
            for xindex in range(9):
                if self.grid[yindex][xindex] in possibleNumbers:
                    possibleNumbers.remove(self.grid[yindex][xindex])
                    # if a number already appears in the grid, it removes it from the list
                else:
                    if [xindex, yindex] not in self.lockedCells and [xindex, yindex] not in self.incorrectCells:
                        self.incorrectCells.append([xindex, yindex])
                        # if the user makes an incorrect move, it's added to the incorrectCells list
                    if [xindex, yindex] in self.lockedCells:
                        for i in range(9):
                            if self.grid[yindex][i] == self.grid[yindex][xindex]:
                                self.incorrectCells.append([i, yindex])
                                # if the user enters a locked cell, it's added to the incorrectCells list

    def checkColumns(self):  # used to check each column
        for xindex in range(9):
            possibleNumbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
            # the list of possible numbers that can be inputted
            for yindex, row in enumerate(self.grid):
                if self.grid[yindex][xindex] in possibleNumbers:
                    possibleNumbers.remove(self.grid[yindex][xindex])
                    # if a number already appears in the grid, it removes it from the list
                else:
                    if [xindex, yindex] not in self.lockedCells and [xindex, yindex] not in self.incorrectCells:
                        self.incorrectCells.append([xindex, yindex])
                        # if the user makes an incorrect move, it's added to the incorrectCells list
                    if [xindex, yindex] in self.lockedCells:
                        for i, row in enumerate(self.grid):
                            if self.grid[i][xindex] == self.grid[yindex][xindex]:
                                self.incorrectCells.append([xindex, i])
                                # if the user enters a locked cell, it's added to the incorrectCells list


    def checkSubGrid(self):  # used to check each subgrid in the board
        for x in range(3):
            for y in range(3):
                possibleNumbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
                # the list of possible numbers that can be inputted
                for i in range(3):
                    for j in range(3):
                        # used to check each cell in each subgrid
                        xindex = x * 3 + i
                        yindex = y * 3 + j
                        if self.grid[yindex][xindex] in possibleNumbers:
                            possibleNumbers.remove(self.grid[yindex][xindex])
                            # if a number already appears in the grid, it removes it from the list
                        else:
                            if [xindex, yindex] not in self.incorrectCells and [xindex, yindex] not in self.lockedCells:
                                self.incorrectCells.append([xindex, yindex])
                                # if the user makes an incorrect move, it's added to the incorrectCells list
                            if [xindex, yindex] in self.lockedCells:
                                for k in range(3):
                                    for l in range(3):
                                        xindex2 = (x * 3) + k
                                        yindex2 = (y * 3) + l
                                        if self.grid[yindex2][xindex2] == self.grid[yindex][xindex]:
                                            # extra measures caused by bug which wouldn't register some locked cells
                                            self.incorrectCells.append([xindex2, yindex2])
                                            # if the user enters a locked cell, it's added to the incorrectCells list
```

```
# ------------------------------------------------------------------------------------------------
# Helper Functions
# ------------------------------------------------------------------------------------------------

    def shadeIncorrectCells(self, window, incorrect):
        # draws a coloured square to indicate the cell is incorrect
        for cell in incorrect:
            pygame.draw.rect(window, INCORRECTCELLCOLOUR, (
            cell[0] * cellSize + gridPosition[0], cell[1] * cellSize + gridPosition[1], cellSize, cellSize))

    def shadeLockedCells(self, window, locked):
        # draws a coloured square to indicate the cell is locked
        for cell in locked:
            pygame.draw.rect(window, LOCKEDCELLCOLOUR, (
            cell[0] * cellSize + gridPosition[0], cell[1] * cellSize + gridPosition[1], cellSize, cellSize))

    def drawNumbers(self, window):
        for yindex, row in enumerate(self.grid):  # for each row in the grid
            for xindex, num in enumerate(row):  # for each cell in each row
                if num != 0:  # while cell is not blank
                    position = [(xindex * cellSize) + gridPosition[0], (yindex * cellSize) + gridPosition[1]]
                    # draws the numbers in relation to position of grid on the window
                    self.textOnScreen(window, str(num), position)
                    # calls textOnScreen to output numbers

    def drawSelection(self, window, position):
        # draws a coloured square to indicate the cell has been selected
        pygame.draw.rect(window, TEAL, (
        (position[0] * cellSize) + gridPosition[0], (position[1] * cellSize) + gridPosition[1], cellSize, cellSize))

    def drawGrid(self, window):
        # draws border square
        pygame.draw.rect(window, BLACK, (gridPosition[0], gridPosition[1], WIDTH - 150, HEIGHT - 150), 2)
        for x in range(9):
            if x % 3 != 0:  # for every line it draws lines, thickness 1
                # vertical lines
                pygame.draw.line(window, BLACK, (gridPosition[0] + (x * cellSize), gridPosition[1]),
                                 (gridPosition[0] + (x * cellSize), gridPosition[1] + 450))
                # horizontal lines
                pygame.draw.line(window, BLACK, (gridPosition[0], gridPosition[1] + (x * cellSize)),
                                 (gridPosition[0] + 450, gridPosition[1] + +(x * cellSize)))
            else: # for every third line it draws lines, thickness 2
                # vertical lines
                pygame.draw.line(window, BLACK, (gridPosition[0] + (x * cellSize), gridPosition[1]),
                                 (gridPosition[0] + (x * cellSize), gridPosition[1] + 450), 2)
                # horizontal lines
                pygame.draw.line(window, BLACK, (gridPosition[0], gridPosition[1] + (x * cellSize)),
                                 (gridPosition[0] + 450, gridPosition[1] + +(x * cellSize)), 2)


    def mouseOnGrid(self):
        # if we click off the grid, it returns false
        if self.mousePosition[0] < gridPosition[0] or self.mousePosition[1] < gridPosition[1]:
            return False
        if self.mousePosition[0] > gridPosition[0] + gridSize or self.mousePosition[1] > gridPosition[1] + gridSize:
            return False

        # if we click on the grid, it returns the position of the mouse
        return (
        (self.mousePosition[0] - gridPosition[0]) // cellSize, (self.mousePosition[1] - gridPosition[1]) // cellSize)

    def loadButtons(self):
        # initialising buttons for the GUI
        self.playingButtons.append(Button(75, 40, WIDTH // 8, 40, function=self.checkAllCells, colour=(210, 140, 220),
                                          text="Check"))  # check button

        self.playingButtons.append(
            Button(170, 40, WIDTH // 8, 40, function=self.easyDiff, colour=(100, 230, 110),
                   text="Easy"))  # easy button

        self.playingButtons.append(
            Button(264, 40, WIDTH // 8, 40, function=self.mediumDiff, colour=(255, 180, 50),
                   text="Medium"))  # medium button

        self.playingButtons.append(
            Button(356, 40, WIDTH // 8, 40, function=self.hardDiff, colour=(240, 130, 130),
                   text="Hard"))  # hard button

        self.playingButtons.append(
            Button(450, 40, WIDTH // 8, 40, function=self.solver, colour=(110, 200, 250),
                   text="Solve"))  # solve button
```

```
    def textOnScreen(self, window, text, position):
        label = self.font.render(text, False, BLACK)
        fontWidth = label.get_width()  # gets the width of the label object
        fontHeight = label.get_height()  # gets the height of the label object
        position[0] += (cellSize - fontWidth) // 2  # finds the centre of the cell width
        position[1] += (cellSize - fontHeight) // 2  # finds the centre of the cell height
        window.blit(label, position)  # outputs onto screen

    def load(self):
        # contains code to setup a new game
        # sets lists empty to initialise board
        self.playingButtons = []
        self.loadButtons()
        self.lockedCells = []
        self.incorrectCells = []
        self.finished = False

        # setting up locked cells from original board
        for yindex, row in enumerate(self.grid):
            for xindex, num in enumerate(row):
                if num != 0:
                    self.lockedCells.append([xindex, yindex])

    def checkInt(self, string):
        # contains code to stop users passing in non-numeric values into cells
        try:
            int(string)  # tries to convert value in variable string into an integer
            return True
        except:
            return False  # if a non-numeric string is entered, it returns False and returns an error

    def changeGrid(self, puzzle):
        # contains code in charge of changing the grid
        self.grid = puzzle
        self.load()

    def split(self, var):
        # used by database functions to restore grid into 2D array format
        # sets lists empty each time to stop duplication of data
        self.splitString = []
        self.newGrid = []

        # appending numbers as 1-D array
        for char in range(len(var)):
            self.splitString.append(int(var[char]))
        # appending numbers as 2-D array
        for index in range(0, 81, 9):
            self.newGrid.append(self.splitString[index:index + 9])
```

```
# ------------------------------------------------------------------------------------------------------
# Solver Functions
# ------------------------------------------------------------------------------------------------------

    def updateSolved(self, window):
        # updating solved board
        self.drawNumbers(self.window)
        pygame.display.update()

    def possible(self, y, x, n):
        # checking to see if it's possible to solve the board
        for i in range(0, 9):
            if self.grid[y][i] == n:
                return False
        for i in range(0, 9):
            if self.grid[i][x] == n:
                return False
        x0 = (x // 3) * 3
        y0 = (y // 3) * 3
        for i in range(0, 3):
            for j in range(0, 3):
                if self.grid[y0 + i][x0 + j] == n:
                    return False
        return True

    def solver(self):
        for yindex, row in enumerate(self.grid):  # for each row in the grid
            for xindex, num in enumerate(row):  # for each cell in each row
                if self.grid[yindex][xindex] == 0:
                    for n in range(1, 10):  # counts to n-1, so 10 is never reached
                        if self.possible(yindex, xindex, n) and self.finished == False:
                            self.grid[yindex][xindex] = n
                            self.solver()  # recursion
                            self.grid[yindex][xindex] = 0  # backtrack if not possible so make space empty again
                    return

        # updating the solved board
        self.updateSolved(self.window)

        # freezing the screen and getting an input
        input("Enter to continue: ")
```

```
# ------------------------------------------------------------------------------------------
# Database Functions - Accessing Tables
# ------------------------------------------------------------------------------------------

    def easyDiff(self):
        try:
            # fetches a puzzle from easy table
            cursor.execute(f'''
            SELECT Data FROM 'Easy'
            ORDER BY RANDOM()
            LIMIT 1
            ''')
        except sqlite3.OperationalError:
            print("Error occurred selecting data")
            return
        data = cursor.fetchall() # assigns this data to a variable
        string = str(data)[4:85] # gets the appropriate string
        self.split(string) # reverts it back into a 2D array
        self.changeGrid(self.newGrid)

    def mediumDiff(self):
        try:
            # fetches a puzzle from medium table
            cursor.execute(f'''
            SELECT Data FROM 'Medium'
            ORDER BY RANDOM()
            LIMIT 1
            ''')
        except sqlite3.OperationalError:
            print("Error occurred selecting data")
            return
        data = cursor.fetchall()  # assigns this data to a variable
        string = str(data)[4:85]  # gets the appropriate string
        self.split(string)  # reverts it back into a 2D array
        self.changeGrid(self.newGrid)

    def hardDiff(self):
        try:
            # fetches a puzzle from hard table
            cursor.execute(f'''
            SELECT Data FROM 'Hard'
            ORDER BY RANDOM()
            LIMIT 1
            ''')
        except sqlite3.OperationalError:
            print("Error occurred selecting data")
            return
        data = cursor.fetchall()  # assigns this data to a variable
        string = str(data)[4:85]  # gets the appropriate string
        self.split(string)  # reverts it back into a 2D array
        self.changeGrid(self.newGrid)


    def puzzles(self):
        try:
            # fetches a puzzle from Puzzles table
            cursor.execute(f'''
            SELECT Data FROM 'Puzzles'
            ORDER BY RANDOM()
            LIMIT 1
            ''')
        except sqlite3.OperationalError:
            print("Error occurred selecting data")
            return
        data = cursor.fetchall()  # assigns this data to a variable
        string = str(data)[4:85]  # gets the appropriate string
        self.split(string)  # reverts it back into a 2D array
        self.changeGrid(self.newGrid)
```

BUTTONCLASS

```
import pygame

# ------------------------------------------------------------------------------------------------
# Button Class
# ------------------------------------------------------------------------------------------------

class Button:

    def __init__(self, x, y, width, height, text=None, colour=(180, 180, 180), highlightedColour=(102, 102, 102),
                 function=None):

        self.image = pygame.Surface((width, height))
        self.position = (x, y) # coordinates used for drawing buttons
        self.rect = self.image.get_rect() # shape of buttons
        self.rect.topleft = self.position # top left of buttons
        self.text = text
        self.colour = colour
        self.highlightedColour = highlightedColour
        self.function = function # function to be called
        self.highlighted = False # indicates if mouse is hovering over button
        self.width = width
        self.height = height

    def update(self, mouse):
        # contains code used to indicate if mouse is hovering over button
        if self.rect.collidepoint(mouse):
            self.highlighted = True
        else:
            self.highlighted = False

    def draw(self, window):
        # draws the buttons
        if self.highlighted:
            # highlights button if mouse if hovering over it
            self.image.fill(self.highlightedColour)
        else:
            # highlights button with its corresponding colour
            self.image.fill(self.colour)
        # blits button onto screen in the right position
        window.blit(self.image, self.position)

        #if self.text:
        # draws text on buttons as well
        self.drawText(self.text)
        window.blit(self.image, self.position)

    def click(self):
        # calls the function when you click the button
        self.function()


    def drawText(self, text):
        # contains code for drawing text on buttons
        font = pygame.font.SysFont("system", 24, bold=1)
        text = font.render(text, False, (50, 50, 50))
        width, height = text.get_size()
        # returns tuple, first width and then height
        x = (self.width - width) // 2
        y = (self.height - height) // 2
        # working out the coordinates for positioning the text properly
        self.image.blit(text, (x, y))
        # blits text onto screen
```

DATABASE

```
import csv
import sqlite3
from settings import *

# establishing a connection to the database
connection = sqlite3.connect('database.sqlite')
cursor = connection.cursor()

# opening the csv file which stores the puzzles
with open('SudokuData.csv', 'r') as csvfile:
    # reading the csv file
    reader = csv.reader(csvfile)
    puzzles = []
    for line in reader:
        puzzles.append(line)

puzzles = puzzles[2:] # data for puzzles table
easy = puzzles[0:10] # data for easy table
medium = puzzles[10:20] # data for medium table
hard = puzzles[20: 30] # data for hard table

# ------------------------------------------------------------------------------------------------------
# Database Functions - Creating Tables
# ------------------------------------------------------------------------------------------------------

def createPuzzles():
    # creating the puzzles table
    try:
        cursor.execute('''
            CREATE TABLE Puzzles (
                ID int NOT NULL,
                Level STRING,
                Data STRING
            )
        ''')
        connection.commit()
    except sqlite3.OperationalError:
        # runs exception if table already exists
        print("Puzzles has already been created")
        return

def createEasy():
    # creating the easy table
    try:
        cursor.execute('''
            CREATE TABLE Easy (
                ID int NOT NULL,
                Data STRING
            )
        ''')
        connection.commit()
    except sqlite3.OperationalError:
        # runs exception if table already exists
        print("Easy has already been created")
        return
```

```
def createMedium():
    # creating the medium table
    try:
        cursor.execute('''
            CREATE TABLE Medium (
                ID int NOT NULL,
                Data STRING
            )
        ''')
        connection.commit()
    except sqlite3.OperationalError:
        # runs exception if table already exists
        print("Medium has already been created")
        return

def createHard():
    # creating the hard table
    try:
        cursor.execute('''
            CREATE TABLE Hard (
                ID int NOT NULL,
                Data STRING
            )
        ''')
        connection.commit()
    except sqlite3.OperationalError:
        # runs exception if table already exists
        print("Hard has already been created")
        return

# ---------------------------------------------------------------------------------------------------
# Database Functions - Inserting Into Tables
# ---------------------------------------------------------------------------------------------------

def addPuzzles():
    # adding data into the puzzles table
    for puzzle in puzzles:
        # setting values to insert
        id = int(puzzle[0])
        level = str(puzzle[1])
        data = str(puzzle[2])
        # executing command in SQl
        cursor.execute(f'''
            INSERT INTO Puzzles (ID, Level, Data)
            VALUES ('{id}', '{level}', '${data}');
        ''')
        connection.commit()
    cursor.close()
    connection.close()
```

```
def addEasy():
    # dding data into the easy table
    for puzzle in easy:
        # setting values to insert
        id = int(puzzle[0])
        data = str(puzzle[2])
        # executing command in SQL
        cursor.execute(f'''
            INSERT INTO Easy (ID, Data)
            VALUES ('{id}', '${data}');
        ''')
        connection.commit()
    cursor.close()
    connection.close()

def addMedium():
    # adding data into the medium table
    for puzzle in medium:
        # setting values to insert
        id = int(puzzle[0])
        data = str(puzzle[2])
        # executing command in SQL
        cursor.execute(f'''
            INSERT INTO Medium (ID, Data)
            VALUES ('{id}', '${data}');
        ''')
        connection.commit()
    cursor.close()
    connection.close()

def addHard():
    # adding data into the hard table
    for puzzle in hard:
        # setting values to insert
        id = int(puzzle[0])
        data = str(puzzle[2])
        # executing command in SQL
        cursor.execute(f'''
            INSERT INTO Hard (ID, Data)
            VALUES ('{id}', '${data}');
        ''')
        connection.commit()
    cursor.close()
    connection.close()
```

SETTINGS

```
# colours used in program
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
GREEN = (0, 200, 0)
TEAL = (174, 227, 245)
LOCKEDCELLCOLOUR = (245, 245, 245)
INCORRECTCELLCOLOUR = (255, 150, 150)

# default board
grid = [[0,0,0,0,0,0,0,0,0],
        [9,0,1,2,5,0,8,4,0],
        [0,4,7,3,8,0,2,9,0],
        [0,0,5,8,3,0,0,2,0],
        [4,0,2,0,0,0,6,0,0],
        [3,8,0,0,0,0,1,5,9],
        [6,1,0,9,7,3,0,0,0],
        [0,0,3,0,2,0,9,6,0],
        [0,2,9,4,0,0,0,0,0]]

# positions and sizes for creating the board
WIDTH = 600
HEIGHT = 600
gridPosition = (75, 100)  # the top left of the grid
cellSize = 50
gridSize = cellSize * 9

# join function joins numbers in a 2-D array format into a string
def join(string):
    string = ""
    for listNumber in range(9):
        for index in range(9):
            string = string + str(grid[listNumber][index])
    print(string)
    # used when adding numbers into csv file
```