

Final Year Project Report

Full Unit - Interim Report

Playing Games and Solving Puzzles Using Artificial Intelligence

Terique Carnegie

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Michail Fasoulakis



Department of Computer Science
Royal Holloway, University of London

December 12, 2024

Declaration

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 Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count:

Student Name: Terique Carnegie

Date of Submission:

Signature: Terique Carnegie

Table of Contents

Abstract	3
1 Introduction	4
1.1 Artificial Intelligence Solving a Sudoku	4
1.2 Purpose of my Application	4
1.3 Project Milestones	4
2 Application Development	5
2.1 Development of Terminal Interface	5
2.2 Development of Backtracking Algorithm	6
2.3 Development of Graphical User Interface	7
3 Current Progress	9
3.1 Application Architecture	9
3.2 Running the Application	9
4 Next Objectives	11
4.1 Development of Neural Network solution	11
4.2 Further Improvements planned for the GUI	11
5 Conclusion	12
Bibliography	13
Appendix	14

Abstract

The general purpose of artificial intelligence is to give computers the ability to replicate human intelligence through the use of problem-solving algorithms. AI has the potential to outperform human intelligence in solving many different and difficult problems due to its ability to “think” by running problem solving computations at much greater speeds and accuracy than a human and it does this through leveraging the superior processing capabilities of a computer [1].

Due to the nature of problem-solving algorithms, Artificial Intelligence is able to provide assistance and insight into many different fields of expertise making it popular and very much in demand as an asset to humanity. In order to increase the value of AI, it is necessary to deeply research ways to improve problem solving algorithms and to discover which of the algorithms works best for a specific problem, and it is also important to develop methods for humans and AI to safely interact. One way this can be achieved is through games and puzzles. Using games and puzzles as a means to research and further develop AI can be very beneficial as it allows for problem solving algorithms to be tested against fun solvable challenges which can provide insight on how well the algorithms are able to solve problems, using games can also provide a way for humans to interact with artificial intelligence through either testing the speed and efficiency of both the player and the algorithms or through the use of 2 player games like chess and checkers. This type of approach to using artificial intelligence has been seen in the past from the first working checkers program to appear in 1952 [3], and chess playing programs being developed shortly thereafter [2] which had a role in furthering research and development of modern algorithms and uses of Artificial Intelligence.

Chapter 1: Introduction

1.1 Artificial Intelligence Solving a Sudoku

Artificial Intelligence has been applied to many different fields of expertise and has had many advancements in their problem solving algorithms. Some of the more modern and complex problem-solving AI algorithms that can be engineered to play and solve challenges in games are backtracking algorithms, constraint satisfaction algorithms, search algorithms, minimax algorithms, Monte Carlo Tree Search, reinforcement learning and neural networks. All of these algorithms allow for AI to be able to play games and potentially interact with human players, however for the sudoku solver, the project will focus on and utilise only a few of these AI algorithms. The selected algorithms for solving this problem will be back tracking and neural network algorithms. The other problem-solving algorithms are capable of providing a solution but my reasonings for my selection will be explained in the following paragraphs.

There are several different types of backtracking algorithms, such as consistency pruning, looking ahead, efficient backtracking and variable and value heuristics, but the basic principle behind backtracking algorithms is to explore all possible solutions through attempting different paths recursively and regressing back to try a different path when the current is no longer deemed viable. A sudoku can be solved by assigning numbers to an empty square but validating the number before to check if it is safe to assign, however this kind of approach has a worst-case time complexity of $O(9^{n \cdot n})$ [4].

Neural networks like convolutional neural networks and recurrent neural networks can be used to solve problems through pattern recognition. It does this by imitating how the human brain works through layers of interconnected nodes that takes in data, performs computations on the data and provides a prediction based on the data analysed. This can be applied to solving a sudoku as if trained on the rules and sequences of numbers of sudoku training data, the algorithm could be able to predict the numbers that will fit sudoku sequence with up to 99% accuracy [6].

1.2 Purpose of my Application

Through this project I plan further the understanding of how Artificial Intelligence can be applied to solve games and puzzles with logical thinking and pattern recognition by developing a platform using the Python programming language for developing my AI algorithms, with the pygame library for the development of the Graphical User interface and the Visual Studio code IDE to effectively manage my work files, to aid in my projects goal of demonstrate the capability of Artificial intelligence and how it can quickly assess and solve puzzles and problems in a gaming environment, by creating multiple problem-solving algorithms which will tackle and solve sudoku puzzles of increasing sizes and complexity, and evaluating which of the following algorithms between both backtracking and neural networks is better for solving a sudoku puzzle efficiently and potentially faster than humans in most cases.

1.3 Project Milestones

Chapter 2: Application Development

2.1 Development of Terminal Interface

The main focus for the start of the sudoku solver needs to be the interface for displaying the sudoku. This is a necessary starting point as in order to demonstrate Artificial Intelligence solving the sudoku, the product needs to be able to display both the unsolved and solved solution in a way that is easily digested by the target audience. The borders, regions, numbers and void values need to be distinct and observable to those trying to understand what the AI algorithms is trying to achieve, and if the algorithms are doing so correctly. That makes this element of the product integral to the development of my project.

The best approach to initially work on this feature would be to work on a console based interface. This would be effective as the first iteration of the sudoku interface as it only needs to display an easily readable representation of the sudoku puzzle. This was achievable in the console as the borders can be represented through the '-' character for row division and '|' character for column division, the numbers as just numbers and the void values as an '.' character. this gave a simple and easy to read sudoku display on the console.

Division of the grid into sub-grids in the terminal would work through calculating the size of the sub-grids which can be based upon the size of the inputted grid as in most cases the length of the sub-grid is the square root of the length of the grid, for example, a 9x9 grid would have 3x3 sub-grids or a 16x16 would have a 4x4 sub-grids. However, in some cases a sudoku can have sub-grids with unequal length rows and columns, such as a 6x6 grid which has 2x3 sub-grids or a 12x12 which has 3x4 sub-grids. For the solver to be able to adapt so that it can print and solve sudokus of different sizes the application would need to be able to work out the correct row and column length based of the grid size.

A way to approach the interface algorithm is to calculate the square root of the grid length, and storing this value as a row and column divisor, however in the case where the square root is not a whole number, the algorithm rounds the value down for the row divisor and rounds it down for the column divisor but adds 1, which in most cases is enough to find the correct sub-grid lengths, as in the case of a 6x6 grid, the algorithm would calculate the root to 2.449... and round it down to 2 for the row divisor and set the value to 3 for the column divisor, creating the 2x3 sub-grids necessary for displaying the puzzle.

The divisors would be used by the algorithm to print a series of '-' characters for every row which is fully divisible by the row divisor, which separates the rows in the sub-grids. And for the columns, the algorithm would print the '|' character in the column index which is fully divisible by the column divisor. With this algorithm, the application is capable of taking in sudokus in the form of a 2 dimensional array and displaying a user friendly way of observing both an unsolved and solved sudoku problem.

A look at the final sudoku printing algorithm:

```
def print_grid(self, grid):
    if(grid == None):
        print("No found solution!")
        return
    divisor = grid.__len__() ** 0.5
    if(divisor%1!=0):
        row_divisor = round(divisor)
```

```

        column_divisor = round(divisor)+1
    else: row_divisor, column_divisor = divisor, divisor
    # logic for dividing sudoku grid into regions depending on sudoku length
    for i in range(grid.__len__()):
        if i % row_divisor == 0:
            print(" - " * grid.__len__())
            # uses dividing value to place row borders
            row=""
            for j in range(grid[i].__len__()):
                if(j % column_divisor ==0):
                    row+= " | "
                    # uses dividing value to place column borders
                row+=" "+str(grid[i][j]) if grid[i][j]!=0 else " *"
                # adds number or * to be printed for final sudoku output
            print(row+" | ")
    print(" - " * grid.__len__())
    # printing the numbers of grid list or an * where there is a 0
    # and the borders to display the sudoku in the console to the user

```

2.2 Development of Backtracking Algorithm

2.2.1 What is Backtracking?

2.2.2 Implementation of Backtracking Algorithm

A backtracking algorithm works similar to how a human would use backtracking as once a human discovers a mistake, an approach to fixing it would be to go back on their working out and find the cause of fault which gives them the undesirable output. The algorithms would work by making its own attempt to solve a problem. so in the case of a sudoku it would attempt to fill in blank squares with a potential valid value, and if a problem occurs in its solution later on, such as the next blank square not being able to find a single valid value, then the algorithm would need to progress back to the cause of the problem, which would be the incorrect value in the previous square, or the ones before until it finds a different valid value to progress. The backtracking algorithm will then continue its attempt with a different solution for the sudoku, and this is done repeatedly till a viable solution is formed.

When developing the solving algorithm, first came the crucial element of finding a blank square in the grid, this would be represented by a 0 in the sudoku grid's 2 dimensional array. the coordinates on the grid would be returned to the main algorithm when found, then using these, the main loop will attempt to fill the square with numbers ranging from 1 to the max value, which is also the length of the grid (usually 9), until a viable value for the square is found. In order to check the validity of a value, the algorithm would exhaustively search through the row, column and sub-grid (checking every value in that area) for the value that is equal to the value about to be inputted into the blank square if the value is not found in any of the specified areas then it is considered valid, if the same number is found, then the current value is invalid and the algorithm attempts to check the next value for validity. If a viable value cannot be found, the algorithm clears the current square and backtracks to the previous square, and checks the next values for viability, if there are no viable values, the algorithm will clear the current square and backtrack once again to the square before and once again try different values until a different viable value is found and the algorithm

searches for the next blank value and the algorithm repeats itself, until there is no more blank squares, or no more values to try on the grid that gives a solution to the problem.

A look at the main Backtracking algorithm:

```
def backtracking(grid, sudoku):
    position = empty_space(grid)
    # locates an empty square on the grid and stores its location
    if position == None:
        return grid
    # if there are no empty squares, the solution has been
    # found and the grid is returned
    for i in range(1, grid.__len__()+1):
        sudoku.update(i, position[0], position[1])
        # updates the GUI to display the current attempt for a viable value
        if safe_value(grid, position, i):
            grid[position[0]][position[1]] = i
            # test a viable number in the open position
            backtracking(grid, sudoku)
            # checks the next space after the previous viable solution
            if(empty_space(grid)==None):
                return grid
            # if there are no empty positions, this solution is correct
        grid[position[0]][position[1]] = 0
        # if the current value is not viable, it is set to 0, but the i remains
        # as the none viable solution to be incremented
        sudoku.update(0, position[0], position[1])
        # updates the GUI to clear the current square
    return None
# if there is no number that goes into the empty position,
# an empty grid is returned as there is no solution
```

2.3 Development of Graphical User Interface

For the development of the graphical user interface to display the sudoku and the solving algorithm, pygame was a powerful python library to leverage in the programming of the GUI as it provides an adaptable environment in which the sudoku grid can be drawn on the window and updated with little difficulty while still being able to display a way of visualising the backtracking algorithm. When creating the grid on the window it was best to give it the length 340 which was big enough for the user to see the sudoku and the values in each square clearly. When dividing the grid into sub-grids, the same root and rounded divisor logic used in the console print function, however in the case of the GUI the grid was divided visually using thicker lines to create the division between the whole and sub grids. Then after the grid has been drawn in the window, the algorithm starts adding values to square by placing a new square over the empty square with the intended value.

It is also necessary to be able to clear the squares within the grid, this is so that the sudoku solver will be able to show its iterations of the solution. In the updating squares method, the squares cleared through overwriting the square by filling the surface with a white square then updating the value to a new square to cover the cleared square. this function would need to be called every time the value changed in the algorithm in order to give the visual backtracking effect.

By introducing a GUI to display the solving algorithm, the solver is able to show the step by step solution being formed through mistake adaptation approach my backtracking algorithm takes as well as showing how quickly the computer can calculate this as the GUI updates the values very rapidly, demonstrating how much faster a computer is compared to a human, despite the GUI technically wasting computer processing speed, so it is faster to print the output directly then updating a GUI, but then the algorithm will not be demonstrated in an easily comprehensible manner when being displayed in that manner.

The main loop for the displaying the grid in the GUI

```

for row in range(grid.__len__()):
    self.board.append([])
    # adds a array inside another array, representing rows on the board
    if(row%row_divisor==0 and row != 0):
        pygame.draw.line(self.screen, (0, 0, 0),
            (col_offset, row*square_size),
            (col_offset+(grid.__len__()*square_size), row*square_size), 8)
        # where the board is divided based on divisor,
        # thicker lines are used to show the divisions that
        # make up the sub-grids
    for col in range(grid.__len__()):
        square = pygame.Rect(col_offset+(col*square_size),
            row*square_size, square_size, square_size)
        value = self.font.render("" if grid[row][col] == 0
            else str(grid[row][col]), True, (0, 0, 0))
        val_rect = value.get_rect()
        val_rect.center = square.center
        self.screen.blit(value, val_rect)
        self.board[row].append(square)
        pygame.draw.rect(self.screen, (0, 0, 0), square, 1)
        # the squares are created within the boundaries of the grid,
        # and smaller rectangles with the preset
        # puzzle values are displayed within the squares where their
        # coordinates are found from the input array
        if(col%column_divisor==0 and col != 0):
            pygame.draw.line(self.screen, (0, 0, 0),
                (col_offset+(col*square_size), row*square_size),
                (col_offset+(col*square_size), grid.__len__()*square_size), 8)
            # where the board is divided based on divisor, thicker lines are used
            # to show the divisions that make up the sub-grids

```

Chapter 3: Current Progress

3.1 Application Architecture

3.2 Running the Application

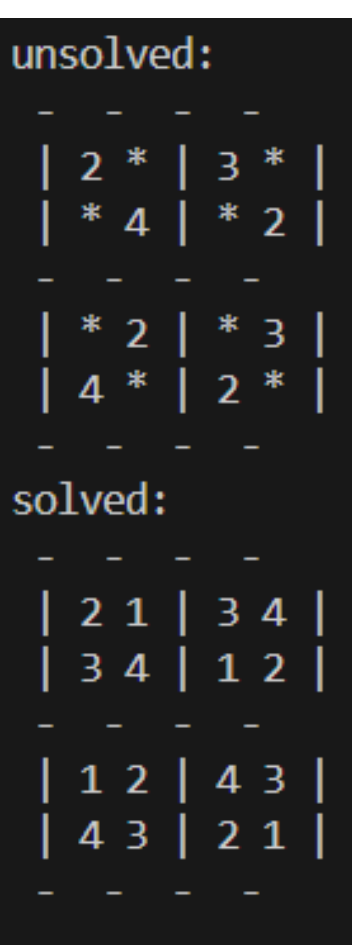


Figure 3.1: 2x2 grid

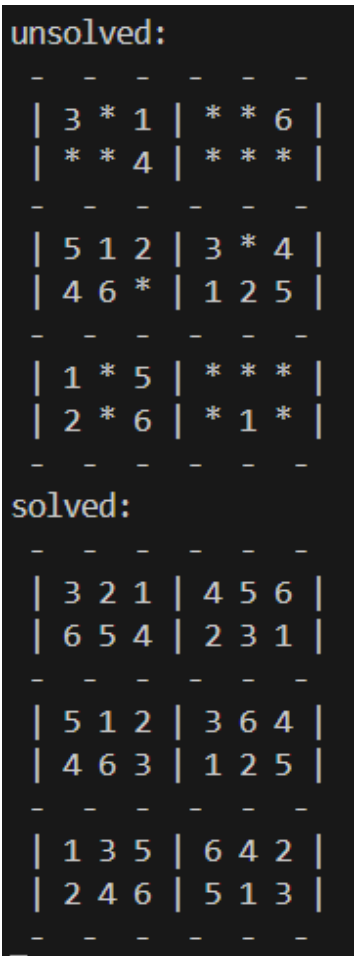


Figure 3.2: 6x6 grid

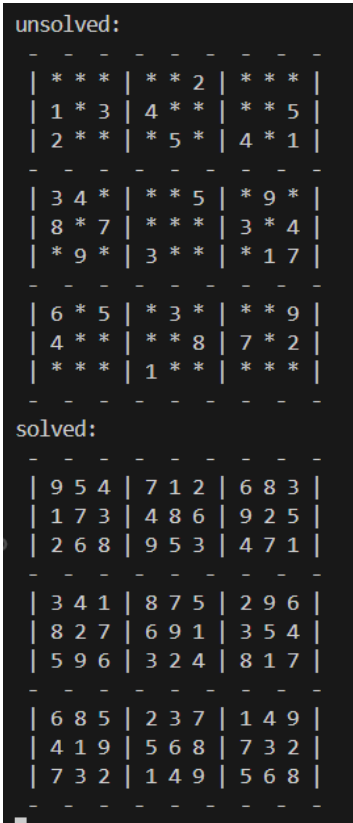


Figure 3.3: 9x9 grid

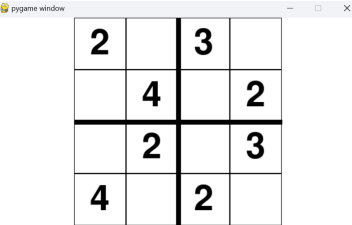


Figure 3.4: 2x2 grid

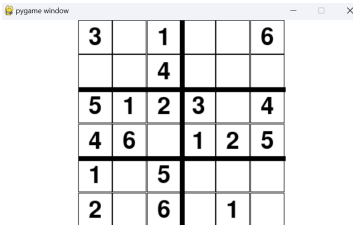


Figure 3.5: 6x6 grid

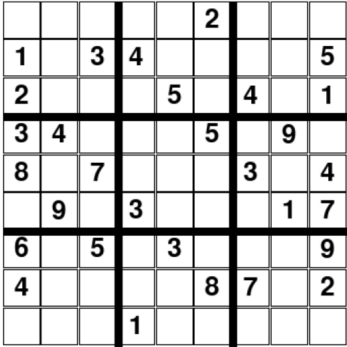


Figure 3.6: 9x9 grid

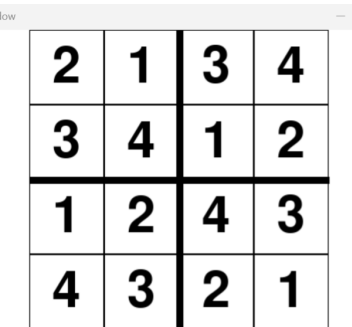


Figure 3.7: 2x2 grid

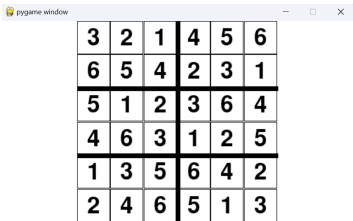


Figure 3.8: 6x6 grid

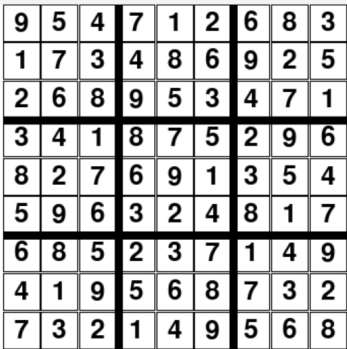


Figure 3.9: 9x9 grid

Chapter 4: Next Objectives

4.1 Development of Neural Network solution

4.1.1 What is a Neural Network?

4.1.2 How can this be Implemented into my Application?

The next focus of the project moving forwards will be on developing a Neural Network that will be trained on taking in sudoku solutions, this is so that it will learn what a solution to a sudoku should look like and it should recognise the patterns/rules of a sudoku so that when it is faced against an incomplete sudoku, it will be able to provide an accurate prediction of a solution, that will hopefully be the correct answer to the inputted problem.

With the use of a neural network, there is a possibility of overfitting, which is when the network produces great results when working within the training data, but when introduced to a new puzzle outside of the training set, the algorithm holds poor results. This is a risk to consider when developing the AI sudoku solver as poor results will impact the validity of the project results which will need to be able to show the effectiveness of artificial intelligence when implementing it in an environment to play games and solve puzzles. To mitigate this risk, it would be important need to dedicate time in the next term to researching solutions to overfitting, such as pruning, cross validation or data augmentation.

4.2 Further Improvements planned for the GUI

Once the other features of the project have become slightly more developed, the next planned step is to return to improve the sudoku interface through various means, such as randomised sudoku generation of varying difficulty, as well as making graphical changes to utilise components such as radio buttons or checkboxes to select which algorithm to use when solving the puzzle, also implementing a reset button to clear the grid and a start button to start the algorithm would help to make the application experience more user friendly and convenient. Additional text would also be needed, and useful for displaying the time it takes for solving the problem as well as potentially a mistakes counter for how efficiently/accurately it initially solves the problem.

Chapter 5: **Conclusion**

Bibliography

- [1] AI for Social Good (n.d.). Artificial Intelligence vs. Human Intelligence: Exploring the Debate and Key Points. [online] Available at: <https://aiforsocialgood.ca/blog/artificial-intelligence-vs-human-intelligence-exploring-the-debate-and-key-points> [Accessed 8 Oct. 2024].
- [2] C. Strachey, Logical or non-mathematical programmes, in: Proc. Association for Computing Machinery Meeting, Toronto, ON, 1952, pp. 46–49.
- [3] J. Kister, P. Stein, S. Ulam, W. Walden, M. Wells, Experiments in chess, J. ACM 4 (1957) 174–177.
- [4] GeeksforGeeks (n.d.). Sudoku — Backtracking-7. [online] Available at: <https://www.geeksforgeeks.org/sudoku-backtracking-7/> [Accessed 9 Oct. 2024].
- [5] Simonis, H., n.d. Constraint Programming in Action. [pdf] Available at: https://ai.dmi.unibas.ch/_files/teaching/fs21/ai/material/ai26-simonis-cp2005ws.pdf [Accessed 10 Oct. 2024].
- [6] Charles Akin-David, Richard Mantey, n.d. Solving Sudoku with Neural Networks. [pdf] Available at: https://cs230.stanford.edu/files_winter_2018/projects/6939771.pdf [Accessed 10 Oct. 2024].

Appendix

Sudoku Interface

Importance of Interface Implementation

The main focus for the start of my project needs to be the interface for displaying the sudoku. this is a necessary starting point as in order to demonstrate Artificial Intelligence solving the sudoku, my product needs to be able to display both the unsolved and solved solution in a way that is easily digested by the target audience. the borders, regions, numbers and void values need to be distinct and observable to those trying to understand what the AI algorithms is trying to achieve, and if the algorithms are doing so correctly. that makes this element of the product integral to the development of my project.

Implementation of Sudoku Interface

In order to implement this feature, I have decided to initially work on a console based interface. I felt this was necessary as my first iteration of the sudoku interface only needs to display an easily readable representation of the sudoku puzzle. this was achievable in the console as the borders can be represented through the '-' character for row division and '|' character for column division, the numbers as just numbers and the void values as an '*' character. this gave a simple and easy to read sudoku display on the console.

Planned Improvements to the Interface

Once the other features of my project have become slightly more developed, I plan to return to improve the sudoku interface through various means, such as randomised sudoku generation of varying difficulty, as well as a better user interface to display the sudoku rather than primarily using the console, which works well and is readable, but is not the best means of displaying the puzzle in a visually appealing and user friendly fashion.

Solving Algorithm

How a Backtracking Algorithm could Solve a Sudoku

A backtracking works similar to how a human would back track. The algorithms makes an attempt to solve a problem. so in the case of a sudoku it would attempt to fill in blank squares with a potential solution, and when a problem occurs in its solution, it goes back to the cause of the problem, which would be the invalid potential solution in that square or a previous one, and continues its attempt with a different solution for the sudoku, and this is done repeatedly till a viable solution is formed.

Implementation of Backtracking Algorithm

When developing the solving algorithm, first came the crucial element of finding a blank square in the grid, this would be represented by a 0 in the sudoku grid array. the coordinates on the grid would be returned to the main algorithm when found, then using these, the main loop will attempt to fill the square with numbers ranging from 1 to the max value, which is

also the length of the grid (usually 9), until a viable value for the square is found (a value that is not repeated in the row, column or sub-grid). if a viable value cannot be found, the algorithm clears the current square and backtracks to the previous square, and checks the next values for viability, if there are no viable values, the algorithm will clear the current square and backtrack once again to the square before and once again try different values until a different viable value is found and the algorithm searches for the next blank value and the algorithm repeats itself, until there is no more blank squares, or no more values to try on the grid that gives a solution to the problem.

Planned Improvements for the Backtracking Algorithm

In order to improve the speeds of my backtracking algorithm, I would need to look into my uses of exhaustive searches which is used a lot in my algorithm for different functionalities such as, finding an empty square by looping through every value in the grid array, checking validity of a value by searching through every value in the row, column and sub-grid for a repeated value. this is very inefficient, and I believe this can be rectified by maybe implementing faster search algorithms or making use of other methods and techniques to make the searches slightly quicker and efficient, which is necessary to prove AI's superior problem solving capabilities.

Improvement of the User Interface

Creating a Graphical User Interface

When creating my graphical user interface to display the sudoku and the solving algorithm, I decided to pygame as it provides an adaptable environment in which the sudoku grid can be drawn on the window and updated with little difficulty while still being able to display a way of visualising the backtracking algorithm. when creating the grid, I decided to give it the length 340 which was big enough for the user to see the sudoku and the values in each square clearly. When dividing the grid into sub-grids I used the same divisor logic used in the console print function, however I divided the grid visually using thicker lines to create the division between the whole and sub grids. I then started adding values to square by placing a new square over the empty square with the intended value, this could be cleared through overwriting the square by filling the surface with a white square then updating the value to a new square to cover the cleared square. this function would need to be called every time the value changed in the algorithm in order to give the visual backtracking effect.

The Benefits of Having a GUI to Display the Problem Solving Algorithm

By introducing a GUI to display the solving algorithm, I am able to show the step by step solution being formed through mistake adaptation approach my backtracking algorithm takes as well as showing how quickly the computer can calculate this as the GUI updates the values very rapidly, demonstrating how much faster a computer is compared to a human, despite the GUI technically wasting computer processing speed, so it is faster to print the output directly then updating a GUI, but then the algorithm will not be demonstrated in an easily comprehensible manner.

Planned Improvements to the GUI

In order to improve the Graphical User Interface I plan to make use of radio buttons or checkboxes to select which algorithm to use when solving the puzzle, I also feel a reset button to clear the grid and start button to start the algorithm would also be a nice touch, and will make the application experience more convenient. Additional text would also be needed, and useful for displaying the time it takes for solving the problem as well as potentially a mistakes counter for how efficiently/accurately it initially solves the problem.

Next Solving Algorithm

How could a Neural Network Solve a Sudoku

My focus moving forwards will be on developing a Neural Network that will be trained on taking in sudoku solutions, this is so that it will learn what a solution to a sudoku should look like and it should recognise the patterns/rules of a sudoku so that when it is faced against an incomplete sudoku, it will be able to provide an accurate prediction of a solution, that will hopefully be the correct answer to the inputted problem.