Final Year Project Report

Full Unit - Interim

Playing Games and Solving Puzzles Using AI

Luke Sell

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Iddo Tzameret

Department of Computer Science

Royal Holloway, University of London

December 13, 2019

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 Where other published and unpublished source materials have been used, these have been acknowledged.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: 5389 - Inclusive

Student Name: Luke Sell

Date of Submission: 13/12/2019 - Extension

Signature: l.sell

Table of Contents

Abstract

Introduction and Background

Sudoku is a logical puzzle where the goal is to fill an incomplete grid with numbers so that each row, column and square has all numbers up to the size of that grid. These simple rules allow Sudoku puzzles to be seen as a Constraint Satisfaction problem and thus they can be generated and solved significantly quicker by Artificial Intelligence using Constraint Satisfaction algorithms than by people.

To produce an incomplete Sudoku puzzle, a generator is required so that numbers are placed in a grid, where the amount of numbers given relates to the difficulty of that puzzle, in such a way that there is only one solution. To prove that a generated Sudoku puzzle not only has a solution, but a unique one, a solver is required to complete the puzzle and find this solution. It would also be useful to see and modify the Sudoku puzzle on a computer, so that the user of that application can solve the puzzle themselves.

Aims and Objectives

The aim of this Project is to show how AI can be used to solve a puzzle, in this case Sudoku, a puzzle easily solvable by AI, there is only one solution and three rules. Therefore Sudoku can be modelled as a Constraint Satisfaction Problem, where one state can is not compatible with another. A simple algorithm would be able to search for this solution and backtracking if any of the states is not valid, constraint propagation allows such algorithms to solve puzzles very quickly and efficiently.

Overview of Early Deliverables

Proof of Concept Programs

A Main Menu GUI for the Sudoku program was created using C++ and the framework SDL 2.0 using [3], this involved setting up the GUI window in the main method to display the screen, with buttons that link to other screens.

A priority queue was not needed for the current implementation of the program, but one might be used for dynamic variable ordering, for this I am going to use the STL priority queue and extend it with reordering if necessary.

Solving the Eight Queens problem first involved using the existing GUI and extending it to display the Chess Board, then the solver used a backtracking search whilst checking constraints to find the solution quickly and efficiently.

I started development of my AI solver by first solving the N Queens problem, this involved creating a backtracking algorithm that would find a state where the Queens could be placed on the chessboard without being able to attack each other.

This algorithm would start by attempting to place a Queen on the chessboard and then check these constraints and if this was valid it would move on to the next square if not it would remove the queen and then try the next square, if the algorithm had backtracked to an already valid Queen it would then remove it, as this state would not allow for all the queens to be placed.

For Sudoku, I started by displaying the grid on the screen, to do this I rendered each square in its respective position and then rendered any overlays afterwards. The user is then able to select squares on the grid and enter any number, so that they can attempt to solve the puzzle themselves.

To solve Sudoku I used a similar algorithm to the N Queens problem, once again a solution can be found using a backtracking algorithm and checking the constraints are satisfied, every square must be filled in with a number between one and nine. This is done by trying the first possible number for a square then moving on to the next square in the grid, then backtracking if a number is not valid and trying the next number until all numbers have been tried or a solution has been found.

This was then extended so that the constraints are modelled with the possible values of a square being used and then eliminating numbers from these possible values as numbers are filled in on the grid.

An inefficient Sudoku solver was created to solve a Sudoku puzzle of easy difficulty, this involved using a backtracking search to attempt to fill the partially completed grid with possible numbers until a valid solution is found.

Reports

A report on the Design Patterns that can be used within the AI searching algorithm and other parts of the program was written, this involved researching Design Patterns such as Iterator, Flyweight, Strategy, Memento and other Design Patterns using [4], this was then used to refactor the program code and this will be continued for the final deliverable.

A report on Backtracking searching and recursion was written to help design the AI solving algorithm for the Sudoku puzzle, this covered how backtracking works, as well as the time and space complexity and other topics like value ordering and backjumping to make the algorithm more efficient, as well as example code that can be used to better understand how the AI would solve the Sudoku puzzle.

A report on Constraint Satisfaction and Consistency techniques was written as Sudoku can be modelled in this way, by researching CSP algorithms and techniques such as arc consistency an efficient AI solving algorithm can be design to find a solution to Sudoku puzzles of any difficulty very quickly using constraint propagation.

A report was written on the solving techniques used by human solvers to see if this can influence the design of the AI solving algorithm, human solving behaviour is similar to that of CSP algorithms, but other techniques used allows for an extension of the solving algorithm to be designed, that can implement these techniques to solve Sudoku puzzles quicker and more efficiently, as well as estimating the difficulty of Sudoku puzzles for Human Solvers.

A report was written on the Design of the User Interface and User Experience while using the application, this involved drawing mock up UI designs to test different ways to display the Sudoku puzzle, this was helpful for creating the actual application and made sure it is easily usable for a person to attempt to solve Sudoku puzzles and use the AI solver to find the solution to that puzzle.

A report was written on the Complexity and Hardness of the Sudoku puzzle and the solving algorithm, this will involved using O Notation to find the scaling term to see how the algorithm scales with increasingly grid sizes and difficulty, and was useful in designing more efficient algorithms that find the solution quicker so that the AI solver can be used for a wider range of grid sizes and more difficult puzzles.

Overview of Final Deliverables

The Program

The Application will also generate Sudoku puzzles to be solved by a person of the AI solving algorithm, this will involve first filling in an empty grid until it is complete, at the same time as making sure that there is one solution, so that invalid Sudoku puzzles are not generated, and then randomly removing numbers that do not create extra solutions until the smallest possible puzzle has been generated.

The Application will also be able to save and load puzzles so that a person attempting to solve the Sudoku puzzle can continue after exiting and restarting the Application, this will involve converting the grid of numbers into a format that can be written to and read from a save file, this will require the creation of a parser to correctly read the stored text and create entries to be inserted into the grid object.

I will also improve the GUI so that the user can make notes of possible numbers in squares, so that they can keep track of constraint propagation like the solver.

I will also allow the user to modify the application settings to use different screen sizes, use a timer and set different grid sizes.

The Report

The report will also include description of Software Engineering Processes used while coding, this will involve conforming to googles checkstyle, using GitHub correctly as a Version Control System, writing tests for written code and using Design Patterns, it will also include the use of backlogs and task allocation boards like Trello or GitHubs integrated Project board and developing the program using an Agile approach.

The report will also detail and look at examples of other AI algorithms that could have been used to solve Sudoku, such as modelling the puzzle as an exact cover and using the dancing links algorithm.

The report will also discuss the Matrix and Priority queue data structures used to store the Sudoku grid and use backtracking in the AI solver respectively, and how they will affect the performance of the program.

The report will describe and compare Backtracking and Recursion algorithms, this will involve accurately and precisely timing both algorithms on a wide range of Sudoku puzzles of different difficulties and sizes to see which one is the quickest at solving Sudoku.

The report will also describe the Constraint Satisfaction Problem and algorithms to solve it, this will summarise the algorithms that have been used in solving the Sudoku puzzle and involve detailing the multiple algorithms that can be used.

Literature Survey

The book, Foundations of Constraint Satisfaction, has been very useful in learning about Constraint Satisfaction problems and how to solve them, it provides a complete overview of the topic, with detailed descriptions of what a CSP is, examples of algorithms to solve it, as well as optimizations that can be used to extend a solver, and finally explanations on the theory behind this.

Theory

Design Patterns

The following is some of the Design Patterns I found from reading [4] that may be relevant to my Program with a brief summary of their usefulness, as described in [4] , and an explanation of how I will implement them in my program.

Iterator

An iterator allows a way to access the elements of a data structure sequentially without knowing its underlying structure, this simplifies moving through a data structure as the code for knowing the next element is all in one place.

I could use this for the grid class, it stores a two dimensional array of entries that represent the grid of numbers, by using an iterator I would be able to get the next element in a one dimensional way without needing to know how to move from one row to the next, which can be used in the solver to work out which is the next square in the search.

Memento

A memento allows the internal state of an object to be saved and restored later, this means the values of an object fields are stored in the memento which is passed back to the object later when it needs to be restored.

This could be used to store the grid entries constraints, this would allow for the constraint solver to simulate the constraint propagation and reverse it easily if a later modification is not valid.

Observer

An observer is used to keep track of an object state, this way when an object changes state all of the objects observers can be notified of this, this allows a simplification of the relationships between the objects by decoupling them.

This could be used for grid entries and constraint propagation, when a grid entry is updated with a number all entries in the row, column and box can be notified of the new constraints from this number and update the possible numbers for that square appropriately.

State

Having states allows an object to change its behaviour when its internal state changes, this means one object can represent many different behaviours.

This could be used for grid entries, so that when they are selected or locked, their behaviour changes, i.e. a locked entry cannot be selected and a selected entry can be modified.

Strategy

A strategy allows algorithms that perform the same task to be grouped together and be used interchangeably, this would allow multiple algorithms to be designed and used in the program.

This could be used for the solver, the program would have multiple different algorithms that solve the Sudoku, but do it in different ways or more efficiently, this would allow for different optimizations such as dynamic variable ordering to be used and benchmarked against a basic solving algorithm.

Visitor

A visitor allows a different operation to be perform based on the type of object it will be performed on, this means an algorithm can be applied to a data structure which contains objects with different sub types.

This could be used in the solver to represent the different operations to be performed if the entry is an initial number on the grid or an empty square, which would simplify the solving algorithm structure.

Adapter

An adapter or wrapper converts the interface of a class to be usable by another class, i.e. allowing for an int to be added to a data structure that holds objects. This Design Pattern is often used in a similar way to the Facade Design Pattern.

This will be useful in my grid class to store the numbers on the grid as entry objects, while still allowing numbers to be passed as input.

Flyweight

A flyweight allows objects to be created for a highly used type, without using too much memory and time constructing the objects, this is best used when the amount of objects needed is small with each object being reused many times.

I could use this to represent numbers on the grid, i.e. instead of having numbers in an entry as a primitive I can make them into objects allowing for methods to be used on them, this works by creating an instance of each number from one to nine and storing them in a factory object, from which they can be returned. This way I can many of one number on the grid but only one object to represent it.

Bridge

A bridge decouples the abstraction from the implementation, this allows the two to vary independently, i.e. the implementation can be extended without changing the abstraction, this makes it easier to add new sub classes of a parent object.

I could use this to bridge to the correct implementation of a grid, depending on if the user

selects Sudoku or Eight Queens or another puzzle, this would allow me to add different puzzles that sub class from the grid object without changing the code for the abstraction.

Backtracking and Recursion

Sudoku can be solved by backtracking, a variation of a depth first search, Sudoku has an initial and a goal test and actions must be performed by a solver to find the next state, so that the goal state may be found as described in [2]. A backtracking search on a Sudoku puzzle will search up to the depth of 81 if possible before trying other branches, however as the constraints of the puzzle can be checked during the search using forward checking, not all states in a branch have to be checked. This results in an algorithm that is very efficient in both time and memory used, having at most 81 instances stored, and is complete, always finding a solution if there is one. Other search algorithms like breadth first search and iterative deepening search are not efficient at solving a CSP like Sudoku, as the solution will also be at the maximum depth [1]. Optimizations like backjumping can be used so that when a square is found to have no possible values, even though it must if there is a solution, the squares that have constraints on this square can be identified and so the solver can backjump to these and change them without backtracking through other squares that are not relevant [1]. Dynamic Variable Ordering can also be used to order the variables, so that the variables with the least possible values are tried first, as this will give the highest chance of assigning the correct value.

Constraint Satisfaction

A Constraint Satisfaction Problem is modelled as having variables, domains of each variable and constraints between variables as described in [2] and [1], Sudoku can be represented as such with each square being a variable, the domain being the numbers one to nine and the constraints being that a number cannot be repeated in a row, column or box. A CSP can be reduced to make it easier to solve, this involves eliminating redundant values from domains of variables and tightening the constraints, this can eventually result in a minimal problem, although most of the time this will be too costly as doing so is NP hard. To do this consistency techniques are used to eliminate redundant values, . A variable can be made node consistent by eliminated all values in its domain that do not satisfy the variables unary constraints. A variable is arc consistent if all the values in its domain satisfy the variables binary constraints. This would mean that between two variables and for all values in the first variables domain there is a value in the second variables domain that satisfies the binary constraint on the arc joining them. This represents how constraint propagation can be modelled to reduce the possible numbers for all squares on a Sudoku grid and make solving the puzzle significantly easier by reducing the amount of times the solver has to backtrack.

Techniques used by Human Solvers

The most used technique used to solve CSP puzzles like Sudoku is to first work out what is valid, In Sudoku the possible numbers that can go into a square can be worked out by checking the rows, columns and boxes (constraints), using this process to eliminate possible numbers can reduce the numbers to only one, this can then help solve other squares due to

the new constraints, this is constraint propagation, numbers can also be decided by checking if a possible number can go anywhere else, if this is the only square this number can go in the row, column or box then it must go there. For easier puzzles this can find a solution or nearly complete solution alone. If two squares in the same box have the same possible numbers, then those numbers must go into one of those squares, meaning these numbers can be eliminated for other squares in the box, this is known as 'naked twins'. Sometimes these pairs are hidden by other possible numbers a square may be, if the numbers only go in that pair of squares, then the other numbers those squares could be can be eliminated from that square. This method can also be used to eliminate possible numbers from surrounding squares as you know that a number must go in this pair of squares, there are similar techniques to this that eliminate numbers by knowing that one number must be in a specific square.

There are other techniques that are not as useful for easier Sudokus, such as 'X-Wing', 'Swordfish' and 'forcing chain', as explained in [7].

User Interface Design

The User Interface for the application needs to be easy to understand and usable by the user, it should also help them and make clear what the grid is representing so they can solve the Sudoku.

The buttons that the user can use are to the side of the grid with the main grid in the centre, the grid itself has lines drawn to separate the squares with dotted lines to represents the separation between squares within a box. Squares that are not yet filled in are shown as empty rather than their object representation of zero to make it easier for the user to see which squares have not yet been solved. The initial starting numbers given are locked and displayed with a grey overlay so that the user knows which squares these are and that they cannot modify them. The user can also select squares which highlights them in green. When the grid is solved all squares are locked so that the user cannot accidentally modify squares and invalidate the completed grid.

To improve the UI and User Experience when using the application, I will also allow the user to use the arrow keys to move between squares and select them, and will also allow the user to undo a modification to set a square back to a previous number easily. To make it more clear which row, column or box a square is in, I will also highlight those other squares in another colour. I will also allow the user to right click a square to select it for making notes on possible numbers, these can then be modified as the user attempts to solve the Sudoku as the constraints propagate, these would be displayed above the centre of the square in smaller font.
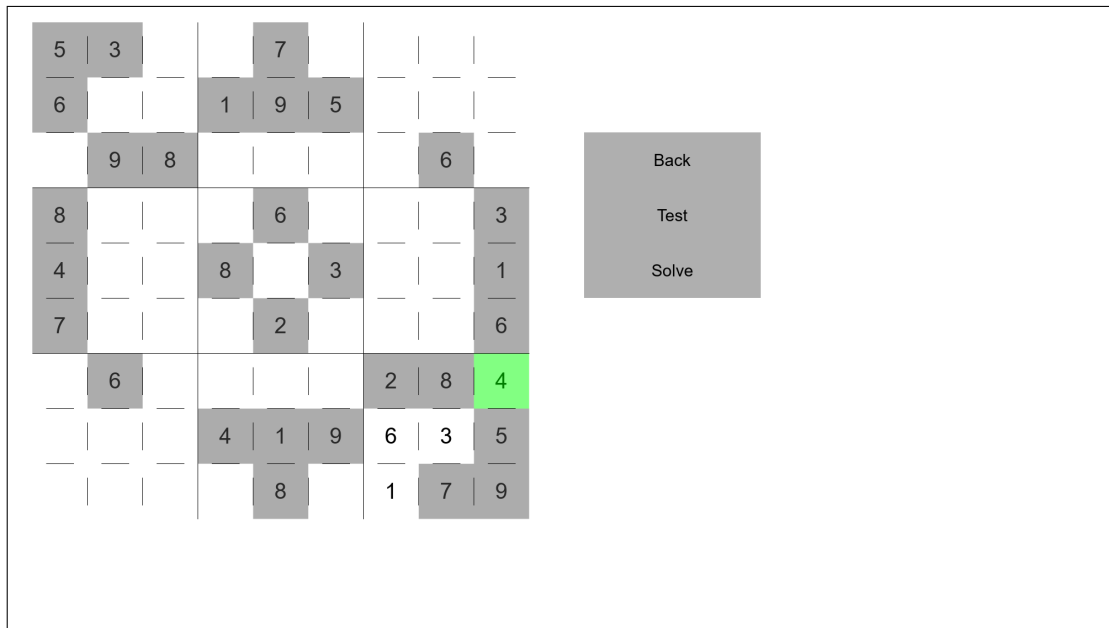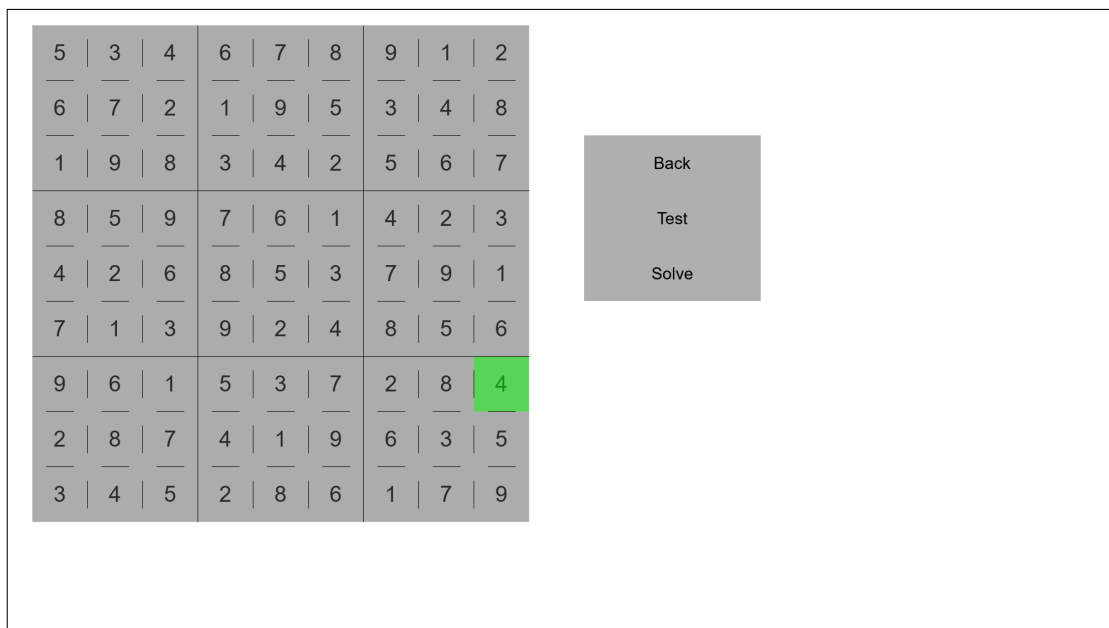
Figure 1:   User Input.

Figure 2:   Solver.

Complexity and NP Hardness

An n^2 x n^2 Sudoku is NP complete, as the problem is in NP, it can be represented as a decision problem, there either being a solution that can be verified in polynomial time or no solution and NP hard, as Sudoku can be represented as a graph colouring problem which is known to be NP hard as there is a polynomial time many-one reduction of this to Sudoku which implies it is NP hard.

Verifying a solution takes $O(M)$ or linear time, for an input of grid size M = n^2 x n^2, as each square in the grid only needs to be checked once to see if its value invalidates any constraints on the rows, columns or boxes.

The problem size is largely related to the entire domain and the constraints have negligible effect [1].

Software Engineering

In development of my program I used the Agile development cycle to quickly produce an initial working main menu GUI and then continued iterating on this, to extend the application to solve the eight queens problem, then displaying the grid for sudoku and allowing user input and finally solving sudoku as well.

To help me develop my program I am using the Code Blocks IDE, as it comes with many tools and plugins that are useful for developing my program in C++, this made it significantly easier to set up SDL as well as produce documentation for my code.

For unit testing I used the catch2 library, as this was the simplest to set up.

I am using googles programming standards from their C++ guide, as well as a plugin to automatically format my code and all warning flags for the compiler.

For tasks I am using githubs project boards to keep track of bugs and new features to be added.

I have also been doing system testing on my release candidate branches, as well as testing the application on other machines.

I have used several design patterns to improve the code, such as the flyweight design pattern to limit the number of objects created for numbers.

Bibliography

[1] Edward Tsang (1993). *Foundations of Constraint Satisfaction.*
As Sudoku can be described as a Constraint Satisfaction problem, it is relevant to research what these are and how to solve them, this book has detailed explanations and examples that will help in solving the Eight Queens problem and using CSP algorithms.

[2] Stuart Russell and Peter Norvig (2016). *Artificial Intelligence: A Modern Approach – Third Edition.*
This book has relevant information and examples of AI algorithms like backtracking search and efficient algorithms like alpha/beta pruning that will be helpful in generating and solving Sudoku puzzles.

[3] Shaun Mitchell (2013). *SDL Game Development.*
As the GUI will be made using SDL 2.0, it is relevant to use this book to see examples of GUI initialisation code and documentation on the framework, which will be helpful in creation the screens to display and solve the Sudoku puzzles.

[4] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.*
This books provides a detailed overview of all Design Patterns, which will be relevant when writing about the Software Engineering Processes used in the program and will be helpful in providing examples of how to refactor program code to include Design Patterns.

[5] Peter Norvig. *norvig.com/sudoku.html.*
This website provides useful examples of the backtracking algorithms and how they should work to solve a Sudoku puzzle of any difficulty.

[6] Wikipedia. *en.wikipedia.org/wiki/Sudoku_solving_algorithms.*
This website provides a good overview of different ways of solving a Sudoku puzzle, which could be used to find other more efficient algorithms to use, such as modelling the Sudoku puzzle as an exact cover problem and finding the solution with the dancing links algorithm.

[7] Kristanix. *kristanix.com/sudokuepic/sudoku-solving-techniques.php.*

Project Diary

The following is a summary of the project diary notes I have made during the development of the project program, as I have not had time to type them up until now. I started development of the program by setting up the repository, this included setting up the SDL libraries in Code Blocks, I had some trouble getting my program to compile at first and spent some time modifying the compiler flags, once this was done I then tested the libraries had been initialised properly by running the program with some example SDL rendering code.

Next using the book SDL Game Development for examples, I set up the main game loop to run the program, then I created a class for the buttons on the main menu, this involved handling events when the user clicks on them, as well as rendering them on the screen, I then added these buttons into the game loop and found a bug where a user could click through buttons, i.e. if a button navigated to another screen, then another button in the same place as the button previously clicked would also be clicked, I did not notice this at first, but later it caused me to think another part of the program was broken, after investigating I fixed this by only handling events on the screen before any navigation, rather than updating the current screen immediately, this way I was not changing a condition for the loop while it was running.

During testing of the program I noticed that the CPU usage was very high, this was clearly because the program was rendering new frames as fast as possible, I decided to limit the frame rate to 60 as this is a common refresh rate for monitors and did not use the CPU nearly as much, but realistically this could be much lower as there are no animations or advanced graphics being displayed.

After this I decided to work on displaying the grid for Sudoku, to start I had to allow navigation between screens and update the rendered buttons as well as handling the events for this screen, I rendered lines to separate the squares and then rendered the values of these squares, when testing that this displayed correctly, the program quickly crashed, as well as my computer, as this did not involve any input from me, it was obviously a memory leak, I confirmed this and found that the number objects texture representations were being created every frame without the last frames texture being destroyed, to fix this I first destroyed the texture before creating a new one, then decided it would clearly be more efficient to just store this texture in the object instead of wasting time creating and destroying textures every frame.

While testing the program and the displaying of the grid, I noticed that the numbers that were being rendered were blurred, I found out that this was because SDL was stretching the number texture to fill the square on the grid, to fix this I just had to render the texture in the centre of the square.

I then handled user events so that the user could select a square by clicking, and then modify it using the number keys.

To improve the User Experience, I changed the GUI so that lines within a box were dotted, as well as implementing overlays to show a selected square as well as locked squares, I also fixed the storing of textures to be rendered, this had previously not been working correctly and was drawing empty squares on the screen for example, I then realised it was because I was using the position on the screen when rendering to the texture that would be stored, but the texture was only exactly the size of what needed to be rendered on it, meaning it would in most cases render outside the texture, I fixed this by creating another position rectangle to render it correctly.

To start on the implementation of the solver for Sudoku, I first needed to be able to check the constraints, so I implemented a method to do this, which counted all the instances of a number in a row, column or box and if there was more than one there would be a conflict, I then tested this on the Sudoku grid using the GUI and modifying squares values to make sure it worked as expected.

Next I implemented the method that would actually solve the Sudoku puzzle, it worked by using a recursive backtracking search, going through all squares on the grid that were not locked, then trying all the numbers from one to nine and checking if there were any conflicts, if there were not it would then move onto the next square, otherwise it would try the next number, once it had tried all numbers it would backtrack as clearly a previous number was not correctly placed. I originally considered copying the grid every time the solver moved onto the next square as a standard recursive algorithm would do, however this clearly would be a costly process in both time and space, considering that the grid contained many entry objects, therefore I decided to partially implement a backtracking search by modifying the grid directly as the solver searched through the grid for the solution.

I then used similar methods and algorithms to solve the Eight Queens problem, adapting the methods as necessary, I considered using inheritance for this as the grid class is similar, but decided against this at the time as the puzzles are sufficiently different, although they have similar GUI displays.

I also added a continue button to the main menu to allow the user to return to the Sudoku puzzle they were previously attempting to solve if they had returned to the main menu, I also improved my code by using list initialisation in constructors as this was previously inefficient and not effective C++, as the compiler was warning me about.

To finish my program ready for the Interim Review, I also implemented some more Design Patterns, this involved making several classes Singletons, as well as implemented the Flyweight Design Pattern for number objects, to reduce the amount of memory be used unnecessarily.