Final Report - 'Killer Sudoku' Solver

CARDIFF UNIVERSITY
PRIFYSGOL CAERDYDD

CM3203 - One Semester Individual Project

Thomas Petty - 1119707

Supervisor: Ralph Martin

Moderator: Richard Booth

Abstract

The main aim of this project was to develop a software application which could solve Killer Sudoku puzzles. An application would be created with which a user could create a puzzle, and then the application would attempt to solve that puzzle. The software would then be tested and analysed using different solving strategies, in an attempt to identify the most effective way of solving a Killer Sudoku puzzle.

Two components of the application were built during the course of this project: a graphical user interface with which the user could build a Killer Sudoku puzzle; a solver component which could apply different rules to solve a given puzzle. This solver component was the subject of a comprehensive experiment, in which different solvers were implemented and tested against a test suite of 100 puzzles of varying difficulty to determine which solving strategies were the most effective.

This report documents the requirements of the software I set out to create, as well as the design and implementation of that software. Furthermore, it also documents the experiments performed to analyse the solver component and evaluates the findings of those experiments.

Acknowledgements

For all of his help and guidance throughout a very challenging project, I would like offer my sincerest thanks to my supervisor Dr. Ralph Martin.

Contents

Figures

1. Introduction

The goal of this project is to create a software application capable of solving Killer Sudoku puzzles (described in detail in section 2). This application would allow a user to construct a Killer Sudoku puzzle digitally, which could be saved and restored to the application. The application would contain a component which would read the given puzzle and attempt to solve it using a variety of methods. The different solving strategies would be analysed to identify which techniques were the most effective and how to order the execution of different strategies to improve efficiency.

1.1 Goal of Implementation

In the implementation phase of this project, I intend to create a single piece of software which allows for the creation and resolution of Killer Sudoku puzzles. The application in question will have an intuitive graphical user interface (GUI), with which a user can easily create a complete Killer Sudoku puzzle. A saving/loading function will be provided to allow users to store puzzles they have created. The application will support two different grid size for the puzzles, 4 x 4 and 9 x 9.

The solving component will attempt to solve the puzzle currently loaded into the application, by combining a number of solving techniques and rules. It will update the GUI to show it's progress towards solving the puzzle, and when the puzzle is solved it will display the solution.

The entire application will be designed and built as a highly extensible object-oriented solution, in which new techniques and strategies can be added, removed or modified with ease.

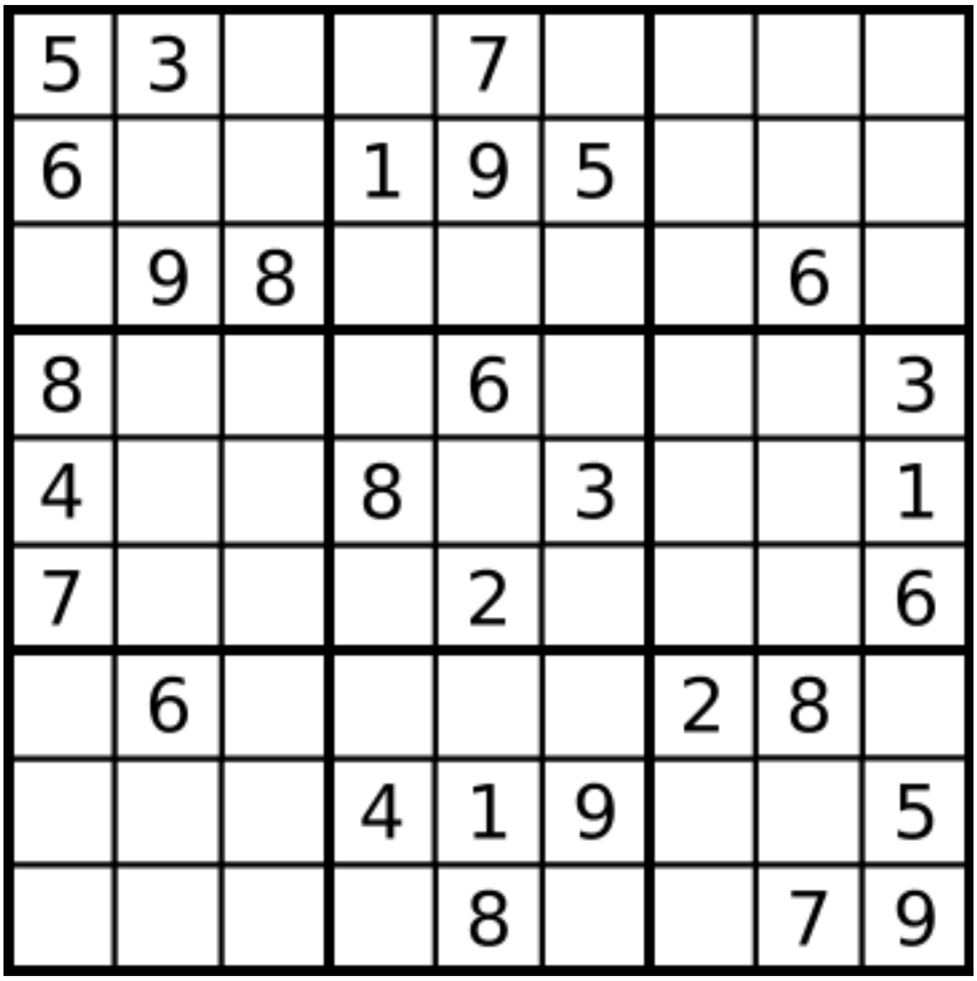
1.2 Goal of Analysis

Once the initial application has been developed, I will carry out experiments to analyse the efficiency and effectiveness of different puzzle solving strategies. The solver component of the application will contain a number of different strategies, rules and techniques which help to solve any given Killer Sudoku puzzle. Tests will be performed, using a test suite of puzzles ranging in difficulty from easy to extreme, in order to analyse the software's ability to solve puzzles when using different combinations of strategies and different orders of execution of different strategies. Results will be recorded in terms of time taken to solve a given puzzle, and they will be evaluated in order to determine which strategies work better.

2. Background

2.1 Classic Sudoku

Sudoku is a logic-based puzzle in which the player must populate a grid of *n x n* cells with values between 1 and *n*. The placement of these values is constrained such that no cells in the same row or column can contain the same value. A Sudoku grid is typically 9 x 9 cells, and in this case each row and each column must contain all the values from 1 to 9 only once. Furthermore, the grid is also divided into *n* square regions, known as boxes. Each square is √*n* *x* √*n* big. Like the rows and columns, all the values in these boxes must be distinct. A unique puzzle is created by placing fixed values into the grid to begin such that there is only one possible solution.[1] An example Sudoku puzzle is given below.



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

Figure 2.1.1 - Example of a traditional Sudoku puzzle[2]

The boxes are the regions with a thick border. The values in this grid are provided at the beginning of the puzzle, and are placed such that there is only one possible solution. The player should be able to use these values and the general constraints of the puzzle to fill in the empty cells.

2.2 Killer Sudoku

A Killer Sudoku puzzle is an expansion of the traditional Sudoku puzzle. Killer Sudoku uses the same grid and inherits the same constraints as a classic Sudoku puzzle, in that none of the values in any given row, column or box in the grid can be identical. However, in Killer Sudoku there are no starting values. Instead, the grid is covered in a set of cages. A cage is a group of adjacent cells in which there must be no identical values, and the values of the cells in the cage must sum to a given total for that cage.[3] An example Killer Sudoku puzzle is given below.

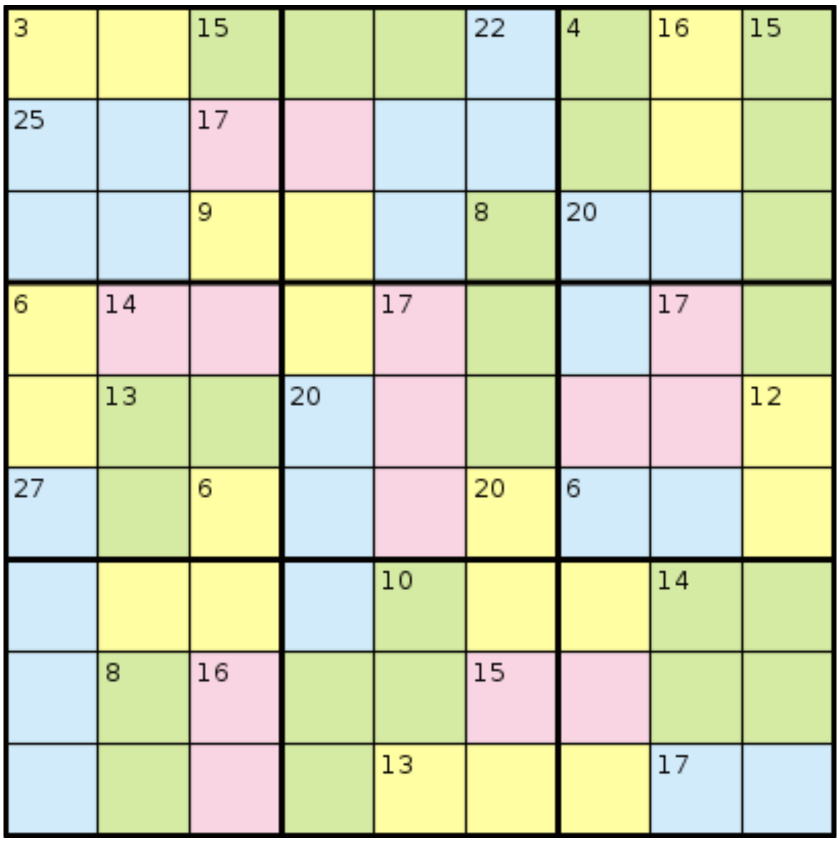


Figure 2.2.1 - Example of a Killer Sudoku puzzle[4]

In this example, the coloured regions of the grid represent different cages, with the number in the top corner of a cell representing the value the cells of that cage must sum to. These cages constrain the grid such that there is only one possible solution. The extra constraints and lack of starting values generally make Killer Sudoku more difficult to solve than classic Sudoku.

2.3 Common Solving Strategies

There is no single definitive strategy for solving a Killer Sudoku puzzle. However, there are a number of basic rules and techniques which can be used together to solve a puzzle. Again, there is no single definitive structure for the execution of these rules, and they are generally applied on an ad-hoc basis as the player sees fit for any given puzzle. Most of these basic rules work out which values **cannot** be in a cell, so the best approach is usually to start under the assumption that any value can be placed in any cell, and then apply rules to eliminate possibilities from cells or cages. As the possibilities become less and less, it becomes easier to assign values to cells, which in turn constrains more of the grid. When describing these rules, I will be assuming that the Killer Sudoku puzzle is using a 9 x 9 grid. I will also use the term "region" to describe a row, column or box.

- **Rule of 1**: Since no region or cage can contain duplicate values, once a value has been assigned to a cell, that value can be eliminated as a possible value from all the cells in the same region or cage as that cell.[5]
- **Rule of Necessity**: For any region, if all the values from 1 to 9 appear apart from one, that value must appear in the empty cell. Likewise for a cage, if only one cell in the cage is unassigned, only one value will sum with the rest to make the total, so that value must be assigned to the empty cell. For example, if a cage containing 3 cells has a sum of 15, and the values 4 and 6 have already been assigned, only the value 5 is both distinct from 4 and 6 and will sum with 4 and 6 to produce 15.[5]
- **Rule of 45**: This rule involves the comparison of regions with cages. A region must contain all the values from 1 to 9, meaning the sum for that region is 45. Therefore, if **S** is the sum of the totals of every cage **contained entirely** within the region in question, the remaining cells not covered by these cages must sum to **45-S**. For example, in figure 2.2.1 above, a cage of size 4 and total 15 and another cage of size 2 total 12 are entirely contained within the right-most column. Therefore, they represent a total sum of 27. The remaining 3 cells in that column which are in cages not entirely contained with the column must sum to 45-27, giving 18. Those 3 cells can now be treated as a new cage of size 3 and total 18.[5]
- **Rule of K**: This rule expands upon the Rule of 1. If a collection of *k* cells contain exactly *k* possible values, then these cells cannot appear in any region all of the cells are also in. Therefore, these values can be eliminated from the rest of the region. For example, in figure 2.2.1 above, there are 2 yellow cells in the bottom-left box. If both of these cells have possible values of 1 and 2, then we know that those values will definitely appear in those cells. Therefore, the values 1 and 2 can be eliminated from the other cells in the same box and row as the 2 cells.[5]
- **Sum Elimination**: This technique involves calculating all the possible combinations of values which can make up the total for a given cage. For example, a cage of size 3 and total 6 can only contain 1 possible combination of values, 1, 2 and 3. Therefore, the values from

4 to 9 can be eliminated from that cage. For all cages of size 2, at least 1 value can be eliminated by using this method.[5]

3. Specification and Design

3.1 Specification

For this project, I will create a software application which allows for the creation and resolution of Killer Sudoku puzzles. The application will be created using the Java programming language and has the following requirements:

- It must have an intuitive graphical user interface (GUI) with which the user can create a Killer Sudoku puzzle.
- The user must be able to save a puzzle to a file, and likewise be able to load a stored puzzle from a file into the application.
- The software should work for two different puzzle sizes: 4 x 4 puzzles; 9 x 9 puzzles.
- It must be possible for the GUI to be updated while the puzzle is being solved to show the progress being made. The user should be able to toggle this feature on or off.
- An object-oriented programming approach should be used, to make the application extensible and allow for the easy implementation of multiple solving components.

In addition to this, the solver component of the application has it's own requirements:

- Multiple puzzle-solving rules must be implemented and used in combination with each other to solve a Killer Sudoku puzzle.
- The solver must use a cache of sub-solutions to improve the overall speed and efficiency of it's algorithms.

3.2 Software Architecture

The architecture of the application will be similar to a model-view-controller (MVC) layout. MVC typically contains 3 separate components, which interact with each other in a cyclic process. The view component represents the graphical UI which the user sees. The user uses the controller component to interact with the application, and the actions taken by the controller manipulate the data in the model component. The model component contains the background data required by the application. Changes to the model are then sent to the view component, to update the GUI.[6] The diagram below illustrates this design.

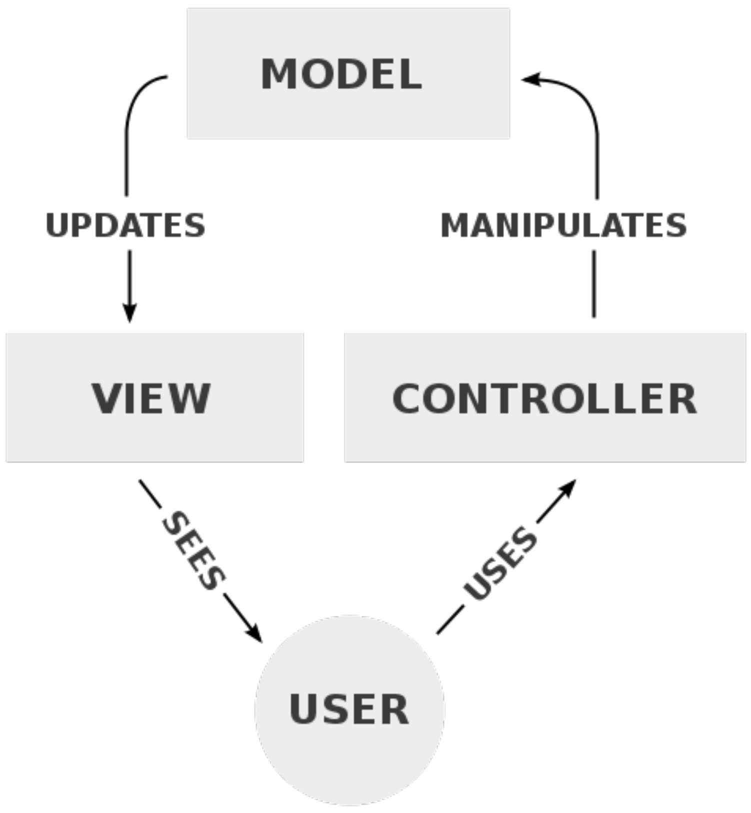


Figure 3.2.1 - Top-level structure of the model-view-controller software architecture[7]

However, the structure of the application I am creating will differ slightly from this architecture for a number of reasons. Firstly, user interaction is reduced, as they will only interact with the system to create/save/load a puzzle. The user has no further impact on the system once the solving begins. Secondly, as I will be using Java to develop this application, I intend to create data objects which inherit properties from Java Swing components, which will remove the definite distinction between model and view. Therefore, I intend to use a modified version of the MVC format containing just two components.

Due to my use of class inheritance in Java, the model and view components shall be combined. This component will be called the Puzzle, and will contain all the relevant data for the given instance of a Killer Sudoku puzzle as well as the graphical components displayed on screen. As almost all of the user's interactions with the system relate to the creation of a puzzle, I think it is also sensible to have the Puzzle component handle these interactions. The fact that the model, data and controller elements of the puzzle itself are so similar, means it is only logical to merge them into the same class. Attempting to separate them into a classic MVC architecture will only lead to a more convoluted design which does not provide much benefit. The design of the Puzzle component is discussed in more detail in section 3.3.

The second component will be the Solver. This component shall be initiated by the user, but after that point it will act independently and it will not be possible for the user to interfere with it's actions. The Solver will begin to solve the puzzle represented by the Puzzle component. It will make changes to the data in the Puzzle, and tell the Puzzle when to update the GUI. A diagram to illustrate this architecture is given below.

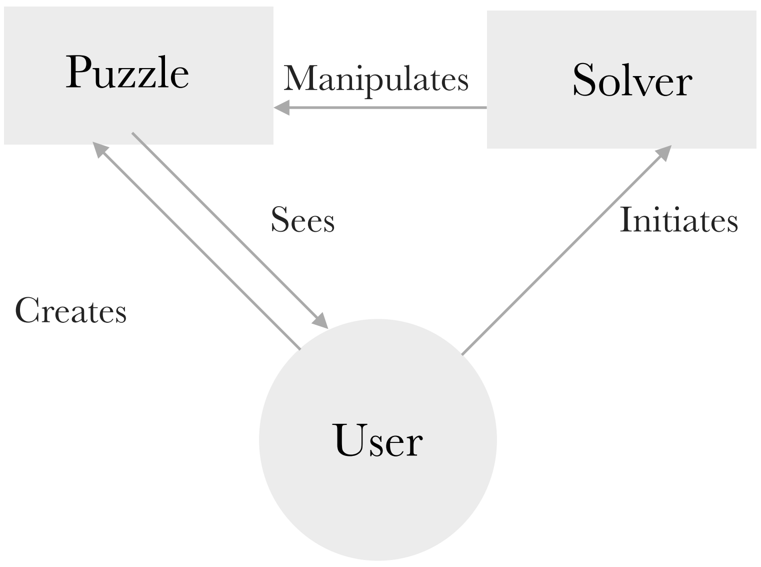


Figure 3.2.2 - Top-level design of the Killer Sudoku software I will create in this project

This separation between the Puzzle and Solver components will allow for the easy implementation of different solving strategies for the analysis part of this project. This architecture also creates a simple relationship between Puzzle and Solver, which should make the development of the Solver component simpler as all of the data will be provided by a single object. Isolating the Puzzle from the rest of the application should also facilitate easier saving/loading of puzzles to/from files. By having the Puzzle class and it's constituent elements implement the Serializable Java interface, I will be able to simply write the entire object to a data file in one command.

3.3 Digital Representation of a Puzzle

In order to design a digital representation of a Killer Sudoku puzzle, I have to consider the basic constituent elements of the puzzle, and how they interact with each other. The basic elements of the puzzle are the cells, cages, regions (rows, columns and boxes) and values.

- **Cells and values**: I shall create a Cell class, and an instance of this class will be created for each cell in the puzzle grid. Values are applied to single cells, so the assignment of values can be handled by the individual Cell objects. The solving techniques discussed in section 2.3 all work by eliminating values from a cell. Therefore, each Cell will contain a domain of possible values the cell can still take. When the Solver is initiated, values will be removed (and when necessary, added) to the domain of a Cell. When the domain contains only one value, that value will be assigned to the Cell as a fixed value. Each Cell will also contain column and row variables to signify it's position in the grid. Column 0 will be the left-most column, and row 0 the top-most row.

- **Cages and regions**: I shall create a Cage class, and an instance of this class will be created for each cage. Regions have all the same characteristics as cages, and so will be represented as Cages in the puzzle. For a 9 x 9 grid, a region can be defined as a Cage with a total of 45 (the sum of the values 1 to 9). Regions will not appear on the GUI as cages, but the box regions will have a thicker border. Each Cage object will contain a list of Cell objects which the Cage consists of. It will also contain the total the values of the Cells must sum to.

The Puzzle class shall be built upon these two classes, Cell and Cage. A 2-dimensional array of Cells will be used to represent all of the cells in the grid. They will be arranged in the array in the same way they are arranged in the puzzle grid; this should help to streamline the implementation for the Solver, as this layout will make it easier to identify which cells are affected by a change to another.

The Puzzle will also contain a list of Cages. When a new Puzzle object is created, the grid of Cells will be built and the rows, columns and boxes will be stored as Cages. In a 9 x 9 grid, there will be 27 of these regions, and 12 in a 4 x 4 grid. Once the Puzzle has been built by the user, each Cell will be in 4 different Cages: row; column; box; custom. The structure of the Cage object means the Cells in the Cage can be easily accessed, but there also needs to be a way of finding which Cages any given Cell is in. Therefore, I will store the Cages in a HashMap object in the Puzzle. Each Cage will have a corresponding Integer key, which can be used in the HashMap to find the correct Cage. Each Cell object will contain an array of 4 Integers, corresponding to the 4 Cages it is a part of. This will allow the Solver component to easily find the Cages any given Cell is in and make changes to the Cells in those Cages accordingly. The three classes discussed in this section are described in the UML-style class diagrams below (for a 9 x 9 puzzle).

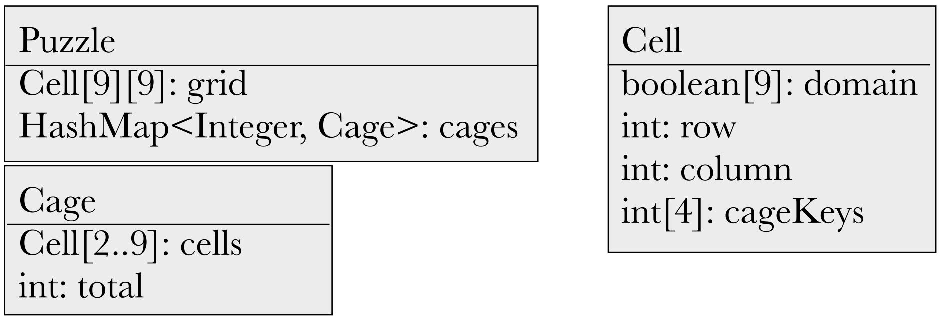


Figure 3.3.1 - UML-style class diagrams for the main classes in the Puzzle component

3.4 User Interface

In order for the user interface to be intuitive and easy to understand and use, it must closely resemble a Killer Sudoku puzzle. This is why I will be implementing the Puzzle class as a visual component of the application. The Puzzle and Cell classes will both extend the JPanel Java class, and the Cell panels will fill the Puzzle panel in a grid. This panel will make up most of the user interface, while there will also be a side panel containing controls to handle other features (saving/loading etc) and instructions on how to use the interface to create a new puzzle. The individual Cells will contain labels for each domain value. If the domain contains multiple values, they will appear in the Cell as small digits. If the domain contains only one value, that value will appear as a large digit. The hand-drawn diagram below shows the basic design for the interface (with a 9 x 9 puzzle grid).

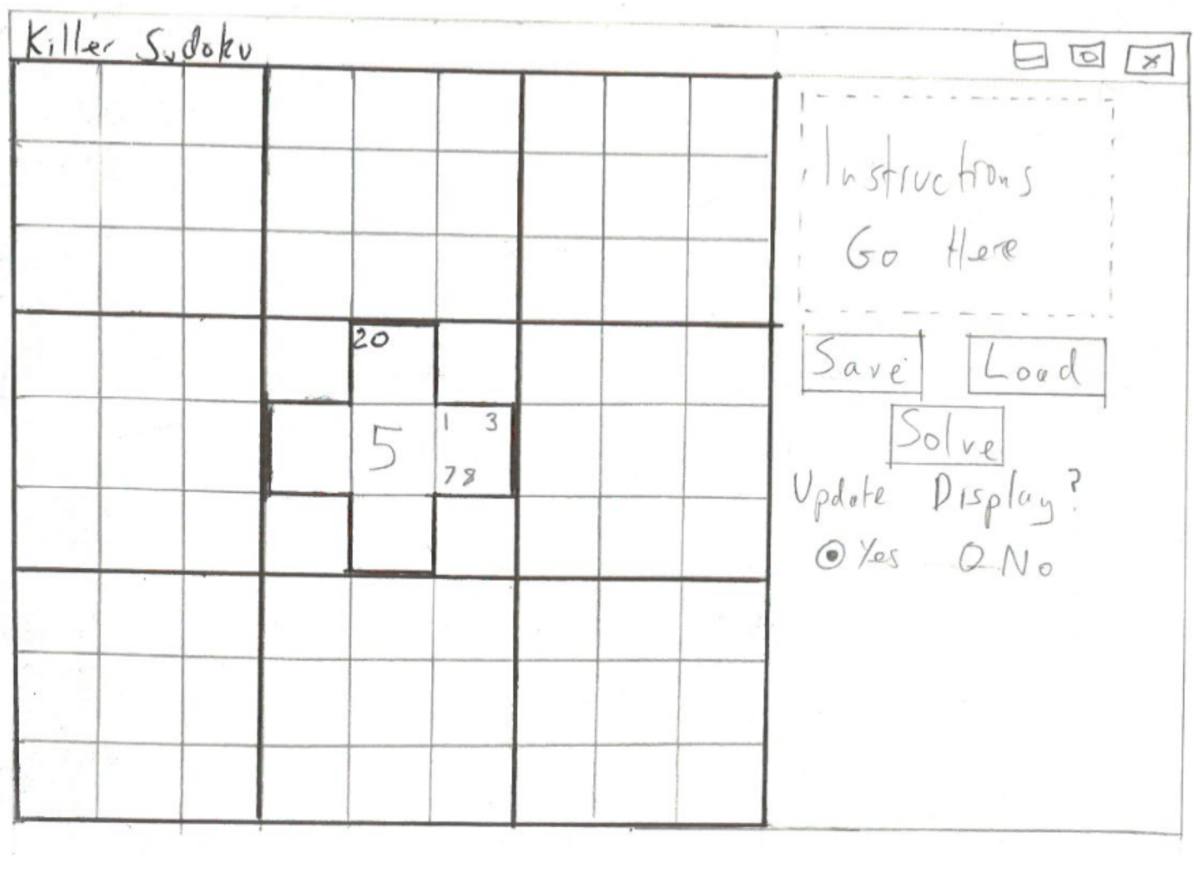


Figure 3.4.1 - Hand-drawn interface design for the Killer Sudoku application

The plus shaped collection of 5 cells in the centre of the grid shows how a cage will appear, with the total for the cage shown in the top-most cell. To add a cage to the current puzzle, the user simply clicks on the cells they wish to include. Selected cells will change colour, and clicking a selected cell will deselect it. Once the user has selected the cells they want, they will press the enter key. This will display an addition dialog box in which they will enter the total for that cage. Submitting that dialog will confirm the creation of the cage; the cells will be surrounded by a thick border and the total will be displayed in the top-most cell (if there are multiple top-most cells, the left-most of those cells will be used). The user will not be able to save or solve the current puzzle if any cell is not in a custom cage or if the totals of all the custom cages do not sum to the total of all the values in a solution (in a 9 x 9 grid, the values in each row/column should sum to 45. 45 x 9 = 405, so the cage totals should sum to 405). The interface will also support 4 x 4 puzzle grids, and which grid is loaded will be chosen using a command line argument when the application is launched.

3.5 Structure of the Solver

The Solver will be implemented with extensibility in mind. It must be designed in such as way as to allow for the creation of multiple different solving strategies, each of which utilises a set of rules in a different way. Therefore, I will utilise object-oriented programming techniques with a focus on inheritance. I will first create a Solver superclass. This class will contain the implementations of the different solving techniques and rules as functions. Then, the different solving approaches will be developed as subclasses of the Solver class, therefore inheriting these functions. To facilitate this, I will develop these subclasses as threads, meaning each subclass will have a run() function in which the order of execution of rules can be defined. If a subclass needs to use an altered implementation of a certain rule, it can be easily overridden by the subclass.

The Solver superclass will contain implementations of each of the rules described in section 2.3. Each rule will have it's own function, which can be called by the subclasses. In addition to these rules, I will also implement some others, such as checking the domains in a region to find any values which appear only once; in those cases the value in question must be assigned to the cell it was found in. I will also implement a guessing strategy by which the application can guess a value if the execution of the other rules is yielding no change to the puzzle. Likewise, I will also develop a way of backtracking on a guess if it is found to be incorrect.

The implementation of the sum elimination rule will utilise a memory cache. Combinations of values found for cages will be stored in a HashMap object, and will be retrieved when necessary. The algorithm for this rule will be recursive; the function will call itself by passing the current cage minus one cell as an argument. This will be done for every possible value a cell can take (typically 1 to 9), and the cage passed to the function call will have it's total adjusted to reflect that. The value being used will also be added to a closed list of values. This means that the recursive call is essentially finding the combinations the other cells in the cage can take given the first, removed cell carries a specific value. The following pseudocode describes the basic algorithm.

```
SumElimination(cage,closedList,memory) returns combinations:
if cage is in memory:
     return combinations from memory
else if cage.size == 2:
     for each combination for cage:
          if combination does not contain value in closedList:
               add combination to list
     return list
else:
     remove cell from cage
     for each possible value:
          cage.total -= current value
          add current value to closed list
          combos = SumElimination(cage,closedList,memory)
          add current value to each combination in combos
          cage.total += current value
          remove current value from closed list
     store all combinations in memory
     return all combinations
```
Figure 3.5.1 - Pseudocode to describe the design of the Sum Elimination algorithm

4. Implementation

4.1 Implementation of the Puzzle

As discussed in section 3.3, I have implemented a Puzzle class, which uses Cell and Cage objects to represent a Killer Sudoku puzzle. The Puzzle and Cage classes are defined in the Puzzle.java and Cage.java files respectively, but in a departure from the original design, I have implemented cells using two different classes: Cell and JCell, defined in the Cell.java and JCell.java files respectively. For each of these files, I will briefly describe the basic features of the implementation, and explore some of the more critical features and complex algorithms in more depth. A UML class diagram generated by my IDE has also been provided in the appendices, which helps to further describe these classes and their relationships. All source code for the application has also been included in an appendix, along with JavaDoc generated documentation for each class.

Cell.java

The Cell class contains all of the non-graphical data relating to a cell (row, column, domain etc), while the JCell class contains a Cell object, along with every graphical component required for the cell to be represented in the GUI. The reason for this design change is to create a class which can be more easily manipulated by the Solver, and which can also be used to store combinations of values in memory (see sections 4.2-4.3 for a more detailed explanation of this).

The Cell class contains the row and column of the cell, the current domain of the cell, and the integer keys of the Cages the cell is part of. The domain is stored as a boolean array. Each record refers to whether or not it's index is present in the domain. As Java does not support dynamic arrays using primitive data types, such as int, using this boolean array allows for easier manipulation of the domain by removing the need to resize the array every time a value is added or removed. However, when the domain is retrieved from the Cell using the getDomain() function, it is returned as an int array of all the values currently in the domain to make it easier for the calling class to evaluate. The domain can also be queried using the valueInDomain(int) function, which returns whether or not a given value is in the domain.

The keys for the Cages the Cell is in are stored in the "cages" array in the order row, column, box and custom. All the keys can be retrieved using the getCageKeys() function, and a cage key can be added using the setCage(int) function. A separate function exists to retrieve the key of the cell's custom cage, getCustomCageKey(). The total for the custom cage is also stored in the cageValue variable, and can be set and retrieved using the setCustomCageValue(int) and getCustomCageValue() functions respectively. This has the simple benefit of saving time when trying to find the custom cage total, as opposed to retrieving the key, then the Cage, then the total. If the Cell is not yet in custom cage, cageValue is -1 (an impossible cage total).

Many of the functions in the Cell class simply get or set variables, but there are some exceptions. First, I have overridden the equals(Object) function to compare the domain of the Cell with the Cell object passed in to it. I chose to override this function as opposed to writing a new one, so that this comparison is used whenever two Cells are compared, even by code in the core Java library (such as contains(Object) functions of ArrayList objects etc). I have also implemented a collection of functions which perform set operations on the cell domains. These operations are used in the Solver implementation, and the uses are described in more detail in section 4.3.

- **getUnion(Cell)**: Performs the **union** operation between this domain and the domain of the given cell. This creates and returns an int array containing any value which appears in either Cell's domain.
- **getIntersection(Cell)**: Performs the **intersection** operation between this domain and the domain of the given cell. This creates and returns an int array containing any value which appears in the domains of both the Cells.
- **getInverse()**: Performs the **complement** operation on this Cell's domain. This creates and returns an int array of any value which does not appear in this Cell's domain.
- **unionOfSet(Cell[], int)**: Performs the **union** operation on the domains of any number of Cells. This creates and returns an int array of any value which appears in the domain of any of the Cells in the given array.

The changeValue(int, boolean) method is also an important function, which adds or removes a value from the cell's domain depending on the arguments passed in. If the boolean variable is true, the given value is added to the domain, and the value is removed if the variable is false.

JCell.java

The JCell class is a subclass of the JPanel Java Swing class. The JCell constructor calls the JPanel constructor to create a small square panel on the GUI, and then fills the panel with the required components. The UI design in section 3.4 outlined the use of different sized digits to show the difference between a domain of multiple values, and a single assigned value. As such, different objects are used for these instances in the JCell class. A single assigned value (big digit) is shown using the bigNumLabel JLabel object, while a domain of multiple possible values is shown using the numGrid JPanel object, which is populated by the JLabels stored in the numLabels array. bigNumLabel and numGrid are added and removed from the JCell accordingly, using the add(JComponent) and remove(JComponent) functions inherited from the JPanel class. JLabels are not removed from the numGrid JPanel; instead, the visibility of the JLabels are toggled accordingly, using the setVisible(boolean) function.

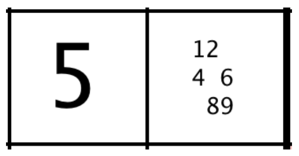


Figure 4.1.1 - Screen capture of two JCell objects in the puzzle grid of the GUI

Similar to the Cell class, most of the functions in the JCell class are for getting and setting variables. The entire Cell object contained in the JCell can be retrieved using the getRawCell() function. Many of the getters and setters from the Cell class are repeated in JCell, and simply call and return the output of their Cell counterpart. One such function is the changeValue(int, boolean) function, which performs the same operation as the corresponding function in the Cell class. It does not carry out any updates to the GUI relating to the modification of the domain, and this is handled instead by a separate function updateDisplayForValue(int, boolean). This way, the data can be altered without the UI being affected, which is necessary when the user does not want the puzzle to update while it is being solved.

The JCell constructor handles the initialisation and placement of all the relevant components within the panel. It is also responsible for drawing the cell's border which, along with the other cells, visualises the puzzle grid. The constructor uses the row and column variables passed in as arguments to determine whether or not the cell is on the border of a box region and if so, thickens the borders on the correct edge or edges. This means that the boxes appear on the puzzle grid as areas with a thicker border, as is traditional for Sudoku puzzles (see figure 2.1.1).

JCell also handles the drawing of cage borders. The Cage class determines which JCell borders the Cage border appears on, and passes this data to each JCell as boolean variables. These arguments (eight in total) are passed to the displayCageBorder(boolean, boolean, boolean, boolean, boolean, boolean, boolean, boolean) function. The eight arguments refer to whether a border should be drawn for each edge, and each corner of the cell. If the argument is true, the corresponding border is drawn. For edges, this is simply done by drawing a new border object parallel to the JCell's original border. However, for the corners this isn't possible, so a small JPanel object is made visible in the correct corner. This is what the topLeftCorner, topRightCorner, bottomLeftCorner and bottomRightCorner global JPanel variables are for. The displayCageValue(int) method also handles the displaying of the total for a Cage in the top left hand corner of the JCell. Again, the Cage class determines which Cell to use for this (the top-most cell).

Finally, instead of overriding the equals(Object) function as I did in the Cell class, I have implemented the distinct function hasSameDomain(JCell). While this function simply calls the Cell.equals(Object) function, I wanted to make sure this was not an override in the JCell class as these objects will populate the puzzle grid, and I will be using JCell.equals(Object) and other functions which use it to locate an exact cell in the grid. Therefore I needed to leave the JCell.equals(Object) function as it's default.

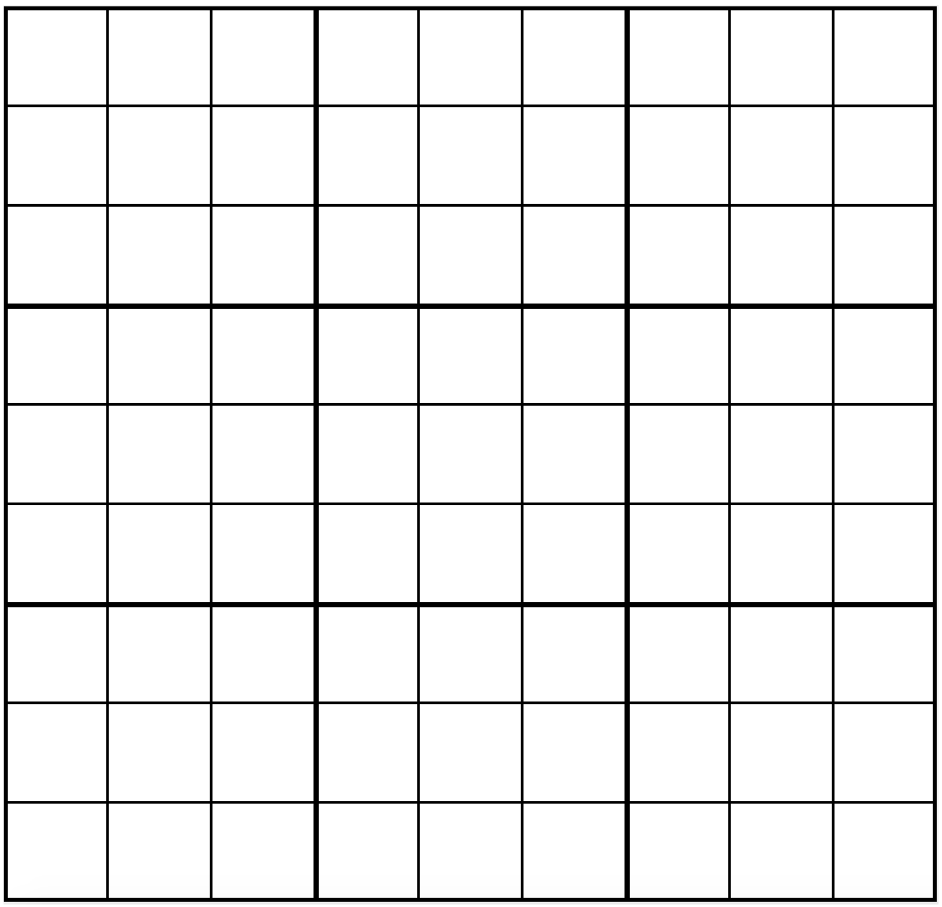


Figure 4.1.2 - Screen capture of the empty puzzle grid in the application GUI

Cage.java

Unlike JCell, the Cage class is not a graphical class and as such does not inherit any properties or functions from a Java Swing class. It contains three basic variables: cells, an ArrayList of JCell objects; total, an int containing the total value for this cage; type, a CageType enumerator object specifying the type of cage this object represents (row, column, box or custom).

Again, many of the functions are simply getters and setters for the object's attributes. I have also tried to mimic the functions which would be found in an array-type object in Java, by implementing add(JCell), remove(JCell), contains(JCell), size() and toArray() functions. As the names suggest these functions respectively add a JCell to the cage, remove a JCell from the cage, return a boolean indicating whether the given JCell is in this Cage, return the number of JCells in the cage and return a JCell array of all the cells in the Cage. The setTotal(int, int, boolean, boolean) method, which sets the total for the Cage, has the option of checking whether the given total is possible for the size of the Cage. The use of this option is defined by a boolean argument.

One function of particular interest is the checkOverlap(JCell) function. This function returns true if the current Cage contains every JCell in the Cage passed in as an argument, and false if not. This function is used by the Solver class, and is discussed in more detail in section 4.3.

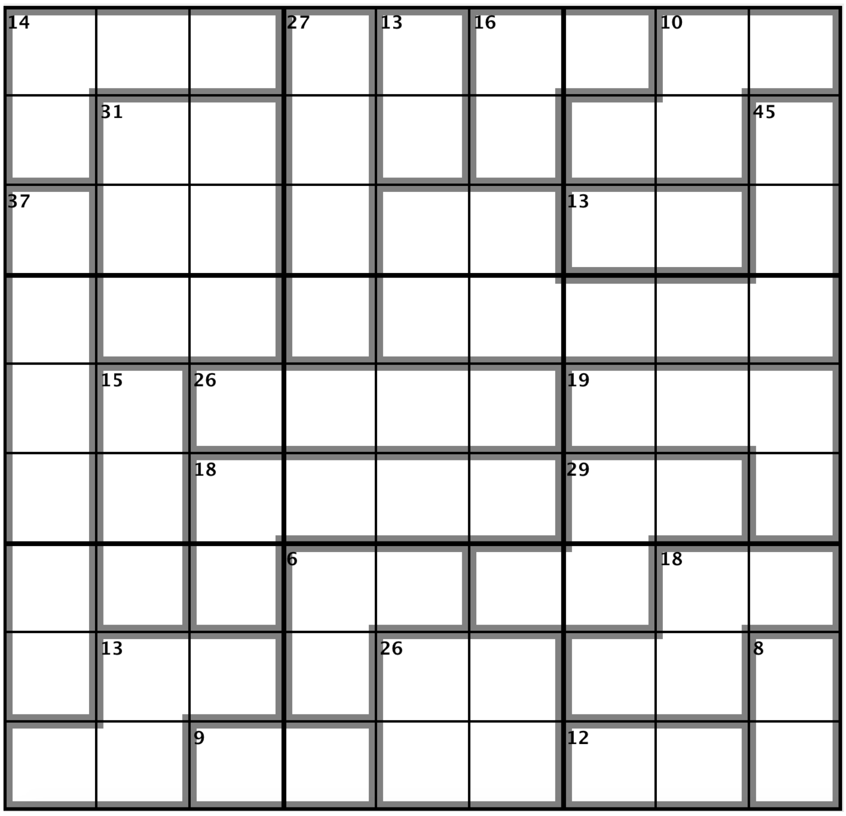


Figure 4.1.3 - Screen capture of a newly created Killer Sudoku puzzle in the GUI

Puzzle.java

The Puzzle class combines the three previously described classes to create a digital representation of a Killer Sudoku puzzle. This is the object which is saved and loaded by the user and the object which the Solver interacts with when attempting to solve a puzzle. Like JCell, Puzzle is a subclass of the JPanel Java Swing class, and inherits its properties and methods.

The Puzzle contains all of the cells and cages which make up the puzzle. The cells are represented by a 2-dimensional array of JCell objects, called “grid”. The cages are stored in a HashMap<Integer, Cage> object called “cages”. As previously discussed, each JCell object contains the Integer keys for the Cages it is a part of, and they can be used to find the correct Cages from the “cages” object. The row, column and box regions are also stored as Cages in this HashMap, as they share the same properties as a custom cage, and as such can be processed in the same way by the Solver. There is also an individual Cage object called newCage. This is the Cage which JCells are added to or removed from when the user selects or deselects them. When a Cage is finally created, the total entered by the user is added to newCage, which is then added to the cages HashMap. The Integer key for the Cage is then stored in the relevant JCells.

The Puzzle's constructor function creates a blank puzzle grid for the user to then fill in with cages. To begin it calls the constructor of it's superclass, JPanel, to create the panel region of the GUI the grid will inhabit. The panel's layout is then defined as a GridLayout, with the dimensions of the grid defined by the constructor's only argument, inSize. This refers to the size of the grid, either 4 or 9. This size has been previously determined by a command line variable when the application is launched ("-s" = 4 x 4, "-l" = 9 x 9). The constructor then sets about initialising the JCell's in the grid array (the size of which is also defined by inSize) and adding them to the panel. The GridLayout layout manager being used ensures that they are arranged in a grid format to create a Sudoku puzzle on screen.

As the JCells are being initialised, the constructor also builds Cage objects for each of the row, column and box regions of the grid. The initialisation process uses nested loops to move along each row of the grid. This means that every JCell created in one iteration of the internal loop are part of the same row Cage. Therefore, a new Cage can be added to the HashMap at the end of every internal iteration. Meanwhile, a Cage exists for every column, and every JCell is added to the correct column Cage during the iteration. At the end, the columns are added to the HashMap. As for boxes, multiple box Cages exist at the same time (again defined by the grid size). The algorithm uses the row and column values to determine which box the current JCell should be added to. They are then added to the HashMap after the last iteration of the outer loop. The following pseudocode describes how this process works for a grid size of 9.

```
create array of cages columns[9]
create array of cages boxes[3]
for integer i = values 1 to 9
    create empty row cage
    for integer j = values 1 to 9
        create cell
        add cell to row cage
        add cell to columns[j]
        add cell to boxes[j/3 + ((i/3)*3)]
    store row cage
    row cage = new empty row cage
store all column and box cages
```

Figure 4.1.4 - Pseudocode to describe the algorithm for creating Cage objects to represent regions in the puzzle grid

Many basic methods exist in the Puzzle class for the retrieval of data. getCell(int, int) returns the JCell with the given row and column reference, getCage(int) returns the Cage which corresponds to the given Integer key, and getCages() returns all of the Cages stored in the cages HashMap. Functions also exist to determine whether the Puzzle is completely is valid (validatePuzzle()) and whether a Puzzle has been solved (isSolved()). Like the Cell class, I have overridden the equals(Object) method to return true if the domains of the cells in the Puzzle match those of the Puzzle passed in. This is to help the Solver detect when it has stopped make eliminating values, and is discussed in more detail in section 4.2.

The Puzzle class also handles the user's interaction with the Puzzle, in terms of adding Cages to the Puzzle. To achieve this, the class implements the MouseListener and KeyListener Java interfaces. Mouse and key listeners are assigned to receive mouse and key events from every JCell in the puzzle grid, and the following methods have been implemented in the Puzzle class to handle these events:

- **mouseClicked(MouseEvent)**: Called when the user clicks on a cell. If the JCell is not part of the user's current selection of cells, then mark the JCell as selected using the JCell.select() method if the JCell in question is adjacent to another selected JCell in any direction. If the JCell is already part of the user's selection, deselect the JCell using the same method.
- **mouseEntered(MouseEvent)**: Called when the user's mouse pointer enters a cell. Change the background colour of the JCell in question to a light red to feedback to the user that this is the cell that will be selected if they click the mouse.
- **mouseExited(MouseEvent)**: Called when the user's mouse pointer leaves a cell. If the JCell being exited is selected, change the background colour of that JCell to a light blue to inform the user that the cell has been successfully selected. If the JCell is not selected, change the background colour to white to inform the user the JCell is not currently selected.
- **keyTyped(KeyEvent)**: Called when the user presses a keyboard key. If the user has pressed the enter key, and more than one JCell is selected, prompt the user to enter a total for the new cage they wish to create. If this value is valid for the size of the cage, then add this total to the new Cage, store it and display it on screen. If the value is invalid, inform the user and ask them to submit a new value. If an incorrect number of JCells are selected, inform the user of the error.

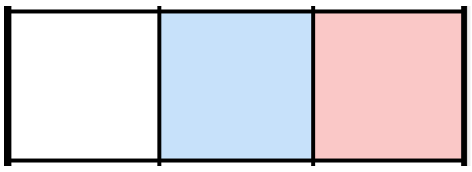


Figure 4.1.5 - Screen capture of three cells in the puzzle grid in the GUI, which are blank, selected and hovered over respectively

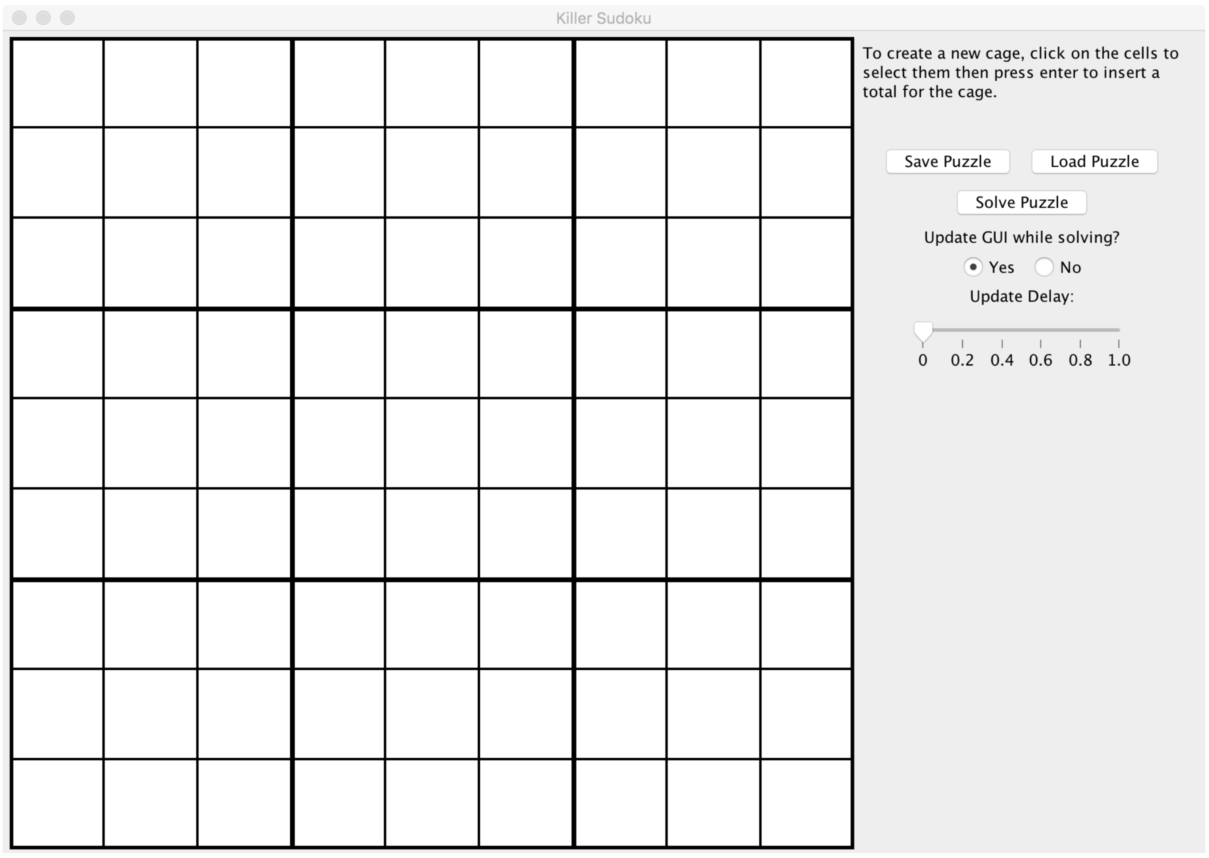
Saving and Loading

The saving and loading of a Puzzle is handled by code in the KillerSudoku.java file. This class is the main class for the GUI, and is responsible for constructing every component of the user interface. When the application is started, KillerSudoku calls it's createAndShowGUI(String) function, which creates a new JFrame object and populates it with every component required. This includes the save and load JButtons, and their function is defined when they are created here. These buttons simply prompt the user to select a location to create the file, or select the file to load. Once selected, the saving/loading is performed.

As previously mentioned, the Puzzle, JCell, Cell and Cage classes are all Serializable. This means that to save a file, I can simply use an ObjectOutputStream object to write the entire Puzzle object to a file, using the writeObject(Object) function provided by this class. Similarly, an ObjectInputStream can be used to read in a Puzzle object from a file and replace the current Puzzle in the KillerSudoku class with the new one. For both saving and loading, the Puzzle.validatePuzzle() method is called on either the Puzzle to be saved, or the Puzzle being loaded from a file. If the method returns false, the user is informed of an error, and the function does not complete (file is not saved or loaded into the GUI).

Other Components of the GUI

The JFrame created by the KillerSudoku class contains two JPanels: the Puzzle and another called the controlPanel. This second JPanel contains all of the controls needed to perform any function unrelated to the building of a puzzle. Along with the buttons for saving, loading and solving, there are two more features accessed in this part of the GUI.



To create a new cage, click on the cells to select them then press enter to insert a total for the cage.

Update GUI while solving?

[✓ Yes] No

Update Delay:

4.1.6 - Screen capture of the entire application GUI, containing an empty puzzle grid

Firstly, there are two JRadioButton controls, labeled “Yes” and “No”, beneath a Label which reads “Update GUI while solving?”. These radio buttons allow the user to freeze the Puzzle so that the GUI does not update while it is being solved. The Puzzle.setFrozen(boolean) method does this, by changing the Puzzle object’s boolean variable “frozen”. More detail on how this prevents GUI updates in given in section 4.2. By default the Puzzle is unfrozen, with the “Yes” radio button to selected, but selecting the “No” radio button will freeze the Puzzle an vice-versa.

There is also a JSlider slider bar object, with the values 0, 0.2, 0.4, 0.6, 0.8 and 1 as the increments, beneath a JLabel which reads “Update Delay:”. This control is used to set how long the application should wait in seconds after making a move when solving the puzzle. By default this is set to 0 seconds, but moving the slider will alter the delay. When the Solver is initiated by the user, the time they have selected in milliseconds is passed into the Solver’s constructor. How the Solver achieves this time delay is discussed in more depth in section 4.2.

4.2 Implementation of the Solver

To allow for the implementation of multiple solvers, I have created a Solver abstract superclass, Solver.java. This class contains the implementations of the different solving rules as separate functions (discussed in depth in section 4.3), and the global variables these methods use. Then, different solving strategies can be implemented in subclasses of Solver and if necessary, the rules can be overridden to provide alternative implementations. In this section, I will discuss the basic properties of the Solver.java class and why I have chosen to implement it in the way I have. Once again, a UML class diagram generated by my IDE has been provided in the appendices.

Using SwingWorker

In order to provide an efficient way of updating the GUI while solving a puzzle, I have implemented the Solver class as a subclass of the SwingWorker class. SwingWorker is an abstract class which uses threading to facilitate lengthy GUI-interactions while performing other processing tasks.[8] This is perfect for this application, as the Solver may need to make thousands of GUI updates before solving the puzzle. If the processing was to pause and wait every time this happened, the puzzle would take much longer to solve. Handing GUI updates to a separate thread is a more efficient method, as it allows the Solver to run uninterrupted (thus providing more accurate solving times for analysis), and if the machine used uses a capable multi-core processor, the use of multiple threads should not impact the performance of the application. Creating a separation between processing and updating also makes it easier to prevent updates if the user has elected to do so.

As an abstract class, SwingWorker defers the implementation of several methods to its subclass. The two most important of these are doInBackground() and process(List). These method represent the two different threads, performing calculation and GUI updates respectively. SwingWorker must also be defined with two type parameters. The first of these is the type the doInBackground() method returns when called, and the second is the object type the List input parameter contains in the process(List) method.[8] For this implementation, the type parameter are Boolean and UpdateObject respectively. UpdateObject is a class I have created, and will be described in more detail further on.

When the SwingWorker is created, it is then initiated using the execute() command. This calls the doInBackground() thread. In my implementation, this method begins to solve the puzzle. Whenever an update to the GUI has to be made, the doInBackground() method calls the publish(UpdateObject) function. The object passed in is added to a List in the update thread. This thread continually calls the process(List) function, which processes every object in this List and updates the GUI accordingly.

The UpdateObject class I have created contains three attributes: a JCell, cell, referring to the cell which must be updated; an int, value, referring to the value in the cell which must be updated; a boolean, display, referring to whether the value should be displayed (true) or removed (false). Whenever a value is changed by the doInBackground() thread, it calls changeValue(int, boolean) on the relevant JCell to make the change on a data level. Then, if the Puzzle is not frozen, it creates a new UpdateObject containing the data relating to the change it made and publishes it using the publish(UpdateObject) method. The process(List) method, for each item in the List argument, calls updateDisplayForValue(int, boolean) on the correct JCell to perform the GUI update.

Developing Multiple Solving Strategies

The different solving rules are implemented as functions in the Solver class. To analyse the different rules, I intend to execute different combinations and orders of rules, and this is the main way in which the subclasses of Solver will differ.

I have already explained that the doInBackground() method is used to define the function of the class extending SwingWorker, and that the implementation of this class is deferred to the SwingWorker's subclass. By defining Solver as an abstract class and not defining doInBackground() in Solver, I can defer the method's implementation again, so that it must now be defined by it's subclasses (i.e. the different solvers I will use in analysis). This is how I will be able to develop multiple solving strategies. When a new solver is created which extends the Solver superclass, a new doInBackground() function will need to be implemented. In this method, the execution of the solving rules will be defined by calling their functions (inherited from the superclass). If the inherited methods are correctly documented, this format should allow for the easy implementation of new solvers by new users, making the solver component of the application highly extensible.

By default, I have setup the application to use the LinearAvgLeast subclass as the main puzzle Solver.

Snapshotting the Puzzle

The solver must be able to detect when it is no longer making any progress towards solving the puzzle (i.e. no changes are being made to any cell's domain), so that it can either guess a value (section 4.4) or make some other corrective action. In order to do this, the Solver class provides methods for creating "snapshots" of the Puzzle object.

If the doInBackground() method of a solver is set up to iterate over a collection of rules, it will be stuck if after one iteration, the Puzzle has not changed. In order to detect this, the current Puzzle state needs to be compared to the state at the beginning of the iteration. The Solver class contains two methods to achieve this. The first, takeSnapshot(), serializes the Puzzle object as if it was being written to a file, but stores it in a ByteArrayOutputStream object, which is returned by the function. The second method, checkSnapshot(ByteArrayOutputStream), takes this object as an argument and restores it to a Puzzle object, as if being read from a file. It then compares this object to the current Puzzle, and returns true if they match and false if they don't.

I have used this process because copying the Puzzle object into a new variable would only create a memory reference to the original object, due to the way the Java Runtime Environment handles memory. By serialising the object, I can ensure that the copied Puzzle is not altered during processing, so the comparison of states will be accurate and correct.

The idea is that when implementing a new doInBackground() method, the takeSnapshot() method is called at the start of a new iteration, and the checkSnapshot(ByteArrayOutputStream) method is called at the end of that iteration. If checkSnapshot(ByteArrayOutputStream) returns true, no progress has been made during the iteration, so corrective action must be taken (e.g. a guess).

4.3 Implementation of the Solving Techniques

In the Solver class I have implemented the rules discussed in section 2.3 as individual methods. I have also implemented another addition basic rule, and a method which calls other rules in response to a value being removed from a domain. Each of these methods can still be overwritten by the subclasses of Solver to provide a different implementation.

I chose not to use the names of the rules given in section 2.3 when naming the different methods. This is because I felt that they did not suitably describe what the rule does, and I have instead tried to use names which do this, to allow for easier development of new solvers by new users.

I will describe the implementation of each rule in detail, and explain why I have developed them in the manner I have.

Cage Total Rule

The cageTotalRule(Cage) and ctrForCage(Cage, Cell) methods contains my implementation of **Sum Elimination**: This techniques involves calculating all the possible combinations of values which can make up the total for a given cage.[5] This is by far the most complex algorithm I have implemented for this project, and it makes extensive use of recursion and a memory cache.

This implementation uses two methods. The ctrForCage(Cage, Cell) method is the recursive function which returns the different combinations which the given Cage can have. The cageTotalRule(Cage) function calls this method, and then removes the values which are not in the combinations from the cells of the Cage. Before it calls ctrForCage(Cage, Cell) however, it first checks every cell in the given Cage. If a cell contains only one value in it's domain, then that JCell is removed from the Cage and it's value is subtracted from the Cage's total and added to a Cell object called closed. Once every cell has been evaluated, the Cage and closed Cell are passed in to the ctrForCage(Cage, Cell) function.

The ctrForCage(Cage, Cell) method will return an array of Cell objects, where the domain of each Cell refers to one possible combination of values that can fill the given Cage. The algorithm functions by working out what combination would be available in the rest of the Cage if one of the cells in the Cage took a specific value. This is done through a recursive algorithm, where a JCell is removed from the Cage before the Cage is passed into another ctrForCage(Cage, Cell) call. The value which is temporarily assigned to the removed Cage is added to the closed Cell, and removed once the recursive call returns. This recursive process continues until the Cage contains only two JCells, in which case the combinations can be found easily by iterating through the possible domain and removing each value from the Cage total to find it's compliment. When combinations are received from a call, the temporarily assigned value just added to closed, is added to each combination before being passed back up to the previous call. The only exception to this process is when the number of cells in the Cage is one less than the grid size. In this case, the value to be removed can be easily calculated using the Rule of 45 (for a 9 x 9 grid, the value to remove is S-45).

This is obviously a time-consuming process, especially for a 9 x 9 grid where there are a large number of Cages to process, and some of these Cages will be very large, leading to more recursive calls. Therefore, to improve performance, a memory cache of previously discovered solutions is used. Before ctrForCage(Cage, Cell) returns, the array of combinations it has created is stored in a global HashMap object called memory. The key for this HashMap is a Point object: an object containing two int values. In this context, the first value refers to the size of the Cage in terms of number of JCells, and the second value refers to the sum total for the Cage. The combinations are stored as the value. When ctrForCage(Cage, Cell) is called, previously stored combinations for a Cage with the same size and total are looked up in the memory variable. This prevents the algorithm from calculating combinations which have already been found, greatly improving performance and efficiency.

I have already discussed how the closed Cell object contains the values which have been temporarily selected for removed JCells in previous recursive calls. Any combination which contains a closed value, cannot be valid in the current context, but the combination is still valid for the current Cage so must appear when the combinations are stored in memory. To work around this difficulty, the Cell class contains a boolean variable called display. This variable determines whether a combination should appear in the Cage in the current context or not (true and false respectively). So, after receiving an array of combinations, the algorithm checks each Cell to see if it contains a closed value. If so, display is set to false, and set to true if otherwise. Every combination is then stored in memory. When the combinations are returned to cageTotalRule(Cage), it only considers those for which display is true. This means that only the valid combinations are used.

The cageTotalRule(Cage) method uses the Cell.unionOfSet(Cell[], int) method to combine the combinations returned by ctrForCage(Cage, Cell) into a new domain. It then inverses this domain, using the Cell.getInverse() method, and removes these values from every JCell in the Cage.

This algorithm is slow for the first few times it is called, as the number of solutions stored in memory is small. However, each sub-solution found by the ctrForCage(Cage, Cell) method is stored in memory, so a large cache is quickly built, increasing the efficiency of the algorithm with each new stored set of combinations. Repeating the method on a Cage yields benefits if none of the cells in the Cage have been assigned a value. If the same Cage is handed to ctrForCage(Cage, Cell), the previously found combinations will be retrieved, and no new values will be removed. However, if a cell has been given a final value, that JCell will be removed from the Cage, and an entirely new set of values will be retrieved for the rest of the Cage.

Constraining Cage Rule

The constrainingCageRule(Cage) and ccrForCage(Cage) methods contains my implementation of the **Rule of K**: If a collection of *k* cells contain exactly *k* possible values, then these cells cannot appear in any region all of the cells are also in. Therefore, these values can be eliminated from the rest of the region.[5]

Like the Cage Total Rule implementation, this rule uses two functions. The constrainingCageRule(Cage) method finds groups of cells within the given Cage which have identical domains. If the size of the domain is less than or equal to the number of cells which share it, that group is passed to ccrForCage(Cage) as a Cage object. That method then sets about removing the values of that domain from the cells of the related regions.

The constrainingCageRule(Cage) method works by evaluating custom Cages. It starts by looping through every JCell in the Cage. If the JCell's domain size is less than or equal to the number of cells in the Cage, the JCell is added to a new Cage. JCells with the same domains are added to the same Cage. Once this iteration is complete a new loop is started, this time looping through every new Cage created. If the Cage contains more than one JCell, and the JCell's domain size is less than or equal to the number of JCell's in the Cage, it is passed into the ccrForCage(Cage) function, as these values will not be able to appear in the other JCell's of any region the Cage is entirely contained in.

The ccrForCage(Cage) method starts by finding which regions are constrained by the given Cage. It does this by comparing the Cage keys stored in each JCell of the Cage. Note that there is no need to compare the actual Cage objects, as the keys can only refer to one specific Cage. If the same region key appears in every JCell in the Cage, then the Cage for that region is retrieved. There should be a maximum of two regions to constrain, as a Cage cannot constrain a row and column at the same time, but a box can be constrained alongside either a row or a column. Then for every region Cage retrieved, simply loop through the JCells of the Cage and remove the values from the given Cage's domain, making sure not to make any changes to the JCells in the given Cage.

This is a very important rule as it can help to eliminate a lot of values from a lot of cells in a few simple steps. Once a group of cells has constrained a region, there is no benefit to attempting to constrain with that group again. However, there is still a benefit from calling the algorithm on the same custom Cage multiple times, as changes to the domain of the other cells in the Cage may create another collection of cells which constrain a region.

Cage Difference Rule

The cageDifferenceRule(Cage) method contains my implementation of the **Rule of 45**: A region must contain all the values from 1 to 9, meaning the sum for that region is 45. Therefore, if S is the sum of the totals of every cage contained entirely within the region in question, the remaining cells not covered by these cages must sum to 45-S.[5]

My algorithm takes a non-custom Cage as input (custom cages cause the method to return null). The first task is to divide the cells of the Cage into two groups: cells from fully contained Cages, cells from partially contained Cages. The algorithm iterates through the JCells in the Cage, and uses the Cage.checkOverlap(Cage) method to evaluate whether the current JCell's custom Cage is entirely contained within the region passed into the function. If the method returns true (full overlap), the custom Cage is added to an ArrayList of Cages. The Cage's sum total is also added to a running total. This running total is the variable S referred to in the description of the Rule of 45. If the method returns false (partial overlap), the JCell is added to a new Cage, called "cells".

Once the iteration is complete, the sum total of the "cells" Cage is set as S-45. We now have a new Cage of values, with a sum total, which we can pass into the ctrForCage(Cage) method. This will return the different combinations which apply to this Cage, as an array of Cell objects. Combining these Cells and inverting the domains (Cell.unionOfSet(Cell[], int) and Cell.getInverse() respectively) gives the values which cannot appear in the "cells" Cage, and they can be removed from the domains of the Cage's JCells. The only exception to this is if the "cells" Cage only contains one JCell, in which the value the JCell contains must be S-45.

Once the correct values have been removed from the cells, we must evaluate whether the Rule of K applies. If the number of values left in the domains of the cells is the same as the number of cells there are in the "cells" Cage, then none of the values can appear in the other cells of that region. Calling constrainingCageRule(Cage), and passing in the "cells" Cage as an argument, carries out the Rule of K on those cells.

This is the only rule I have implemented which is guaranteed to only yield results once for each region. If it was called for a second time on the same region, it would identify the same cells and the same total and attempt to eliminate the same values from the cells domains. The method could be made more effective if I modified it to treat JCell's containing only one value as cells from entirely contained Cages (similar to cageTotalRule(Cage)). This way, the method would yield different results if a cell was assigned a value between calls.

Last Cell Rule

The lastCellRule(Cage) method contains my implementation of the **Rule of Necessity**: For any region if all the values from 1 to 9 appear apart from one, that value must appear in the empty cell. Likewise for a cage, if only one cell in the cage is unassigned, only one value will sum with the rest to make the total, so that value must be assigned to the empty cell.**[5]**

The function of this algorithm is fairly simple. Firstly, it iterates through the JCells in the given Cage to find the only JCell with multiple values in it's domain (i.e. no fixed value). While it does this, it also sums the values assigned to the other cells. Then, when the loop is complete it simply subtracts the sum from the Cage's total to get the value to be assigned to the empty JCell. After checking this value will not contradict another cell in the Cage, the correct values are added and removed from the last JCell to complete the Cage. The valueFoundInCageRule(JCell) method is also called, to make changes to the JCell's regions in response to the value assignment (see below).

Obviously this rule can only be applied once to any given Cage (given it completes successfully), but it is a very useful rule as it assigns a final value to a JCell. This can have a knock-on effect to the regions the JCell is in, and can initiate a chain of rule applications which help to assign values elsewhere in the grid.

Value Found In Cage Rule

The valueFoundInCageRule(JCell) method contains my implementation of the **Rule of 1**: Since no region or cage can contain duplicate values, once a value has been assigned to a cell, that value can be eliminated as a possible value from all the cells in the same region or cage as that cell.**[5]**

This algorithm is also very simple. The value which has been assigned to the given JCell is found. Then for each region and cage the JCell is in, that value is removed from the domains of it's JCells. If the assignment of this value to this JCell has lead to only one more empty JCell in the current Cage, lastCellRule(Cage) is also called.

Similarly to lastCellRule(Cage), this function can only be called once for any given JCell. However, it can remove values from the domains of cells in three different regions, so can be very effective. Like lastCellRule(Cage), this can initiate a chain of rule application which make good progress towards solving the puzzle. I suspect that during my analysis, I will discover that this rule is best applied in immediate response to a value being assigned as opposed to routinely applied to every JCell by iteration, because I believe creating the knock-on effect described will be the most effective way of solving a puzzle quickly.

Only Occurrence In Cage Rule

The onlyOccurenceInCageRule(Cage) method implements a rule based on the principle that for any region, if a value only appears in one domain in that region, that cell must contain that value.

For each possible domain value, the algorithm loops through every JCell in the given Cage. If the value only appears in the domain of one JCell, all other values are removed from that domain. This only works for region Cages, as it is based on the fact that a region must contain one of every domain value.

Like lastCellRule(Cage) and valueFoundInCageRule(JCell), because this rule assigned a final value to a JCell, it is useful for removing lots from other JCells as a result of the rule application chain it creates. However, unlike these other two rules, it is more difficult to apply in response to changes. It may be useful to apply this rule after a region is processed by another rule which is likely to have removed a lot of value from the region, such as constrainingCageRule(Cage) or cageDifferenceRule(Cage).

Remove Value

The removeValue(JCell, int, boolean) method is used to remove a value from a cell, and make the relevant changes to the related surrounding cells in response to that removal. Removing a value from a cell could have a number of effects: it may eliminate other values from the cells custom Cage because a certain combination of values is now impossible; it may leave only one value in the cell's domain meaning valueFoundInCageRule(JCell) can be applied; it may leave only one cell in a Cage empty meaning lastCellRule(Cage) can be applied. Using removeValue(JCell, int, boolean) to remove a domain value allows the solver to make any relevant changes as soon as the value is removed, and initiate a chain reaction of rules which can make good progress towards solving the problem.

After firstly removing the given value from the given JCell, the algorithm checks the JCell's domain size. If it is now just 1, valueFoundInCageRule(JCell) can be applied. This function will also call lastCellRule(Cage) if necessary. After this is done, it goes about removing values from the JCell's custom Cage which are now no longer valid.

Firstly, the algorithm checks the other JCell's in the custom Cage, to see if their domains contain the value removed from this JCell's domain. If any JCell does have this value in it's domain, no value need be removed, as any combination containing the value is still possible in the Cage. If no JCell's contain this value, the array of combinations for this Cage (stored as Cell objects) is retrieved directly from the memory HashMap. The next step is to identify the combinations which have been invalidated, but we must make sure not to remove a value from an invalid combination which is still also part of a valid combination. Therefore, a boolean array is created, with one record for every possible domain value (similar to a Cell object's domain array). This array is called remove, and every record is initialised to true, with the exception of the value removed from this JCell. A loop through every combination then begins, and every value which appears in a combination NOT containing the removed value has it's corresponding 'remove' record set to false. Once this loop is complete, every value for which ‘remove’ still reads true must be removed from the other JCell’s in the Cage. This is done, and valueFoundInCageRule(JCell) is called if necessary for each JCell.

Executing these rules in direct response to a domain change is far more efficient than executing the rules routinely as part of a main loop. If the latter was done, various changes could be easily missed. This method creates the chain reaction I have already described, and can help to assign values to cells much quicker.

4.4 Making Guesses

There will be times when the Solver is unable to make any further process towards solving a puzzle by applying the same rules routinely, especially for the harder puzzles. In these cases, a guess must be made, where the Solver selects a JCell and assigns it a value within it’s domain. The Solver then continues processing as usual, and if it finds that the puzzle cannot be solved with that guess, it simply reverts every change made since the guess was made, and removes the guessed value from the JCell’s domain before making another guess.

There are two parts to this process, making a guess and backtracking on an incorrect guess. The Solver superclass contains an implemented method for each of these components: makeGuess() and backtrack() respectively. The general principle is that the doInBackground() method of the subclass contains a loop, iterating through a series of rules in a routine fashion. At the end of each iteration, the algorithm checks to see if the Puzzle has been altered in the previous iteration (using the snapshotting feature described in section 4.2). If not, makeGuess() is called, and a guess is made in the Puzzle. From that point on, all moves made by the Solver are stored in a Stack object. If any rule later detects that the solution is incorrect, it can drop back to the doInBackground() method where backtrack() is called. This method then uses the Stack to revert every change made since the guess was made, including the guess itself.

makeGuess()

The makeGuess() method has three parts: choosing the JCell to make the guess in; choosing the value to guess in that JCell; making the guess. There are multiple different methods and heuristics which can be used to determine where to make a guess and what value to guess, and I intend to explore some of these in the analysis section of this project. However, in this section I shall describe the algorithm which has been implemented in the Solver superclass.

To select the JCell to make the guess in, the algorithm first looks for the Cage (custom or region) with the lowest average domain size, which is defined as the sum of all the JCell’s domain sizes divided by the number of JCells in the Cage. Within the selected Cage, the JCell with the smallest domain is then chosen as the target cell. This is based on the idea that if there are less possible values remaining in the cell, any value guessed has a greater chance of being correct than a guess in a cell with a larger domain. The JCell selected must have more than one value in it's domain.

Once the JCell is selected, a value is chosen from it's domain. In this algorithm, the Cages the target JCell is in are checked. For each value in the target JCell's domain, the number of times that value appears in the domains of the Cage's JCells is counted. The value with the smallest number of appearances is then chosen as the value to guess. This is because if the value has been removed from many of the cells in these Cages, it is more likely that the value should be in this cell.

Once the JCell and value have been chosen, all other values in the JCell's domain are removed using the removeValue(JCell, int, boolean). However, unlike other calls to this function, the boolean variable is given as true for the first call made, which starts the process of recording moves.

Recording Moves and Guesses

If a guess is proven to be false, then any move made since the guess was made is invalidated. In these instances, all of these moves need to be reversed, and the Solver needs to continue from the point before the incorrect guess was made. In order to achieve this, a record needs to be kept of every guess made, and every move made since the first guess was made.

The Solver uses two Stack object to record moves and guesses, called guessStack and guessesMade respectively. Both of these Stack objects store UpdateObject objects, the custom object I created to facilitate GUI updates from the SwingWorker. This object records the JCell and value of a change, and whether it was added or removed, which is everything needed to reverse the move. Calling the removeValue(JCell, int, boolean) function and setting the boolean argument to true causes the method to store the move being made in the guessesMade Stack. Then, whenever any move is made by any method in the Solver class, that move is stored in the guessStack Stack. The initial move is also stored here, so the same object exists in both Stacks. In the event of a false guess being discovered, the backtrack() method can then use these Stacks to reverse every invalided move.

Discovering an Incorrect Guess

The backtrack() method is called in response to a false guess being discovered. All of the rule functions contain checks when making moves, which ensure that the solution being created is still valid. These checks include making sure a JCell's domain isn't being completely emptied by a move, checking that identical values are not appearing in the same Cage, checking that the values in a completed Cage sum to the correct total, among others. In the event that one of these checks fails, the Solver immediately drops out of any methods until it returns to the doInBackground() method, where the backtrack() function is then initiated.

The Solver class contains a boolean variable called backtrack. By default it is set to false. When an inconsistency in the solution is detected, the guessesMade Stack is checked. If the Stack is not empty, then we know that the inconsistency must have been caused by the last guess made, so the method sets the backtrack boolean to true, and immediately returns null. Throughout the Solver class, the backtrack variable is checked directly after a call to another Solver function returns. If backtrack is true, then that function also returns null immediately. This process essentially halts the Solver to allow for the error to be quickly rectified. Eventually, the Solver will return to the doInBackground() method which will detect that backtrack is set to true, and as a result call the backtrack() method.

The ideal method of backtracking would have been to implement an entirely recursive algorithm, in which the program could automatically move back through the call stack to revert moves. However, the fact that the rules were implemented as different methods would have made this very difficult to implement. I had some difficultly ensuring this method was working, because there are multiple points in each rule method where an inconsistency can be detected, and making sure that backtrack was being set for every check took a lot of time.

backtrack()

The backtrack() method reverts to before the last guess was made by removing items from the guessStack Stack. It pops an item from the Stack (the most recent move coming first due to the "last in, first out" nature of a Stack) and for the JCell and given value it performs the opposite action to that stored in the UpdateObject, thus reversing the move. It continues to do this until it pops an item from the guessStack Stack which is identical to the top item in the guessesMade Stack. This means we have reached the point at which the guess was made. This move is reversed, and the move is removed from the guessesMade Stack.

The failure of the guess also proves that the value guessed cannot be in that JCell's domain, so it can be removed. However, removing the value could disprove other guesses made previously. Therefore, before the value is removed, backtrack() checks the domain size of the target JCell. If the JCell's domain only contains one value, then removing the value will create a false state, meaning the previous guess to this one is incorrect. In this instance, backtrack() calls itself. If not, removeValue(JCell, int, boolean) is called to remove the guessed value. It is not stored as a guess, but is still stored as a move if there are still previous guesses in the guessesMade Stack. Like all other function calls in the Solver, backtrack() checks the backtrack variable when the function call returns, and calls backtrack() if it is set to true. When backtrack() returns to the doInBackground() method, it resets the backtrack boolean to false, and continues to process as normal.

5. Testing

5.1 Approach

Before I began analysing my software, I had to ensure that it functioned correctly. Firstly, I needed to confirm that all the components of the user interface were functioning correctly: the puzzle builder; saving and loading of puzzles; puzzle updates during solving etc. Once this was done, I was able to test the function of the Solver, by ensuring that all of the implemented rules made the correct changes to the puzzle, ensuring it could correctly detect errors and inconsistencies within the solution, and ensuring that it could correctly solve puzzles of all difficulties.

To carry out these tests I used a combination of both white-box testing and black-box testing. Black-box testing (testing the functionality without assessing the internal code or logic) was used to check the basic function of the user interface, and review the output of the Solver. The white-box testing (testing by reviewing the internal operations of the system), was primarily used to ensure that the actual data was consistent with what was being shown on the UI, and that no incorrect calculations were being made by any component of the software.

5.2 Testing the User Interface

I identified five different areas of the user interface to test: cage construction, saving/loading, validation, updating while solving (including an added delay), and puzzle freezing. I shall describe what tests I performed to successfully evaluate these factors.

Cage Construction

Puzzle building is the main feature of the GUI, so it was important to ensure that it functioned correctly. Without the ability to correctly create new puzzles, I would not be able to solve them and therefore not be able to perform my analysis.

- **Cell selection**: I performed black-box tests to check the selection of cells by the user. Every cell I clicked on was successfully selected, and the background of said cells were coloured blue as intended. The reverse happened when deselecting a cell. Once a Cage had been created, I was unable to select any cells in the Cage.
- **Entering a Cage total**: I was also able to successfully create a Cage containing my selection of cells, and the enter the total for the Cage which was displayed in the correct part of the Cage. I created Cages of all possible sizes, and tried multiple shapes of Cage as well, and all proved successful.

Once I had confirmed these tests as successful, I white-box tested by debugging the application to check that what was displayed on-screen was a true reflection of the data stored. I was able to confirm that only the correct cells had been included in the new Cage, and that the total had been correctly stored.

I was also unable to select cells which did not share a border with an already selected cell (with the exception of the first cell selected). However, I was already aware of the fact that a similar type of validation was not implemented for deselecting a cell, and that it was possible to remove a cell in such a way as to create two or more individual collections of selected cells. When I pressed enter to create the Cage, the application did not prevent the selection, and instead showed two caged regions on the grid, with only one showing the entered value. Debugging the application as I had previously showed that all of these cells had been included in one Cage object with the total I had entered.

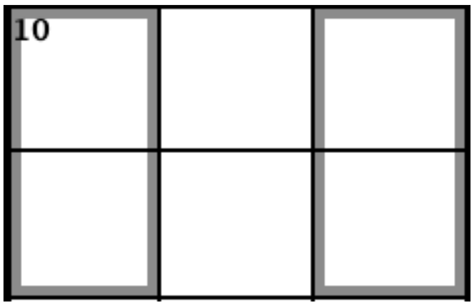


Figure 5.2.1 - Screen capture of an error in which a cage has been split before creation

Although this is clearly a bug, it does not affect my ability to perform analysis of the system as the puzzles will be entered by myself, so I can ensure that the Cages are correct. However, validation should be added in the future to eliminate this error.

Saving/Loading of Puzzles

To test this feature I created five different Killer Sudoku puzzles using the system, and saved them to individual files. I then loaded each file back into the program and checked them to confirm that they were the same puzzles. Each of the five puzzles was saved and loaded successfully with no change to the layout of the puzzle. Furthermore, I then performed a white-box test on each puzzle, in which I checked the data of five random custom Cages in each puzzle, to ensure that they contained the same cells and total visible on screen. Once again, all of these tests were passed successfully.

Validation

The user interface contains validation in two key places: cage construction; saving a puzzle.

- **Cage construction**: The software checks for valid cell selection (already tested) and valid totals when creating a Cage. For this, the software should only allow totals to be entered which are valid for that Cage size. If not, the user should be asked to enter a valid total. Each different Cage size has an upper and lower bound for totals, so for each size I performed five tests: total below lower bound; total on lower bound; total between lower and upper bounds; total on upper bound; total above upper bound. Totals below the lower bound or above the upper bound should be rejected, while all other totals should be accepted. Every test was passed successfully. For the instances where I tested a rejected total, I also white-box tested to ensure that only the correct total was stored, and not the initial incorrect total. This was the case.

- **Saving the puzzle**: Two validation checks are made when saving: every cell is in a custom Cage; the Cage totals sum to the sum of all values in a correct solution (405 for a 9 x 9 grid). Firstly, I checked that a puzzle with uncaged cells could not be saved which was the case. I then tested to see if a fully caged puzzle could be saved in three different instances (all using 9 x 9 grids): sum of Cages less than 405; sum of Cages more than 405; sum of Cages equal to 405. The first two of these tests should reject the save action, and last one should accept the save action and let the user proceed. These tests produced the expected results. I saw no need to perform white-box testing of this validation check, as the error is discovered before the user is prompted to select a save location, so it is not possible for the file to have been saved.

Updating While Solving and Puzzle Freezing

A simple black-box test was performed to evaluate the UI updates while solving a puzzle. I simply attempted to solve the same puzzle multiple times, setting the delay slider to a different increment each time. The different increments of time in seconds were 0, 0.2, 0.4, 0.6, 0.8 and 1. I was able to observe the expected difference in delay, and I also performed a white-box test whereby I measured the delay using system time printouts to the system console. The delay measured was slightly over the expected delay time (a matter of milliseconds, likely caused by the time needed to printout the data). I also performed another black-box test to evaluate the user's ability to prevent UI updates during solving. I froze the puzzle using the radio buttons and initiated the solver. No updates were made to the UI until the puzzle was solved, at which point the correct solution was displayed.

User Testing

After completing my own tests, I also carried out some user testing. The GUI was presented to a number of independent third-party users, who were asked to construct a Killer Sudoku puzzle and save it to a file. They were also asked to explore the interface to try to find any missing features or bugs. The consensus amongst the users was that the GUI was understandable, intuitive and easy to use. However, they did identify that there was no way of removing a Cage once it had been placed, and if a mistake was made while creating a puzzle, their only option was to restart the application. In response to this, I implemented a new feature in which the user can select Cages in the same way they select cells. When the user presses the backspace key on their keyboard, they are asked if they would like to remove the selected Cages. If they click yes, the Cages are removed from the puzzle.

5.3 Testing the Solver

The five areas of the Solver component I need to evaluate are: function of the individual rules; guessing; detecting errors in the solution; backtracking; overall solution to a puzzle. Testing these components involved more white-box testing than my testing of the user interface, as I needed to understand what was happening on the data level to guarantee that the code was functioning correctly.

Function of the Individual Rules

I primarily used white-box testing to analyse the individual solving rules. After analysing the output of a Solver using only that rule, I then used the debugging tools of my IDE to step through the rule and make sure it was processing the data correctly, and not just producing the correct results by chance in those instances. There were also some rules, Cage Total Rule for example, for which I was able to black-box test the output on different puzzles. In this instance, I created a puzzle which contained all possible 2-cell Cages and compared the results with a list of possible combinations for each Cage. After performing all of these tests I was able to conclude that all of the implementations of the rules functioned correctly and produced the correct results.

Guessing

To test the guessing implementation I had to check whether the Solver made a guess at the right time, and chose the correct JCell and value for the guess. Once again, I mainly used white-box testing to achieve this.

- **Electing to Guess**: I used breakpoints in the doInBackground() method to check that guesses were only being made when the other rules were failing to change the puzzle. This was the case.
- **Choosing the Guess**: This time I used breakpoints in the makeGuess() method to check that the correct JCell and correct value were being selected, in line with the method described in section 4.4. This was the case. I analysed the makeGuess() method for multiple puzzles.

Detecting Inconsistencies in the Solution

This testing involved modifying the Puzzle object slightly before performing the test. I created a puzzle for which I knew the solution, and placed values in it which I knew to be incorrect. These included values in the wrong cells, and values in Cages which summed to more than or less than the Cage's total.

- **Misplaced value**: I placed values into the puzzle which I knew were not correct for that cell. I then let the Solver run, and added a breakpoint to pause the application when the

backtrack boolean variable was set to true. By testing this using multiple misplaced values, I was able to verify that all of the error checks were able to detect the inconsistencies.
- **Cage not summing to total**: I tested three instances for this: values sum to less than total; values sum to more than total; values sum to exact total. Once again, I placed the same breakpoint to trigger when backtrack was set to false, and found that it was changed correctly when the sum of the values in a Cage did not equal that Cage's total.

While these tests prove that the checks I have implemented do work, they are not definitive proof that every inconsistency will be caught. There may still be instances where an inconsistency can be created and not found, but I have performed these tests on multiple puzzle containing multiple inconsistencies, and am yet to find any scenario where they are not detected, so I am happy to conclude that these tests were successful.

Backtracking

As I had already tested the error detection, the backtracking tests only involved ensuring backtracking is initiated once an error has been found, and ensuring that all the previous move are reversed. I also had to test what backtrack() does after reversing the moves, i.e. removing the guessed value or backtracking further.

- **Initiating backtrack()**: I used a simple white-box test where I placed a breakpoint in doInBackground() to ensure backtrack() was called when the backtrack variable was true. This was the case.
- **Reversing the correct moves**: This test involved stepping through the loop in the backtrack() method, to first make sure that every move was correctly reversed by checking the state of the relevant JCell after the reversal, and secondly checking that it correctly detects the match between the UpdateObjects in both Stacks when the loop should end. Both of these tests were passed.
- **Action after backtracking**: I stepped through the backtrack() method in two instances. The first, when the guessed value was the last value in the JCell's domain. In this case, the algorithm is meant call itself again to backtrack to the previous guess. I successfully observed this. The second instance was when there was more than one value in the JCell's domain, meaning the guessed value was simply removed from the domain. I also observed this activity successfully.

Successfully Solving a Puzzle

This was a black-box experiment, in which I simply let the application solve a number of Killer Sudoku puzzles ranging in difficulty from easy to extreme. Every puzzle I tried was solved successfully.

I am confident that the results of my tests prove that the application is fit for purpose, and meets all of the requirements set out in section 3 of this report. I also believe that it shows that this software is ready to be used in the analysis phase of the project.

6. Analysis and Findings

6.1 Description of Experiments

Now that I have successfully created a software solution capable of solving Killer Sudoku puzzles, I intend to carry out experiments to analyse and evaluate the different solving techniques.

Although I have implemented many different solving rules, I don't know the most effective way of applying these rules when solving a puzzle. I want to explore the difference between executing the rules in a linear order in which every rule is executed once in an iteration, and an order in which some rules are executed multiple times in a loop, after other rules have completed. Some rules may be best applied in response to changes made by others, so I want to test implementations in which certain rules are applied multiple times throughout an iteration. Conversely, I also want to test implementations where rules are never applied in response to changes in another, such as when a value is assigned to a cell. I am not concerned by which rules should be executed before others, as I don't believe that the differences between different specific orders will be significant, given the current set of available rules. However, this line of enquiry could form the basis of further experiments in the future.

Despite not being originally planned in the specification of this project, I also want to examine how guessing effects the application. Making an incorrect guess can leave the Solver attempting to complete an incorrect solution for some time, which obviously has a large effect on performance. Being able to make a correct guess as often as possible is critical to the success of the software, especially in cases where either the puzzle is extremely difficult or there are only a small number of rules being applied. Therefore, I also intend to experiment with different guessing heuristics and methods, to try and find the method of guessing which is most successful, and evaluate how reliant the Solver is on its ability to make correct guesses.

In total I have devised three different ordering techniques I wish to test, three different methods to selecting the cell to guess in, and two different methods of selecting the value to guess (these are all described in section 6.3-6.4). I will cross-pollinate these techniques to create 18 different Solvers, which I will then test against a test suite of puzzles, ranging in difficulty from easy to extremely difficult. I will record the times taken for each Solver to solve the puzzle in the test suite, and analyse the results to evaluate which methods are the most effective.

6.2 Test Data and Format of Experiment

I have compiled a test suite of 100 individual Killer Sudoku puzzles, all stored in files readable by my software. The puzzles are taken from the www.killersudokuonline.com website, and cover five difficulty levels: easier; easy; moderate; hard; extreme.[9] There are 20 puzzles from each difficulty level in the test suite. The solutions for these puzzles (and by extension, the layouts of the puzzles) are provided in the appendices.

For the purposes of the experiment, I have also saved the solved versions of these puzzle to files readable by my software. I have then created a small command line application, called BatchSudoku. I created 18 different versions of this file (BatchSudoku1.java through to BatchSudoku18.java), which each use a different Solver to iterate over the 100 puzzles in the test suite, applying the given Solver to each one and timing how long it takes to solve them. When a puzzle is solved, it compares it's solution to the solution file to check it has been solved correctly. One example of this application, BatchSudoku1.java has been included with the source code in the appendices of this report.

To carry out the experiments, I will execute each BatchSudoku program on a different machine in the Cardiff Computer Science School's Linux laboratory, by creating ssh connections to each machine from my personal computer. I have chosen this approach because attempting to test all 18 Solvers on the same machine will take an extremely long amount of time, so the only feasible way to perform an experiment as comprehensive as I have planned is to find a way of testing the Solvers concurrently. I realise that this approach has an element of unreliability as I cannot fully guarantee that the environment is the same for every Solver. However, I have considered this and concluded that the hardware of the machines being used is not only identical, but also very powerful, so any other activity on the machine is very unlikely to effect the results of my tests. Therefore, I am happy that the environments of the machines during testing will be essentially identical, and the risk of producing unreliable data with this approach is small enough that the data can still be trusted.

The hardware (identical on all 18 Linux machines) I will use to perform these experiments carry the following specifications:

- Processor: Intel Core i7-4790 8-core CPU @ 3.60 GHz per core.
- Random Access Memory: 15.5 GiB
- Operating System: Linux Ubuntu 14.04

The names of the 18 machines I will use during this experiment have been provided in the appendices.

6.3 Analysis of Orders of Techniques

There are three different patterns of rule ordering I wish to explore:

- **Linear**: This pattern contains a simple routine in which four rules are executed one after the other in each iteration of the main loop. The only exception is when one of the rules calls another in response to a change. The order of rules is as follows: cageTotalRule(Cage); cageDifferenceRule(Cage); constrainingCageRule(Cage); onlyOccurenceInCageRule(Cage). In one iteration, the first rule is called on every Cage, then the second and so on. The lastCellRule(Cage) and valueFoundInCageRule(JCell) methods are still called by removeValue(JCell, int, boolean), so are not included in the loop. I will not be testing different orders of this format (executing a different rule first etc), as I am more concerned with exploring the differences between types of ordering and how that effects efficiency. I don't believe an experiment into different linear orders would provide as much insight as the experiments I am planning to carry out. The names of the Solvers which use this ordering format start with "**Linear**".

- **Repeated Application of Rule in Loop**: This pattern is a departure from the linear format. In this implementation, one rule is always executed after the completion of another, in an attempt to respond to changes made by the initial execution of the rule. The order of the rules is the same, except that instead of executing the onlyOccurenceInCageRule(Cage) as the last rule of an iteration, it is executed whenever another rule call returns. Again, I will not be testing different examples of this format, although I believe that if these experiments prove successful and this orientation shows different results from the others, further tests could be performed in which different rules are repeated in this way. For example, the constrainingCageRule(Cage) method could be used in place of the onlyOccurenceInCageRule(Cage) method. However, some methods such as cageDifferenceRule(Cage), could not be used in this way, as they are only effective once. The names of the Solvers which use this ordering format start with "**OnlyOcc**".
- **No Response to Values Being Removed**: This pattern can be seen as an extension of the linear pattern. This time, the removeValue(JCell, int, boolean) method just removes the given value from the given JCell, and does not call any other methods or make any other changes to any other JCell's. The valueFoundInCageRule(JCell) and lastCellRule(Cage) methods which are normally called by removeValue(JCell, int, boolean), are instead added to the end of the linear sequence of rules, in that order. Again, I do not believe that trying different sequences of rules in this way will yield much change in efficiency. However, implementing all six rules in the "repeated application" pattern described above may prove a useful experiment, trying the use the two addition rules in this pattern as the repeated rule in that design. However, for this experiment I will only the using one implementation of this pattern. The names of the Solvers which use this ordering format start with "**Remove**".

6.4 Analysis of Guessing

There are two different parts of the guessing algorithm I want to test: selecting the cell to make the guess in; selecting the value is guess in that cell. For cell selection, there are three methods I want to test:

- **Smallest Average Domain**: Select the cell with the smallest domain from the Cage with the smallest average domain (combined domain size / cage size). This is the method of selection which is implemented into the Solver superclass. The names of the Solvers which use this selection method contain "**Avg**".

- **Smallest Cage**: Select the cell with the smallest domain which appears in one of the smallest sized unfilled Cages. The names of the Solvers which use this selection method contain "**SCage**".
- **Smallest Domain**: Select the cell with the smallest domain across the entire puzzle grid. In the event that two cells have the same domain size, the cell from the smaller custom Cage is selected. The names of the Solvers which use this selection method contain "**SDom**".

The "smallest cage" and "smallest domain" methods provide two very different approaches to selecting a cell, while the "smallest average domain" method combines the two. It is important to select a cell which does not have too many possible values, but will also have a big effect on the surrounding area. The "smallest domain" method covers the first requirement, and the "smallest cage" method covers the second, as an assignment of a value in a smaller cage is more likely to lead to the rest of the cage being completed. This test will hopefully show which of these requirements is more important, or whether a combination of the two can be realised through use of the average heuristic I have created.

I also intend to test two different methods of selecting the value to guess once a cell has been selected. These methods involve selecting:

- **Least Constrained Value**: The value which appears the least number of times in the target cell's cages and regions. This is the default implementation in the Solver superclass. The names of the Solvers which use this selection method end with "**Least**".
- **Most Constrained Value**: The value which appears the most number of times in the target cell's cages and regions. The names of the Solvers which use this selection method end with "**Most**".

The first method seems the most logical, as the value which has been eliminated more times from the surrounding cages and regions is more likely to be the value that cell must take. However, using the second method would create the most changes to the puzzle, and possibly lead to more values being assigned elsewhere in the puzzle. This could lead to the puzzle being solved more quickly, or lead to an incorrect guess being discovered more quickly, so less time is spent by the program attempting to solve the puzzle with incorrect data. This experiment should prove which of these techniques is the most suitable.

6.5 Findings

To analyse my test results, I calculated average solve times for each difficulty of puzzle, within each class. I have compared the different solving techniques, cell guessing techniques and value guessing techniques, by then averaging the calculated averages for all the classes using a specific technique. For example, for the value guessing results, I have created an average of all nine classes using least constraining value, and another for all nine classes using most constraining value. This cross-pollination of classes, means that the data being analysed for any given technique, is taken from solvers using each of the other types of techniques (e.g. data for one ordering techniques is made from results of that technique combined with every guessing technique). All of the timing data returned as part of this analysis has been provided in an appendix.

Ordering

The graph below shows how the three different ordering techniques I developed performed in my experiment.

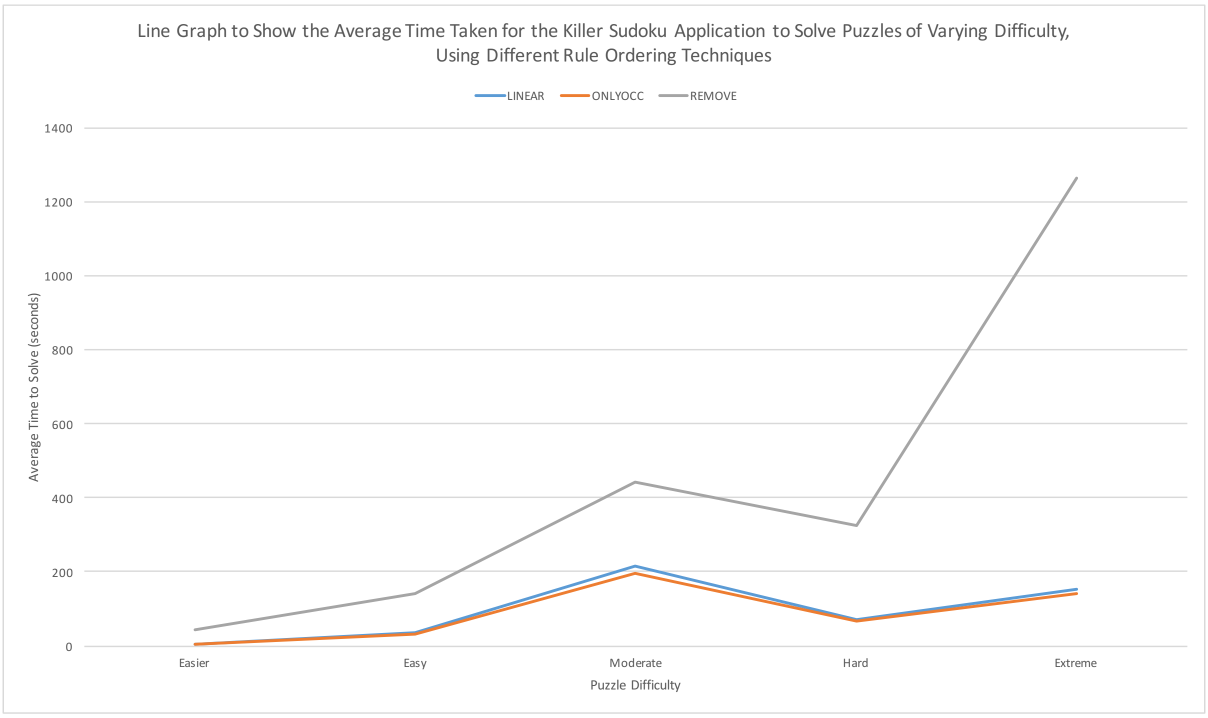


Figure 6.5.1 - Line graph to show the average time taken for the Killer Sudoku application to solve puzzles of varying difficulty, using different rule ordering techniques

The first observation to make is that the Linear and OnlyOcc (onlyOccurenceInCageRule(Cage) applied after every other rule is applied) techniques exhibit almost identical performance, with OnlyOcc performing slightly faster for moderate and extreme puzzles. The onlyOccurenceInCageRule(Cage) method only affects the puzzle if a value is found only once in a Cage, so the number of useful calls is likely to quite small. Therefore, it seems logical that executing this rule more often would have very little effect on performance. Any change which is made by a call in the OnlyOcc setup would eventually be made at the end of the loop in the Linear setup. Any benefit gained from making the changes earlier is likely offset by the processing time needed for all the extra function calls which do not make any changes (onlyOccurenceInCageRule(Cage) is called three times more often in

OnlyOcc than in Linear). However, this benefit is likely the reason for OnlyOcc performing slightly better than Linear for some difficulties.

Both of these techniques are also very efficient for all difficulties of puzzles, with the highest average being 214 seconds for moderate puzzles. These timings show that these formats are clearly very effective, and perhaps a further investigation into specific linear orders or different rules being repeated could be carried out to find an even more efficient solving strategy.

The other observation to make is the significant difference between these two ordering techniques and the Remove technique (no automatic response to a value being removed). This strategy is significantly less efficient for all difficulties of puzzle, and the difference in average solving times between Remove and the other techniques increases as the puzzle difficulty increases. The average solving time for extreme puzzles using this technique is 1264 seconds (approximately 21 minutes), over 1000 seconds faster than the equivalent for Linear and OnlyOcc strategies. I suspect that there are a number of contributing factors. The default implementation of the removeValue(JCell, int, boolean) function responds to a removal by taking values out of a cell's Cage, and calling the valueFoundInCageRule(JCell) and lastCellRule(Cage) methods. However, it only does this when necessary by making simple checks, so no time is wasted trying to make unneeded changes. This means that the Remove strategy has no natural advantage in terms of efficiency. Similarly, the valueFoundInCageRule(JCell) and lastCellRule(Cage) methods are called for every JCell and Cage respectively from the doInBackground() method. Calling these functions on objects which will not be affected by it may also negatively impact solving time.

I have also previously discussed the "chain reaction" that the default removeValue(JCell, int, boolean) method creates, whereby other methods are called and values are removed or assigned in response to just one change. The Remove ordering strategy does not benefit from this effect, meaning cells are not assigned values as quickly. Assigning values helps to constrain the domains of other cells, which in turn helps to assign more values, so assigning values as quickly as possible is obviously going to have a good effect on efficiency. It also effects guessing, as false guesses are detected by contradiction with assigned values. If false guesses are detected later, it means more time has been wasted on attempting to solve the puzzle with incorrect values, again negatively affecting efficiency.

In summary, the results clearly show that the Linear and OnlyOcc ordering techniques are the most efficient, and that the Remove ordering technique is very inefficient and should not be used. The Linear and OnlyOcc techniques should be used as the basis for any Solver, and these techniques should be tested in more depth to determine which specific orientation of rules is the most effective.

Guessing - Cell Selection

The graph below shows how the three different cell selection techniques I developed for guessing performed in my experiment.

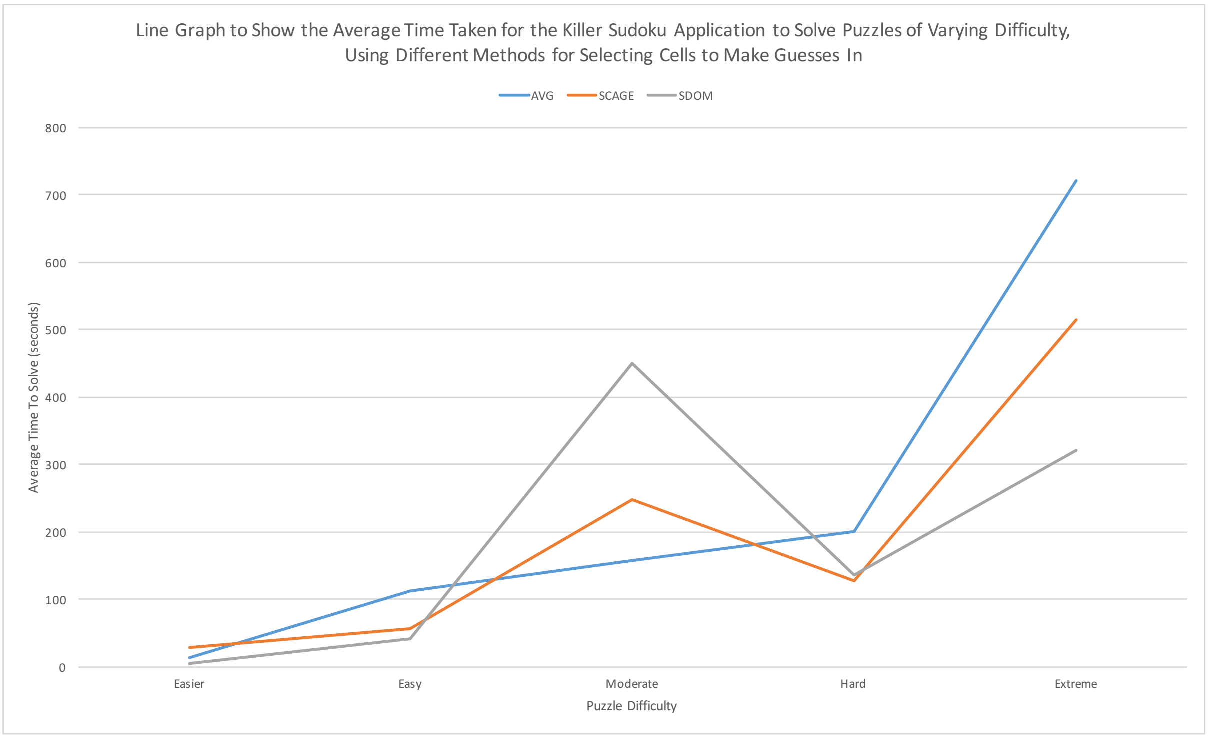


Figure 6.5.2 - Line graph to show the average time taken for the Killer Sudoku application to solve puzzles of varying difficulty, using different methods for selecting cells to make guesses in

These results are difficult to analyse, as there is no consistent pattern to show which strategy is the most effective. Every strategy is the most effective option for at least one different difficulty of puzzle, and the least effective for another. The "Smallest Domain" strategy seems to be the most effective, as it produced the best average solve time for three different puzzle difficulties (easier, easy and extreme), and was the least efficient solver for moderate puzzles only. Likewise, this would suggest that the "Average Domain Size" strategy is the least effective, as it produced the least average solve time for three different puzzles difficulties (easy, hard, extreme), and was the most efficient for just one difficulty (moderate).

It seems logical that the "Smallest Domain" strategy should provide more efficiency. If a guess is made in a cell with a full domain of nine values, there is an 11.1r0% chance that the guess will be correct. The percentage chance of a guess being correct increases as the domain size of the cell decreases, leaving a 50% chance of a guess being correct in a cell with a domain size of two. Using a cell with a smaller domain also helps to detect incorrect guesses made previously. If every guess in a cell is found to be false, the previous guess must also be false, and backtrack() is initiated. If the domain size is smaller, this will happen more quickly, which in turn helps to reduce the time needed to solve the puzzle.

However, in some situations the "Smallest Cage" and "Average Domain Size" provide a more efficient solution. It is not clear what causes this trend, but it would suggest that none of strategies tested have any significant effect on the Solver's ability to make correct guesses. I think that a more detailed experiment should be carried out to focus on identifying a more suitable technique. A larger data set should be used in this experiment to reduce the effect anomalous data has on the results, and other selection techniques should be developed to analyse what factors help to select an accurate and correct guess. I also think that other metrics should be recorded and analysed. Most of the work towards solving a puzzle is made by the solving rules, so these inconclusive results may be down to the fact that guessing does not have as significant an effect. If the guess success rate were to be recorded, in which correct and incorrect guesses are tallied to give a percentage success rate, an experiment would be able to focus more closely on how well each Solver is guessing, and the most suitable strategy could be identified.

Guessing - Value Selection

The graph below shows how the two different value selection techniques I developed for guessing performed in my experiment.

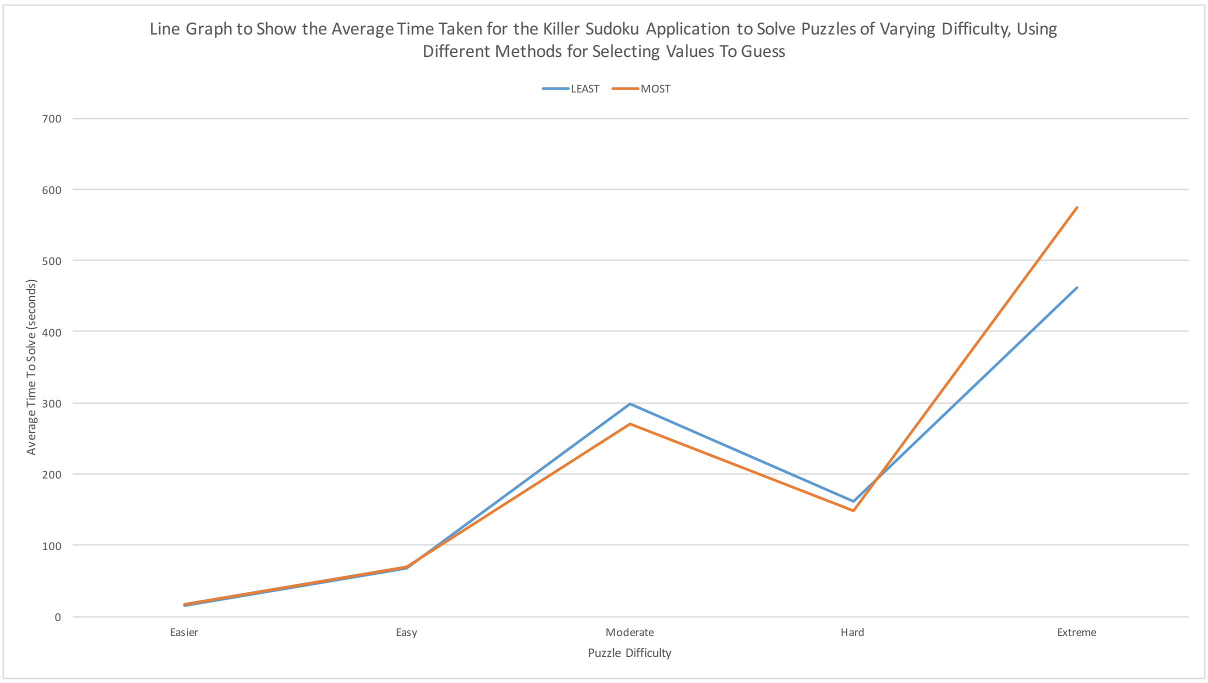


Figure 6.5.3 - Line graph to show the average time taken for the Killer Sudoku application to solve puzzles of varying difficulty, using different methods of selecting values to guess

The results show that the two techniques exhibit very similar performance, with the "Most Constrained Value" being more efficient for moderate and hard puzzles, "Least Constrained Value" being more efficient for extreme puzzles, and the two strategies producing almost identical results for easier and easy puzzles.

Unless a cell is selected with a domain size of two, the value selected will always be more likely to be incorrect than correct, regardless of how the value is selected. This may explain why this heuristic has not greatly affected the solving times, as any guess made is always more likely to be incorrect. The results would seem to suggest that using the "Most

Constrained Value" technique yields better results. This makes sense, because if the guess is more likely to be incorrect, it may be more prudent to attempt to prove it is incorrect so it can be removed from the domain more quickly, thus increasing the chance of the next guess being correct, which was one of the ideas behind using the "Most Constrained Value" strategy.

Again, a more detailed investigation into guessing values may help to shed more light on how to make a successful guess, with larger data sets, different techniques and new metrics being used.

Conclusion

I believe that the experiment I have performed has provided some interesting insights into how the efficiency of the application can be affected by the design of the Solver. The most efficient ordering techniques have been identified, while another has been proven to be unsuitable. I believe that this discovery can provide the basis for further testing and experimentation, in which the Linear and OnlyOcc techniques are analysed in more depth to determine how to maximise their efficiency.

The experiments into guessing have been less conclusive, with no single technique being identified as consistently superior. However, the "Smallest Domain" method of cell selection did perform well, and I think that a more comprehensive experiment would be able to identify this technique as the most efficient. The fact that different guessing techniques performed at different levels for different difficulties of puzzle leads me to believe that an experiment in which the layout of the puzzle is considered may provide further insight. The irregular results of this test may be down to the specific layouts of the individual puzzles, so more insight may be gained from comparing performance for different puzzle patterns.

As I have already mentioned, analysing different metrics could also help to improve experiments involving guessing. The number of guesses needed when solving a puzzle varies from puzzle to puzzle, so the time taken to solve a puzzle may not be the best metric to record and the "guess success rate" metric could be more useful for analysis. This heuristic represents the primary concern when considering guessing, and should always be maximised, so this is likely to be a better indicator of performance when comparing guessing techniques.

7. Future Work

Having built the software in this project from scratch myself, and then having also performed a broad-ranging experiment to analyse the performance of the software in different circumstances, I believe that there is adequate scope for expansion of the software. During the project I have had many ideas for how the user interface could be enhanced to provide greater control over the Solver, and how the Solver itself could be improved, through development of new solving rules. I have also thought of ways to expand the analysis of the application to consider specific orders of the execution the rules, and also comparison of different approaches and implementations of some features.

7.1 Additional Solving Techniques

The number of solving rules I was able to implement during development of the software was limited by the timespan of the project. I am aware of a number of rules which I was unable to implement into the Solver class, some of which are specific to Killer Sudoku but also other which are taken from classic Sudoku strategies, and which could be implemented as extensions of current rules.[10]

For example, a basic Sudoku strategy is the concept of hidden and naked candidates. These terms refer to pairs, triples or quadruples of values in cell domains which, in the correct circumstances, constrain those cell's regions. This essential idea, mainly focussing on naked candidates, has been implemented in the constrainingCageRule(Cage) method of the Solver class, but by exploring the idea of naked candidates further, this rule can be expanded to accommodate collections of values which appear only a certain number of times in a region, but not as the only values in their cell's domains.[11][12]

As well as expanding current rules, there are new rules which can be developed. One such rule I have considered involves combining regions to make new discoveries. For example, combining two rows and treating them as one cage (with a total of 90 in a 9 x 9 puzzle) could help to identify collections of cells which can be treated as a pseudo-cage, either through application of the cageDifferenceRule(Cage) method or otherwise.

I knew that no matter how many rules I implemented, there would still be many more I was unable to include. This is why I designed the Solver component of the application with extensibility in mind, to ensure that the implementation of addition rules in the future would be simple and easy to understand for any developer.

7.2 Alternative Approach to Guessing

When I began to design the guessing feature of the Solver component, I devised two potential methods of implementation. The first design was the one eventually implemented, where a guess is selected and made, and can then be reverted if it is discovered to be incorrect. The alternative design however, involved using parallelism to attempt multiple different guesses at the same time. In this algorithm, a cell is selected in which to make a guess, just as the current implementation does. However, instead of choosing which guess to make, this implementation would instead have created a new thread for each possible guess which can be made in that cell. The threads then proceed to attempt to solve then puzzle with that guess in place. A thread is terminated as soon as the guess in that thread is found to be incorrect, until just one thread remains which must contain the correct guess. The Solver would also terminate in the event of one thread producing a valid solution.

I opted not to develop this design, because I was concerned about how successive guesses would lead to an extremely large number of threads. This is to say that if the threads need to make another guess, these threads will produce child threads, causing the number of threads to increase exponentially. This could cause the application to run out of memory, and is therefore a very risky design. However, with more time to develop the application I believe I could find a way to safely implement this algorithm and, if developed intelligently and efficiently, I think it could greatly improve the performance of the Solver component.

The implementation of alternative guessing methods would open up a new line of potential analysis. A similar experiment to the one performed as part of this project could be carried out to compare the two guessing implementations and determine which it the most efficient. A third implementation, containing a hybrid of both techniques in which the original method of choosing one selection is used if multiple threads already exist (to prevent creating too many threads), could also be analysed as part of this experiment.

7.3 Defining the Solver

I have designed and implemented the Solver component of the software to be extensible, allowing new Solvers to be easily created and implemented. However, while building a new Solver would be easy for a technically astute individual, it would not be easy for a first-time user, or somebody without programming experience. Therefore, I think the development of a feature whereby a user can specify the solving format using the GUI would be beneficial. I envisage this feature as a separate settings window, in which the user can drag and drop different rules into a panel, and reorder them as they see fit to determine how the rules are executed in one iteration of the Solver. The different rules would carry descriptions on the GUI, to educate the user on how they contribute to the solving of the puzzle. The user would also be able to save and load these configurations in a similar fashion to the way puzzles are currently saved and loaded.

The addition of this feature would allow anybody to design a new Solver, which could be very useful if further experiments and tests were to be carried out. It means that the application could be given to experienced Killer Sudoku players (such as those who engage in competitive Sudoku) without programming skills, who could provide further insight into how best to solve puzzles, or what strategies would be best suited to specific puzzle layouts.

7.4 Further Analysis

In section 6.5 I have already discussed potential areas for further experimentation. I believe that in response to my findings in this project, new tests could be performed to analyse how the performance of the Linear and OnlyOcc ordering techniques can be optimised by finding the best specific order of rules, or the best rule to repeat throughout an iteration. The addition of new rules would also create scope for additional tests to evaluate the effectiveness of the new rules. More experiments could also be performed to further clarify which guessing strategies are the most successful. In particular, and given that the analysis provided proof that executing rules in response to other changes was effective, I would like to implement Solvers in which there are more rules executed in direct response to other changes being made (in a fashion different from the OnlyOcc implementation in which rules are repeated regardless of changes). Some of the rules implemented, and rules which could be implemented in the future, may function more efficiently in response to other rules being executed, much like the valueFoundInCageRule(JCell) and lastCellRule(Cage) methods do, and experiments could help to identify these rules and in what context they are most effective.

If further experiments were to be performed, I would also use a different testing environment. While I am confident that the use of multiple machines with identical hardware will not have significantly affected the test results, performing all of the tests on the same machine would have been a more reliable model, and I would have carried out the tests in this way if I had had the time. In future tests, I would run all experiments on the same machine, and I would record additional statistics and metrics alongside solving time, such as total number of moves made, guess success rate, and time spent backtracking. I would also use a significantly larger data set, and I would also include puzzles of even greater difficulty than the ones used in my experiment. These additions would not only allow me to perform more in-depth analysis, but the extra statistics would help me to identify the primary cause of any trend in the data. This would help me to understand exactly why certain strategies are more or less successful than others.

8. Conclusions

This project had three main aims:

- Build a computer application with which a user could create a Killer Sudoku puzzle
- Implement a second component of this application, which would be able to solve a Killer Sudoku puzzle.
- Analyse the performance of the solving component of the application, to determine which order of execution of rules is the most effective.

The first aim has been achieved, as I believe I have built an intuitive and easy-to-use GUI in which a puzzle can be created, saved and loaded. I have tested the application myself, and found that almost all features work without error. I have also carried out user testing with independent third-party users, who were satisfied with the usability of the GUI. Every requirement set in the specification has been met, and this evidence leads me to conclude that this component of the software matches the specification and is fit for purpose.

I also believe that the second aim has been achieved. Tests have proven that the Solver component of the software is capable of solving Killer Sudoku puzzles of varying difficulty, and in an experiment of 100 puzzles, no attempts at solving the puzzle were unsuccessful. The same tests also proved that the Solver can solve the same puzzles using different configurations and rule ordering strategies, meaning that the goal of allowing multiple solvers to be developed has also been achieved. However, I do believe that the current implementation is perhaps too reliant on it's ability to guess values when stuck. While it would appear that the Solver can successfully solve some easy puzzles without needing to make a guess, it is almost always necessary for more difficult puzzles. The only way to alleviate this would be to reduce the likelihood of the Solver getting stuck, which the implementation of more solving rules would achieve. I therefore conclude that while the application is capable of solving a wide range of puzzles, the implementation of additional solving rules would make it more complete.

While some of the experiments performed on the software provided useful insight into the most effective solving strategies, others were inconclusive. The primary goal of the analysis was to identify the most effective way of ordering the rules, which I believe I have done. However, I still believe that there are plenty of areas which future analysis could explore, and given more time I would have liked to have performed these tests as part of this project. Although it was not specified in the initial project goals, I also decided it was important to analyse the guessing function of the Solver. These tests were largely inconclusive, but I do believe that their results have helped to identify areas for further investigation, and have suggested that the format of the guessing implementation can have an effect on the Solver's ability to solve puzzles efficiently. Overall I think that the tests carried out as part of this experiment were successful, as they helped to identify effective strategies for solving Killer Sudoku puzzles.

In summary, I am very satisfied with the end product of this project. I have been able to successfully create a user-friendly GUI for puzzle building, and an extensible and functioning software component for solving Killer Sudoku puzzles. I have also performed analysis of the Solver component, which has provided some useful insight into how best to structure the Solver component.

9. Reflections on Learning

This has been the largest project I have ever attempted on my own, and as such I have learned many valuable lessons during the process. Firstly, I have learned the importance of planning. Given the scale of the project, I have realised that had I not started by creating a formal plan, outlining the different phases of the project and the steps I would take to complete each phase, I would probably have worked at a much slower rate and the end product would have been much worse. When I have worked on large projects in the past, I have either been working with a group of people who have managed the project plan themselves and given me instructions and guidance on what to do next, or I have taken on a project which has already been started and the next steps are clear. Starting a large project from scratch was a new experience for me, and as such I have learned a lot from the process of breaking the project down into individual parts and planning how to achieve each goal within the given timeframe.

Likewise, I have also learned a lot about how to go about starting a new project. At various points during the development of my software, I felt like I didn't know where to begin. The end product of the next development phase was clear to me, but I had no idea how to go about achieving it. For example, once I had created the GUI and implemented all of it's features, the next phase of the project was to begin development of the solver. I knew what rules were going to be implemented and what the structure of the Solver would be, but I didn't know which part to take on first, or in what order I should implement the constituent parts. In the end, I just picked a solving rule and began to implement it. As I wrote the code, it became obvious what should be written next, and I formed a general idea of the approach I should take to complete that phase of the project. I think this has taught me that, while it is important to develop a high-level plan of a project, thinking too much about how to approach the finer details can waste time, and it is often better to simply start working on something, and allow the rest of the plan to form around that work.

The experiments I carried out to analyse the Solver taught me a lot about how to plan an experiment. Before this project, I had analysed software in a similar way, but never on the same scale as this investigation. I think that as a result of this, I didn't fully consider how large and time-consuming a full investigation would be. My approach was to develop the software, and then perform tests once it was complete, but when I came to the testing, I realised how long the tests were going to take. I didn't want to downsize the experiments, so I was forced to use multiple machines for the tests, which produced slightly less reliable results. In retrospect, I now realise that I should have been considering the analysis more during the implementation phase, and I should not have treated them as two distinct phases of the project. The format of the analysis is obviously effected by the implementation, and as such plans for experiments and tests should be created and altered as the software is being developed.

Finally, the fact that I have been able to successfully create a piece of software which can solve Killer Sudoku puzzles has made me realise what I am able to achieve with my skills.

From the outset, I was prepared for the eventuality that the software would not work. I was very unsure as to whether I could create an application capable of solving Killer Sudoku puzzles before the deadline. I expected that even if I got the software working, there would still be bugs, and that it still wouldn't be able to solve difficult puzzles. However, I have been able to create an application which is fully-functioning, and can successfully solve puzzles of any difficulty. This has helped me come to realise how proficient I am with the skill I posses, and what I am capable of achieving if I approach a problem in the correct way.

10. Appendices

All of the appendices for this report can be found in a .zip archive file titled Appendices.zip. The list below contains every appendix included, and the folder or file they can be found in within this archive file.

A. Compiled version of the Killer Sudoku application (including all 18 solvers) - KillerSudoku folder

B. Source code for the Killer Sudoku application - SourceCode folder
    - B.i) All classes which form the Puzzle component
    - B.ii) All classes which form the Solver component
    - B.iii) All 18 individual solvers created for the analysis phase
    - B.iv) The BatchSudoku1 class used during the analysis phase

C. JavaDoc generated documentation for all main classes of the software - JavaDoc folder

D. UML Class Diagrams
    - D.i) Class diagram describing the structure of the Puzzle component - UMLPuzzle.jpg
    - D.ii) Class diagram describing the structure of the Solver component - UMLSolver.jpg

E. List of machines used during the experiment carried out on the Solver component - machines.txt

F. Full listing of times returned during the experiment carried out on the Solver component - timings.xlsx

G. 100 puzzle test suite (including solutions, represented as .gif images) used during the experiment carried out on the Solver component - Puzzles folder

11. References

[1] Wikipedia. Sudoku. Available: https://en.wikipedia.org/wiki/Sudoku. Last accessed 3rd May 2016.

[2] Wikipedia. A typical Sudoku Puzzle. Available: https://en.wikipedia.org/wiki/Sudoku#/media/File:Sudoku-by-L2G-20050714.svg. Last accessed 3rd May 2016.

[3] Killer Sudoku Online. Killer Sudoku Rules. Available: http://www.killersudokuonline.com/rules.html. Last accessed 3rd May 2016.

[4] Wikipedia. Example of a Killer Sudoku problem. Available: https://en.wikipedia.org/wiki/Killer_sudoku#/media/File:Killersudoku_color.svg. Last accessed 3rd May 2016.

[5] Killer Sudoku Online. Killer Sudoku Solving Strategies. Available: http://www.killersudokuonline.com/tips.html. Last accessed 3rd May 2016.

[6] Wikipedia. Model-view-controller. Available: https://en.wikipedia.org/wiki/Model–view–controller. Last accessed 3rd May 2016.

[7] Wikipedia. A typical collaboration of the MVC components. Available: https://en.wikipedia.org/wiki/Model–view–controller#/media/File:MVC-Process.svg. Last accessed 3rd May 2016.

[8] Java™ Platform, Standard Edition 7 API Specification. Class SwingWorker<T,V>. Available: https://docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html. Last accessed 3rd May 2016.

[9] Killer Sudoku Online. Archive by Difficulty. Available: http://www.killersudokuonline.com/archives.html#diff. Last accessed 3rd May 2016.

[10] SudokuWiki. Strategy Families. Available: http://www.sudokuwiki.org/Strategy_Families. Last accessed 3rd May 2016.

[11] SudokuWiki. Hidden Candidates. Available: http://www.sudokuwiki.org/Hidden_Candidates. Last accessed 3rd May 2016.

[12] SudokuWiki. Naked Candidates. Available: http://www.sudokuwiki.org/Naked_Candidates. Last accessed 3rd May 2016.