CS21120: Data Structures and Algorithm Analysis

Assignment 1 – Sudoku

Richard Addicott (ria4@aber.ac.uk)

Student Number: 120046698

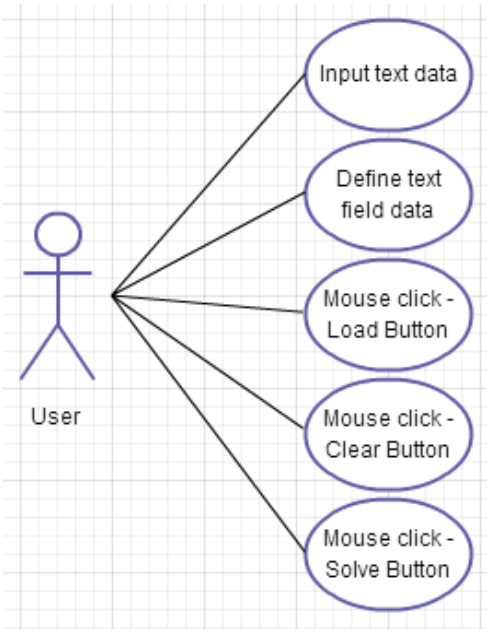
Contents

Analysis 2
Use Case Diagram 2
Design 2
Sudoku.java 2
SudokuDisplay.java 2
SudokuModel.java 3
SudokuCanvas.java 3
Class Diagrams 6
Flow Charts 7
Standard Program Flowchart 7
Solving Algorithm Flowchart 7
Testing 8
Test Table 8
Test Table Evidence 9
JUnit Testing 13
Evaluation 14

Analysis

The aim of this assignment was to design an algorithm to solve Sudoku puzzles. The overview of the task at hand, was to develop an algorithm to load in Sudoku data files that were provided as part of the project files. These would then need to be manipulated into a data structure and shown to the user on a graphical user interface. A detailed algorithm is then going to be needed to solve the Sudoku, producing a completed 9x9 Sudoku grid to the user on the interface.

Use Case Diagram



The diagram to the right shows the Use Case diagram for this project – it shows the user interaction for the project. In an attempt to keep my program as simple as possible, the amount of user interaction needed to make the project function correctly is limited to as little as possible. However, these methods that the user is required to trigger will drive the solving solution for the project.

Design

The design section of this documentation will contain the design of my data structures, including justifications for design decisions that I have made during the implementation of the program. It will also contain detailed descriptions of the various algorithms that I have used throughout the different classes of my program, and it will explain their specific roles in the creation of the program.

Sudoku.java

This class just contains the main method of the program and dictates what will be run at runtime. It creates a new SudokuDisplay and simply initiates the program.

```
SudokuDisplay d = new SudokuDisplay();
```

SudokuDisplay.java

This class contains the code that creates the graphical user interface that the user is able to interact with and see on their screen.

```
public class SudokuDisplay extends JFrame implements ActionListener {
```

This class declaration of extending the JFrame features that are within Java, allow me as the programmer to add a wide range of tools to create a more effective GUI for my user. These features, along with the implementation of Action Listeners allow the use of buttons, text fields and much more. JPanels were used to create the space to add the JButtons and JTextFields onto, and then I was able to implement the Listeners along with their accompanying actionPerformed method – which will be called when the user interacts with any of the JButtons.

```
if (e.getSource() == loadButton) {
	try {
		canvas.loadSudoku(load.getText());
	} catch (FileNotFoundException e1) {
```

```
                System.out.println("File " + load.getText() + " not found. Please
enter a valid file.");
```

The relevant error handling has been used to ensure that the user cannot crash the program with ease. This should offer an enhanced user experience as it will help the user to utilise the system easier. This specific piece of code, shows the process of what happens when the user interacts with the loadButton – the loadSudoku method, which is located in SudokuCanvas.java, and the text that the user has entered within the load JTextField is passed through as a parameter to be used in that method.

SudokuModel.java

This class is fairly basic, however contains two of the most important methods in the program. In this class, we declare a 2 dimensional array which allows you to create a grid array in the class constructor, using parameters passed in from elsewhere.

```
public void setValue(int row, int col, char r) {
        grid[row][col] = r;
```

This setValue method is integral to the project as it is heavily used in algorithms throughout the project. It allows me to store the value of each cell on the Sudoku grid, with the value's accompanying row and column coordinate. This allows me to load in files, and store the data into the grid array.

```
public char returnValue(int row, int col) {
        char x;
        x = grid[row][col];
        return x;
```

This method is also very important and is used greatly in the project algorithms. It returns the value (number in this case) of the cell depending of what row and column parameters are passed in. The value is then returned to wherever it was called from and can be manipulated to produce a solution to a problem.

SudokuCanvas.java

This class includes the main complex algorithms that will go about solving the Sudoku data files loaded in to the program. The class itself extends JPanel – which allows for the use of the Java Graphics components which are heavily used in the graphical user interface in this class.

The declaration of variables is key here, with a main 2 dimensional array being used to store the coordinate values of the data that will be read in of the Sudoku file, this declared as a char array[9][9] to allow for 81 possible coordinates, the standard size of a Sudoku board. I have used an ArrayList of type <String> to hold all of the usedNumbers that are found from my check methods (explained later), as these are of variable length which is perfect for the task that I have. The same has also been declared for an array called possibleNumbers, which will be explained later on.

```
ArrayList<String> usedNumbers = new ArrayList<String>();
```

Using the JPanel components, I have implemented a paintComponent method, which will be used to manage the drawing of the actual grid that the user sees on the interface. It simply takes an input parameter of Graphics g, and uses this information to call the drawGrid method on runtime - which draws the basic 9x9 Sudoku grid and the values that will be placed within the cells later in the program.

The drawGrid method uses the Graphics g component to produce the graphical user interface, using an iteration through i and j values that increase until they reach the preset values of the number of rows and columns (9). The grid is created using some simple mathematical equations – producing a 9x9 Sudoku grid, and also using the g.drawString function, allowing for future input of values to be shown as graphics on the grid.

```
g.drawRect((i *40),(j * 40), 40, 40);
g.drawString(value, (firstValuei + (i * 40)), (firstValuej + (j * 40)));
```

The variable 'value' that is passed through the drawstring function is defined as the code below. It uses the returnValue method that was mentioned within the SudokuModel class to find the string value of the corresponding cell that is being produced on the graphical grid. This is used further through the program to produce numerical graphics which have been stored using the setValue method.

```
value = String.valueOf(board.returnValue(j,i));
```

In this method, I have also used the 2D Graphics feature, and allowed myself to create 4 additional lines on the graphical user interface that act as thicker lines, to differentiate the sub grids in their own 3x3 grids. It uses the variables of width and height to ensure that mathematically accurate values are used in relation to the interface size.

```
int width = getSize().width;
int height = getSize().height;

g2.drawLine((width / 3), 0, (width / 3), height);
```

The loadSudoku method is a complex algorithm that includes the use of the BufferedReader and FileReader to load in the data from the file that the user enters in the textfield. This user input is passed in as a parameter to the load method, and then the file can be opened and manipulated by the algorithms below.

I have used two iterations through the number of rows and columns, this then allows the algorithm to check 81 data values – the number of grid spaces on the 9x9 Sudoku grid.

```
String sCurrentLine = br.readLine();
```

I have used br.readLine() to get the value of the first line of data from the text file, and then I have parsed off the relevant data for the correct coordinate using the index of the loop and index of the line of data.

```
x = sCurrentLine.charAt(j);
gridArray[i][j] = x;
board.setValue(i, j, x);
```

This value (which is either a number or space) is then added to the gridArray and I have utilised the setValue method in SudokuModel to assign the value found to the corresponding coordinate. There are also some command line printing for debugging purposes, testing purposes and additional user information to show them the data that has been stored in the system.

The fillPossibleCandidates method is called when the user interacts with the Solve button, it iterates through the rows and columns once again to check all 81 cells. In this method, the array coordinate is checked if it is equal to a blank grid space – if so, the three check methods are called to produce an arrayList of usedNumbers in the corresponding row, column and square that the cell in linked to. The solveCell method is then called with the input parameters of the iteration (i, j). This is the main driver for the solving algorithm.

```
if (gridArray[i][j] == ' ') {
        checkRow(i,j);
        checkColumn(i,j);
        checkSquare(i,j);
        solveCell(i,j);
```

The checkRow method is used to add values into the usedNumbers arrayList, the input parameters of row and col are the coordinates that are being checked. The loop iterates through the number of

columns (to check the row), a value of 'c' is found by checking the value of [row][i] – the row stays the same and the 'i' increases to look at each cell of the row.

```
for (int i = 0; i < COLUMNS; i++) {
        if (gridArray[row][i] != ' ') {
                c = gridArray[row][i];
                String s = Character.toString(c);
```

The char c is then changed into a String so it can be used during the usedNumbers arrayList – I have used the .contains() function to pass through the value found by the grid coordinate and if the value is already contained within the usedNumbers arrayList, then it is not added to the arrayList.

```
if (usedNumbers.contains(s)) {
        // Value is already present in arrayList - do not add
} else {
        usedNumbers.add(s);
```

Otherwise, it is added to the arrayList and a list of numbers that are linked to that cell is created. Both the userNumbers and possibleNumbers arrayLists are cleared at the start of the method, to ensure that the same code can be used by all cells of the Sudoku grid without complication.

The checkColumn method is very similar to that of the checkRow algorithm, however just makes slight adjustment to the iteration, in changing the number of columns to the number of rows. It also possesses the ability to check what is contained in the arrayList to ensure that duplicate values are not being added to the list. This method provides more detail for the program to later manipulate and helps the solving process.

```
for (int i = 0; i < ROWS; i++) {
        if (gridArray[i][col] != ' ') {
```

The checkSquare method was quite difficult to implement – I was deliberating the choices I had to code this algorithm for substantial time, however I had to settle for what I believe to be quite an inefficient method but was my only option given the time constraints. After much debugging, I was able to produce checks for each 3x3 sub grid but having a number of if statements that take into consideration the input parameters of row and column (examples below). This was difficult to implement as I needed a way to ensure that any cell that was being passed into the method was only checking the 3x3 sub grid that the cell is actually in, and this required some clever (and frustrating) mathematical equations. I have used iteration loops of varying values depending on which sub grid they are checking, to find the values in each sub grid. Once again, the values found are checked to see if they are in the usedNumbers arrayList and if not, are added to it for future use. Ideally, I would have liked to implement this method in a more efficient manner, however due to the time constraints of the project I had to settle for this way.

```
if (row <= 2 && col <= 2) {
if (row <= 5 && col <= 2 && row > 2) {
if (row > 2 && row <= 5 && col <= 5 && col > 2) {
```

The solveCell method contains the main solving algorithms of the program. The row and column coordinate are passed in as parameters, and are used as references for the setValue method which will allow for the values to be eventually printed onto the graphical user interface. Firstly, within a for loop of 1 to 9 (the possible Sudoku number values) it checks whether the number is present within the usedNumbers arrayList which contains all of the values found by the check methods described above.

```
for (int i = 1; i < 10; i++) {
```

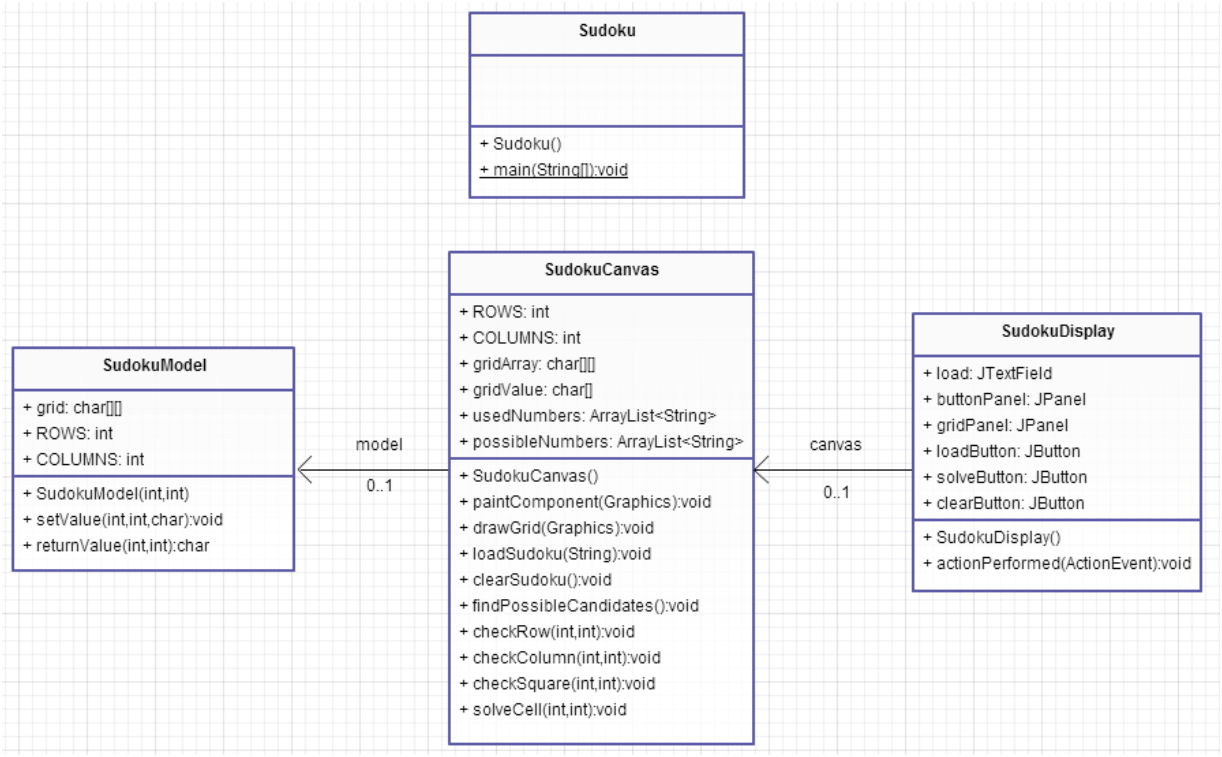
```
if (usedNumbers.contains(s)) {
        // Value is found, do not add
} else {
        possibleNumbers.add(s);
```

If the value is not found, then it is added to the possibleNumbers arrayList – these are the values that would fit in the cell of the grid. The size of the possibleNumbers arrayList is then checked to see if it is equal to 1, this means that there is only 1 possible value that can fit in the cell – and can therefore be printed to the GUI straight away with no further algorithms needed. The setValue method is used to add the information to the other data structures, and then repaint() is called finally at the end of the method to show the user the updated cell.

```
if (possibleNumbers.size() == 1) {
        String str = possibleNumbers.get(0);
        char c = str.charAt(0);
        board.setValue(row, col, c);
```

Class Diagrams

This UML diagram shows an outline of the classes that are within my program and how they are inter-related.



The Sudoku class is simply what the code will initiate first when the program is run at runtime - all it does is create a SudokuDisplay in the main method of the program.

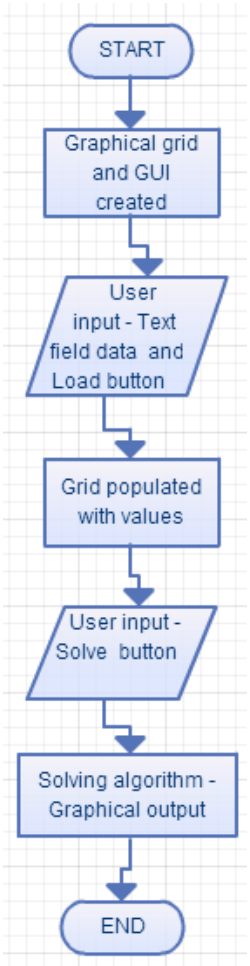
The SudokuModel provides the rest of the classes with two get and set methods that are heavily used to store the values of the Sudoku cells throughout the program. No real algorithms are present in this class, it is just a reference point for the rest of the code to call when needed.

The SudokuDisplay contains the majority of the code that represents and creates the graphical user interface of the program. It uses JFrame features and ActionListeners to implement buttons, panels, graphics and text fields that the user can interact with and effectively use the program.

The SudokuCanvas class contains the code of many of the complex algorithms that are used to load in, and manipulate the Sudoku data in order to solve the data. It also provides graphical tools to produce an output onto the GUI for the user to see.

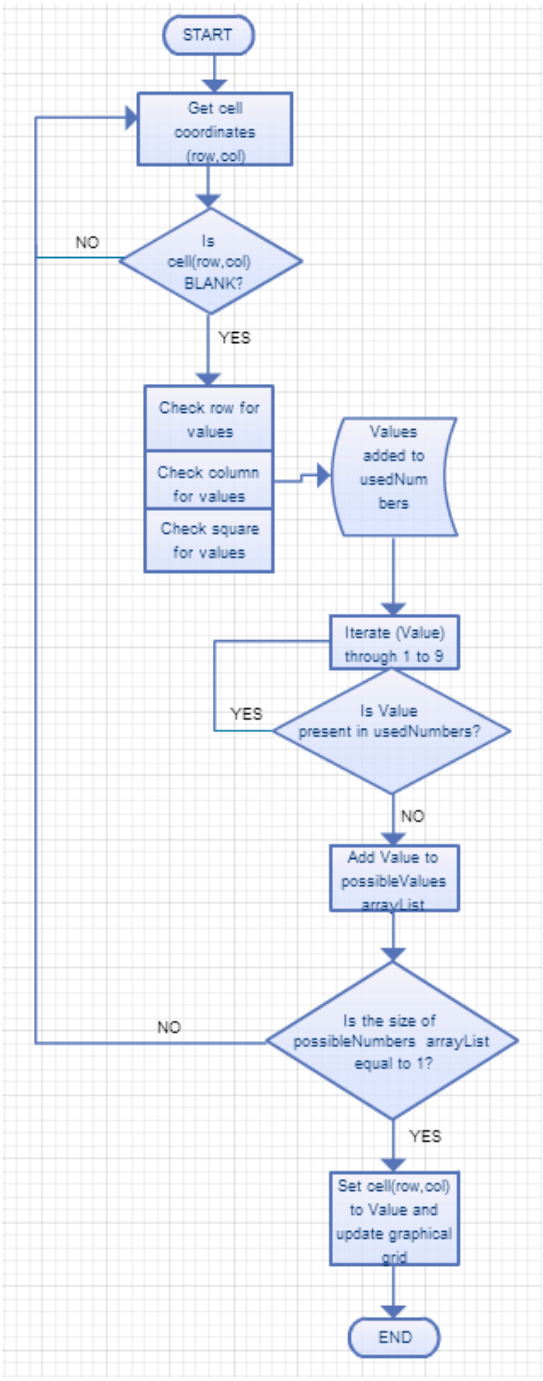
Flow Charts

Standard Program Flowchart



This flowchart (left) shows the processes that the program undertakes to solve the Sudoku, in a very simplistic format. It shows the main processes that take place and the user interactions that are needed, in order to make the program produce a solved Sudoku.

Solving Algorithm Flowchart



This flowchart (right) shows the processes and interaction that program undertakes to manipulate the cell data into a possible value to be put on the graphical grid. The algorithm works as getting the cell coordinates, and checking whether the cell is blank or not – then it will check the corresponding row, column and square of the cell and store the numbers that are present in those cells in an ArrayList data structure. Then, using a 1 to 9 for loop, will check if the value (1 to 9) is present in the data structure and if not, will add that number to another ArrayList. Finally, the size of the possibleNumbers arrayList is checked to see if it equal to 1, meaning that there is only one possible value for the cell and can update the graphical grid for the user to see. Hence, solving the Sudoku.

Testing

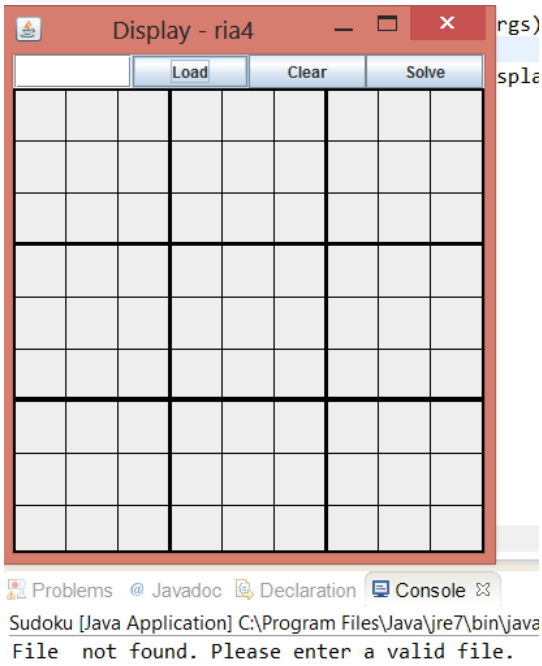
Test Table

The testing approach that I took to this project was to ensure thorough testing of the main features of the program. These are essentially the processes that involve user interaction with the system, and henceforth the parts of the program that could go wrong if the user were to enter incorrect values. The tests included seem relatively simple, however it is imperative that the tests are carried out to prove that the program works correctly as this will ensure a better user experience when interacting with the graphical user interface.

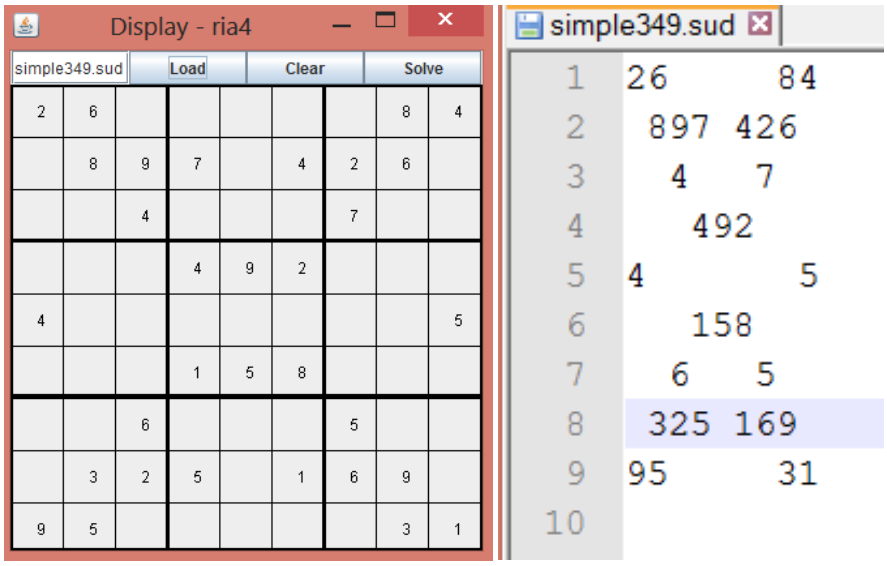
ID	Description	Inputs	Expected Outcome	Pass /Fail
1	Text field entry on Graphical User Interface	" " in Load JTextField	Program won't load anything, and will print out 'File not found' in command line.	P
2	Text field entry on Graphical User Interface	"simple349.sud" in Load JTextField	Program will load the text field with correct data, and print it onto the GUI.	P
3	Text field entry on Graphical User Interface	"simple349.sud" in Load JTextField	Program will load the data, and print array data in the command line which is equal to that on GUI.	P
4	User interaction (Click) on Load Button	" " in Load JTextField, and button click on Load	Program won't load anything and will print out 'File not found' in command line.	P
5	User interaction (Click) on Load Button	"simple349.sud" in Load JTextField, and button click on Load	Program will load data in and produce graphical output of data on the GUI.	P
6	User interaction (Click) on Clear Button	"simple349.sud" already loaded in, and button click on Clear	Program will clear the graphical grid of all simple349.sud data and change it to blank spaces.	P
7	User interaction (Click) on Solve Button	"simple349.sud" already loaded in, and button click on Solve	Program will 'solve' the Sudoku, producing graphical output of completed grid, filling in the empty spaces. Solution tested against online Sudoku Solver (www.sudoku-solutions.com)	P
8	User interaction (Click) on Solve Button	"simple351.sud" already loaded in, and button click on Solve	Program will 'partially solve' the Sudoku, producing a graphical output of a partially completed grid.	P
9	User interaction (Click) on Solve Button	"book55.sud" already loaded in, and button click on Solve	Program will 'solve' the Sudoku, producing graphical output of completed grid, filling in the empty spaces. Solution tested against online Sudoku Solver (www.sudoku-solutions.com)	F

Test Table Evidence

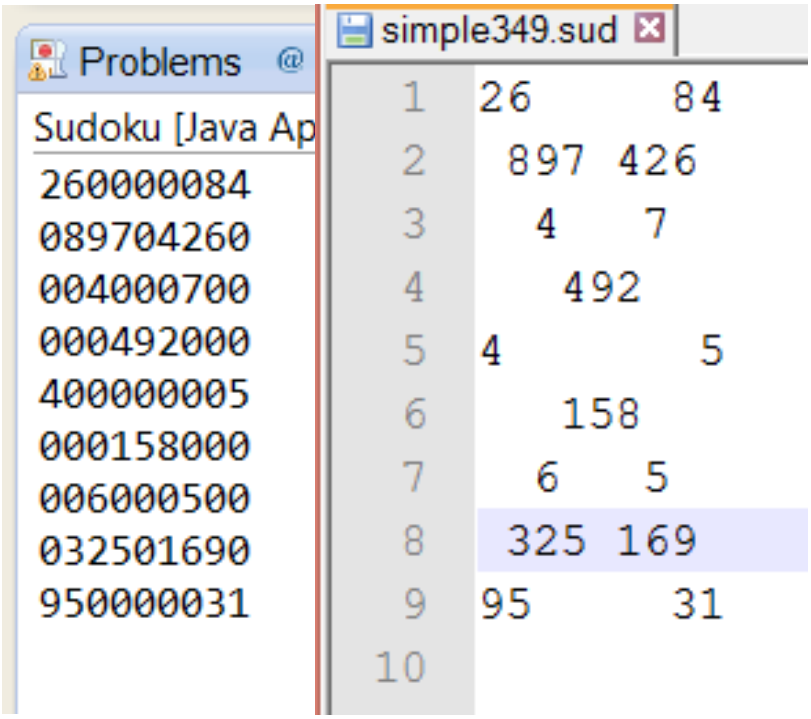
ID1



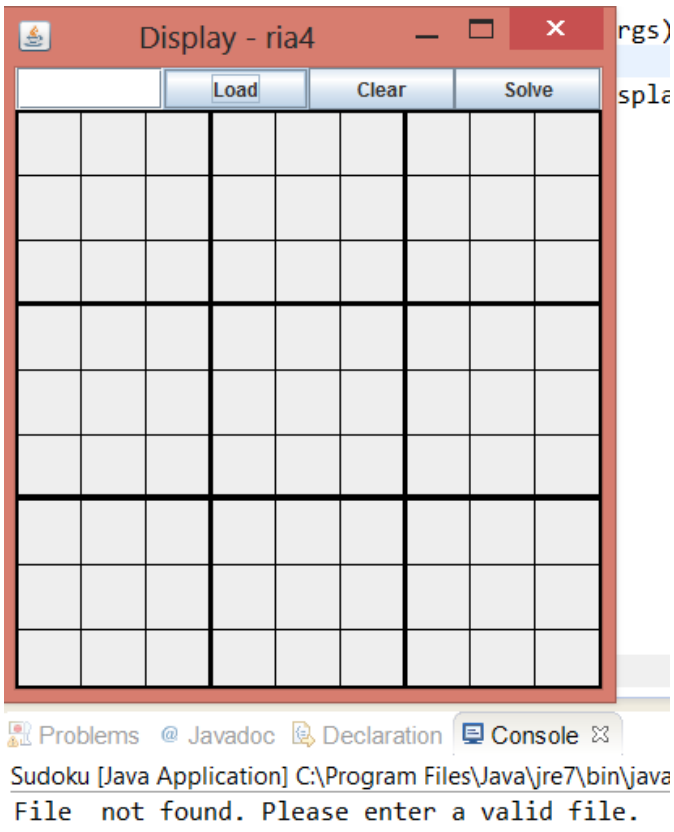
ID2



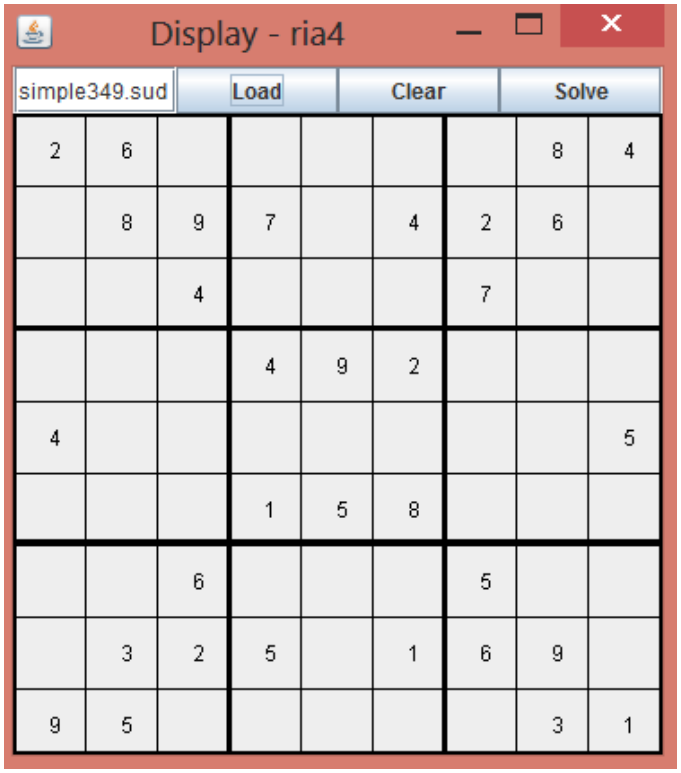
ID3


Problems
Sudoku [Java Ap
260000084
089704260
004000700
000492000
400000005
000158000
006000500
032501690
950000031
simple349.sud
1 26 84
2 897 426
3 4 7
4 492
5 4 5
6 158
7 6 5
8 325 169
9 95 31
10

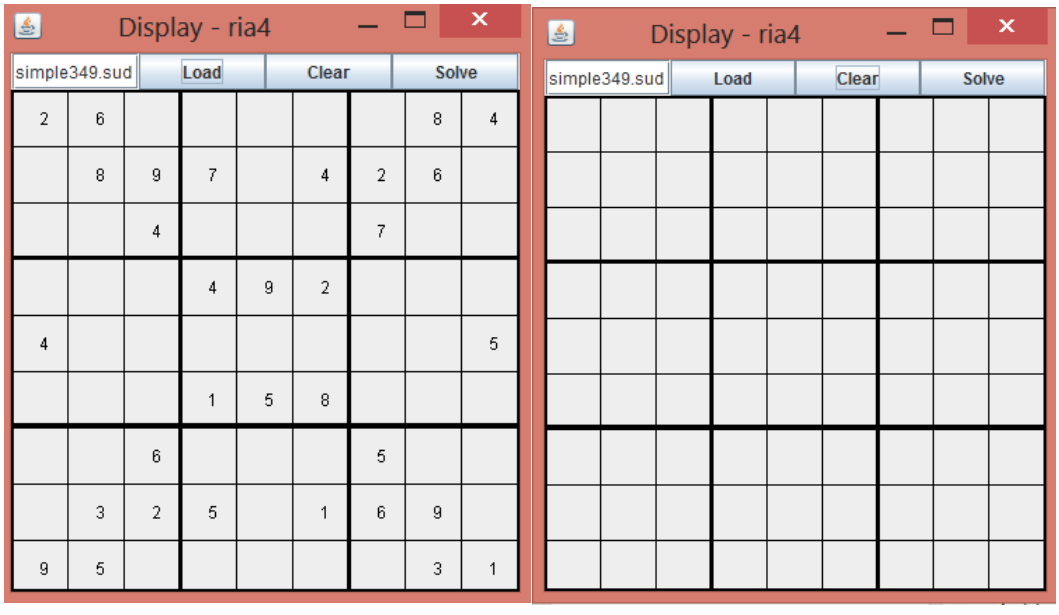
ID4


Display - ria4
Load
Clear
Solve
Problems
Javadoc
Declaration
Console
Sudoku [Java Application] C:\Program Files\Java\jre7\bin\java
File not found. Please enter a valid file.

ID5


Display - ria4
simple349.sud
Load
Clear
Solve
2 6 8 4
8 9 7 4 2 6
4 7
4 9 2
4 5
1 5 8
6 5
3 2 5 1 6 9
9 5 3 1

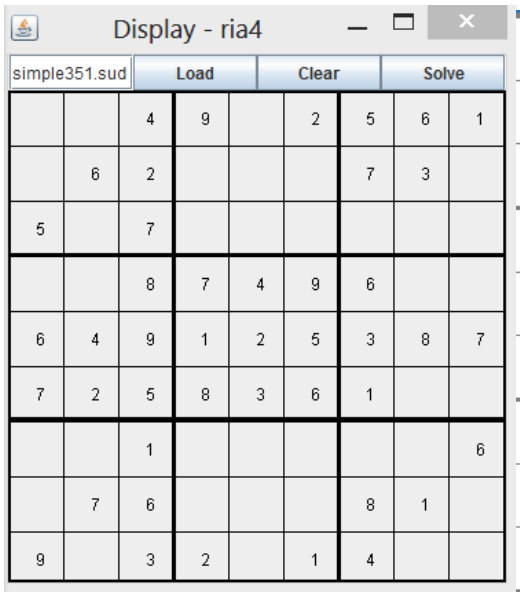
ID6


Display - ria4
simple349.sud
Load
Clear
Solve
2 6 8 4
8 9 7 4 2 6
4 7
4 9 2
4 5
1 5 8
6 5
3 2 5 1 6 9
9 5 3 1
Display - ria4
simple349.sud
Load
Clear
Solve

ID7

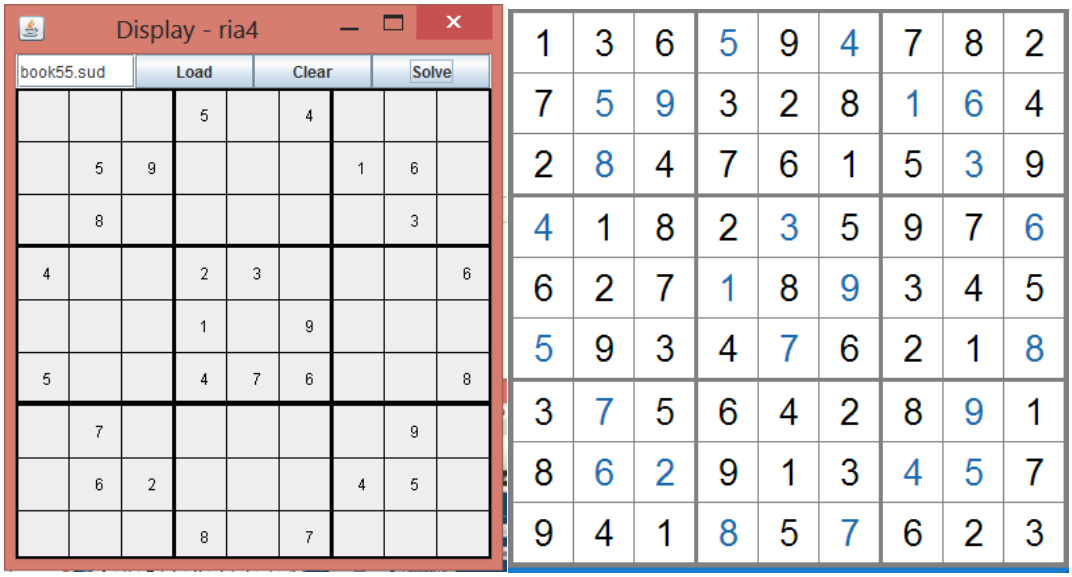
2	6	7	9	3	5	1	8	4
5	8	9	7	1	4	2	6	3
3	1	4	8	2	6	7	5	9
8	7	5	4	9	2	3	1	6
4	9	1	6	7	3	8	2	5
6	2	3	1	5	8	9	4	7
1	4	6	3	8	9	5	7	2
7	3	2	5	4	1	6	9	8
9	5	8	2	6	7	4	3	1

ID8



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

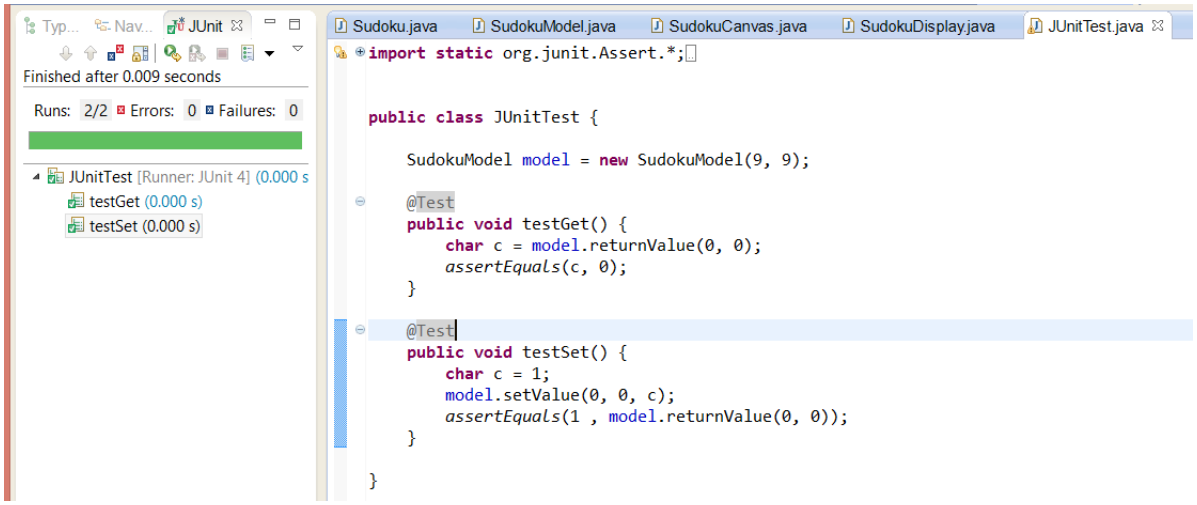
ID9



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

JUnit Testing

In addition to my testing table, I have included two JUnit tests. These are used to check whether my returnValue and setValue methods in the SudokuModel class are working correctly. There is a screenshot of the code and evidence of the tests passing. Ideally, I would have liked to provide more JUnit testing, however time constraints impacted this decision.



Evaluation

In review of my project, I am pleased with the final product that has been created due to the short space of time that was needed for completion. Using the skills that I had obtained from CS124 last year, I had no trouble using the Java Swing framework and implementing the graphical user interface. After much deliberation practicing with Linked List data structures, Stacks and Queues –all too little success, I deciding on using a 2 dimensional array for the grid values as this would enable me to store the coordinates of each cell on the 9x9 Sudoku board and allow me to reference this at later intervals. In addition to this, I am very familiar with 2 dimensional arrays and have experience in manipulating them when file handling which I knew was going to be used in this project. I have used two ArrayLists of type String, which are to store data values – these are used in the solving algorithm as explained earlier in the document. The reasoning behind using these ArrayLists was because they are of variable length, and did not require any size declaration – this meant I could simply add in the value when needed and also make use of the .clear() and .contains() functions that the ArrayLists provide. The choice of this particular data structure ultimately led to the solving of my algorithm.

I had limited success with the use of Linked Lists, I was able to get the Sudoku files reading into the data structures and printed out onto the graphical user interface. However, then manipulating them to solving the puzzle was too difficult for the time frame at hand and meant that I had to look elsewhere for data structures. If I had more time to complete this project efficiently, I would have researched the Linked List data structure. In addition, I would have tidied up the GUI, perhaps using a File drop down menu bar and the addition of different colours for the solved values as this would enhance the user experience. My solving algorithm only works with the easier data files, and I would have liked to produce one that solved all of the files given but perhaps that would be unrealistic for the time frame and complexity of the task.

In conclusion, given the time frame of the project and the complexity of the solving algorithm – I am pleased with my program and am glad that it solves Sudoku puzzles. I am also disappointed that the program requires multiple clicks of the solve button to work correctly. However, if I were given more time, I am extremely sure that the program and consequent algorithms could be improved greatly, thus increasing efficiency of the program. The project has been a good task of my capabilities in coding in Java, and has given me a number of extra algorithms to add to my code library which I will be able to implement into future projects.