Thomas Clarke, 2014

Final Year Project Report

Full Unit – Final Report

A Sudoku Solver Using Constraint Satisfaction (& Other Techniques)

Thomas Paul Clarke

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Dave Cohen

Department of Computer Science

Royal Holloway, University of London

March 25, 2014

Declaration

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 Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 24,600

Student Name: Thomas Paul Clarke

Date of Submission: 26/3/2014

Signature:

Table of Contents

Table of Contents

Table of Contents

Final Year Project Report **1**

Thomas Paul Clarke **1**

Table of Contents **2**

SVN Archive **7**

Chapter 1: **Constraint Satisfaction** **12**

1.1 Constraint Satisfaction Problem **12**

1.1.1 Constraint Diagram For Sudoku Solvers 14

1.2 Constraint Techniques **14**

1.3 Backtracking **15**

1.4 Improving the efficiency of backtracking **16**

1.5 Arc Consistency Algorithm **17**

1.6 Dynamic Variable Ordering **19**

1.7 Complexity, NP Hardness and the Big O Notation **20**

1.7.1 Why Is Sudoku in the NP Complete Class 21

1.7.2 SAT 21

1.7.1 Larger Sudoku boards 21

1.7.1 Big O Notation 22

Chapter 2: **Background Theory** **23**

2.1 Literary Survey **23**

2.1.1 Chapter 1 Resources 23

2.1.2 Chapter 2 Resources 25

2.1.3 Chapter 3 26

2.1.4 Chapter 3 28

2.1.5 Chapter 4 29

2.2 Techniques used by human Sudoku solvers **29**

2.2.1 Background 29

2.2.2 Latin Squares 30

2.3 Techniques used by human Sudoku solvers **31**

2.3.1 Using Pencil marks ... 31
2.3.2 Crosshatch Scanning/Naked Single ... 31
2.3.3 Hidden Single ... 32
2.3.4 Row & Column Checking ... 32
2.3.5 Elimination of Subsets ... 32
2.3.6 X Wing ... 32
2.3.7 Swordfish ... 32
2.3.8 XY Chain ... 33
2.3.9 Comparison of Methods ... 33
2.4 How Programs Solve Sudoku's ... 33
2.4.1 Backtracking (Recursive) ... 33
2.4.2 Exact Cover ... 34
2.4.3 Brute Force ... 34
2.5 Humans vs. Computers ... 35
2.6 Related Programs ... 36
2.6.1 http://www.solvemysudoku.com/ ... 36
2.6.2 http://www.sudoku-solutions.com/ ... 36
2.6.3 http://www.sudokuwiki.org/sudoku.htm ... 37
2.7 The Graphical User Interface ... 37
2.7.1 Displaying Inputs/Outputs ... 38
2.7.2 User Friendliness ... 39
2.8 Heuristics ... 39
2.8.1 Status Of The System ... 39
2.8.2 The User Experience ... 40
2.8.3 Consistency ... 40
2.8.4 Error Prevention ... 40
2.8.5 User Relevant Information ... 40
2.8.6 Design Accessibility ... 40
2.8.7 The Design ... 41
2.8.8 Error Handling ... 41
2.8.9 User Assistance ... 41
2.9 Data Structures ... 41
2.10 Examples Of Data Structures ... 42
2.10.1 Linked Lists ... 42
2.10.2 Tree Structures ... 43
2.10.3 Stacks ... 44

2.10.4 Queues 45
2.10.5 Heaps 45
2.10.6 Hash tables 46
2.11 Swing Worker 47
2.12 Layout Managers 47
2.13 The General layouts 48
2.13.1 Boarder Layout 48
2.13.2 Box Layout 48
2.13.3 Flow Layout 49
2.13.4 Group Layout 49
2.13.5 Grid Layout 49
2.13.6 The Features Of Layout Managers 50
2.13.7 Advantages and Disadvantages 50
2.14 Design Patterns 51
2.15 Using Different Design Patterns 52
2.15.1 Model View Controller 52
2.15.2 Flyweight Pattern 53
2.15.3 Composite pattern 53
2.15.4 Strategy pattern 54
2.15.5 Singleton pattern 54
2.15.6 Advantages and Disadvantages 55
Chapter 3: **Design & Implementation 56**
3.1.1 Programming Structure 56
3.2 Description of Classes 57
3.2.1 <Interface> SudokuMenuView Class 57
3.2.2 SudokuMenuController Class 58
3.2.3 SudokuMenuModel Class 58
3.2.4 FileStore Class 58
3.2.5 XMLHandler Class 59
3.2.6 SudokuController Class 59
3.2.7 CellView Class 59
3.2.8 SudokuBoardModel Class 60
3.2.9 Network Class 60
3.2.10 Request Class 60
3.2.11 Variables Class 60

3.2.12 <Interface> Constraints Class 60
3.2.13 Domain Class 61
3.2.14 AllDifferent Class 61
3.2.15 The Network 61
3.3 Sequence Diagram 62
3.4 Design Research 62
3.4.1 Influences From Other Programs 62
3.5 Graphical User Interface Design Concepts 63
3.6 Design Implementation 64
3.7 Evaluation Of Designs 65
3.8 Using Layout Managers 67
3.9 Using Data Structures 67
3.10 Concept Programs 68
3.10.1 Game Tree 68
3.10.2 Boards 69
3.10.3 Duplicate Boards 69
3.10.4 Symmetric Boards 70
3.10.5 Positions 70
3.11 Algorithms 71
3.11.1 Brute force 71
3.11.2 MiniMax 71
3.11.3 Alpha-Beta Pruning 72
3.11.4 How this relates to Sudoku 73
Chapter 4: **Refactoring 75**
4.1 User interface 75
4.2 Sudoku Grid 75
4.3 Creating a network 76
4.4 Sudoku board Model View Controller 77
4.5 Observer 77
4.6 Swing Worker 77
4.6.1 Swing worker problems & resolution 79
4.7 Structure changes 79
4.7.1 Updated UML 80
Chapter 5: **Evaluation 81**
5.1 Graphical User Interface 81

5.1.1 Load Window 81
5.2 Algorithm Results 82
5.3 Complexity of Algorithms 83
5.3.1 Backtracking 83
5.3.2 DVO 83
5.3.3 GAC 84
5.3.4 The hardest 'found' board results (Please see footnote 42): 84
5.4 Software Engineering Concepts 85
5.4.1 Test Driven Development 86
5.5 Critical Analysis Of Program 87
5.6 Evaluation 89
5.7 Conclusion 90
Chapter 6: **Professional Issues 92**
6.1 Software Piracy 92
6.1.1 Threats 92
6.1.2 Illegitimate software 92
6.2 Prevention 93
Bibliography 95
Appendix 1 98
Appendix 2 102

SVN Archive

Entire Project SVN

https://svn.cs.rhul.ac.uk/personal/zwac016/Year%203/Final%20Project/

Abstract

This report describes the process of creating and implementing a program that solves Sudoku boards using constraint satisfaction algorithms. The program was written in Java and the graphical user interface was created using WindowBuilderPro. This report includes the background theory needed to program the Sudoku solver and concept programs that have been created in order to demonstrate key algorithms in both the programs and the reports. These algorithms have been adapted to solve the Sudoku in the fastest time, as well as other methodologies that will allow the program to be structured and designed correctly.

This project intrigues me, as Sudoku is something I've done as a mental exercise. Yet I've never considered the programming and design aspects (let alone the theory behind its complexity and algorithms) needed to solve some boards. Additionally, this project helped me to refine my skills with both programming and report writing, as I needed to work at both. Having a project balanced with theory, programming and analysis was important for me. Another essential factor was the research, as the algorithms involved a high degree of research and I had not implemented them before.

When the project is completed I'll have a working program that I've designed and created that I might decide to show to potential employers. Or I may continue to develop my program after this stage is completed. Being able to create a program professionally in this manner will prove my ability to employers, as well as giving me the confidence of what I can achieve myself.

The following points outline the structure and content within the project:

- The objectives and requirements of the program, as well as the functions I'll need to create.
- The main theory involved with a CSP and a second chapter involving the other aspects of the theory that can be included within the program.
- The Concept program chapter further describes the theory involved behind the implementations I made in the first term.

- The Design and implementation chapter describes the application of creating a program and the stages involved when programming it.

- Refactoring the program involved learning new methods of programming and the refactoring chapter describes the issues found and how I resolved those problems.

- The evaluation includes the reflective outlook of the program, how the program performed and the results of the tests from the algorithm comparisons.

- Professional issues goes into detail on the current threats of downloading programs without the knowledge of what they contain.

Project Aims & Objectives

Sudoku is a complex number puzzle. Its structure is a three by three box and three by three block matrix, where the numbers 1-9 are to be used in a non-repetitive sequence, horizontally & vertically.

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

My aim is to create a constraint satisfaction problem in the form of a Sudoku solver. The program will contain a Sudoku board input system and allow the puzzle to be solved using constraint satisfaction algorithms. I will also write a report consisting of key explanations and diagrams analysing the project's significant concepts and outcomes, as well as the algorithms used to build the Sudoku Solver.

The Aims & objectives within the program:

1. Write a report and program on a Constraint Satisfaction Problem in the form of a Sudoku Solver.

2. Design the program using research from other existing Sudoku solvers in order to implement the desired functions. Design patterns will also be included to increase the performance and structure of the program.

3. Create a Graphical User Interface using layout Managers that will provide all the functions and user interactions needed. This will include using layouts to create the Sudoku grids. They will allow a user to input the function to solve a Sudoku manually in addition to the algorithm functions.

4. Creating a constraint network that will contain the CSP that will need to be solved. It will include variables, domains and constraints that will use data structures to provide effective ways to store information and manipulate it.

5. To explore and implement new algorithms that will solve the Sudoku in a faster time and provide other functions that will allow the usability of the program to be increased. These will include:

6. Implementing a simple backtracking algorithm on the Sudoku to solve it, using the basic backtracking algorithm will then provide a foundation for time complexity improvements.

7. Implement a dynamic variable ordering algorithm that increases the speed of the solution and a 'next best cell' function, using the priority queue data structure.

8. Implement Generalised Arc Consistency to further increase efficiency and to reduce the domains of the variables within a scope.

9. As an extension I want to incorporate a save and load function to allow many boards to be saved if they are not finished. This will be done via an XML parser.

My Diary, Plan and Time Scale (See Appendix 1)

Chapter 1: **Constraint Satisfaction**

I aim to create a Sudoku solver that uses constraint satisfaction to solve puzzles. To do this the input values are only constrained by the values that they cannot be. Ideally an algorithm is needed to restrict these possible values using the other constraints available. i.e. the all different constraint.

1.1 Constraint Satisfaction Problem

Constraint satisfaction is a method of seeing problems that can be a series of 'if' statements. Thus meaning that there are questions needing to be answered for each variable. If all of these questions are answered correctly then that will provide a correct answer for the variable. Each variable will have at least one other linked variable that will then also depend on that value too. The constraints need to be true for the value to be verified. The more complex the constraint the harder it is to find a set of answers that satisfies them all.

A constraint satisfaction problem is a paradigm that leads to the use of the best solving strategies. Strategies will be discussed later but they are associated with effective algorithms to solve problems that relate to constraints. These algorithms increase the effectiveness of the constraint satisfaction by manipulating the data so there are fewer options for the possible values to be.

A CSP can be defined as a triple, $\langle X, D, C \rangle$. This consists of a list of variables X that are the placements that are requesting a value. The domain D, are the possible values that the current variable can take, each variable contains a domain that will contain its true value. C is the constraint that is taken upon the variable that will check the possible values within the domain. This allows an entire dataset to be linked with the constraints and will be used within the Sudoku solver program.

A constraint[1] is split into two parts: the scope and the relation. The scope consists of the variables that are affected by the constraint that will be able to provide the possible values that the variable can take. This is called the domain; a list that contains the possible values after the constraint has been made. The relation is the constraint type that needs to be satisfied in order for a value to take place within a variable. The relation uses the scope of a variable to check the domains of each variable within that scope; an example relation could be for all the values to be different. This allows all the possible values to be found, as they cannot be values from the other domains.

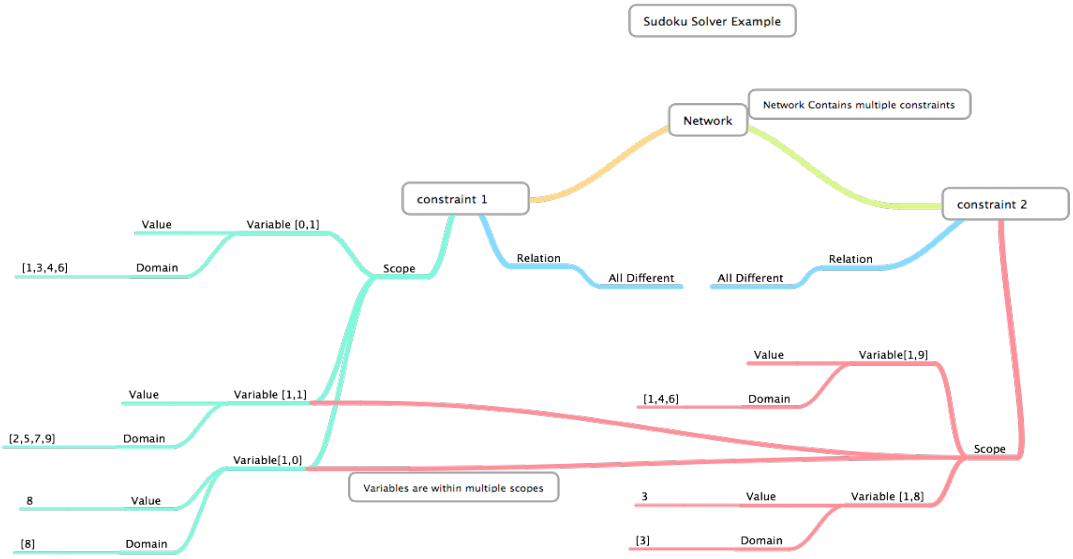
The variables within Sudoku will be a location on a board that needs to be checked if it's in a 'correct' or a 'false' state. The domain is held within the variable, with the values 1-9, until an assigned value is present within the scope. The relation then permits the number in the domain to be removed, as it could not possibly be assigned that value and for the board to be returned in a true or fully correct state.

Use of constraint satisfaction problems are often linked to logic puzzles like Eight Queens and Sudoku. However they're designed to be able to solve very large data sets that can be manipulated with constraints. Sudoku is an ideal CSP as there are many values to be found all with constraints. In this case the constraint uses all of the values within the subject (selected variables) scope must be different. There are many different ways to apply constraints and each program needs to have all of the rules in place so it can be applied correctly.

[1] Professor David Cohen. The Complexity of the Constraint Satisfaction Problem. ISG-CS Mini-Conference Slides http://www.rhul.ac.uk/computerscience/documents/pdf/csisgmini-conf/dac.pdf

1.1.1 Constraint Diagram For Sudoku Solvers

The diagram below shows the concept for the Sudoku Solver as a CSP problem:



1.2 Constraint Techniques

Within constraint satisfaction there are techniques that can be used to increase efficiency. These are deterministic; they will produce the same results when searched, and every time it's performed on the same set of values. So given a particular input there will always be a constant output[2]. It's very useful, as it allows the computation to be made as soon as possible.

To improve the constraint it's possible to select the objects that are no longer needed due to a variable being domain consistent. Thus meaning that within the domain of the variable there contains a value that is matched within the constraints[3]. When a domain is checked there are variables within its scope that are assigned. The assigned value cannot be the value of the current variable and as a result should be removed from the domain. Furthermore it's also possible to remove domain values from other variables domain values, as they are also consistent. By

[2] http://technet.microsoft.com/en-us/library/aa214775(v=sql.80).aspx
[3] http://www.cs.ubc.ca/~kevinlb/teaching/cs322%20-%202009-10/Lectures/CSP3.pdf

pruning the domains as effectively as possible it allows the advantage of efficiency without checking impossible values via backtracking.

1.3 Backtracking

I've used a method called backtracking and unlike forward checking and arc consistency (discussed later in the report) it doesn't have any propagation. Backtracking works by searching all the possible values. If the outcome matches the desired output then it passes.

Backtracking is a recursive algorithm used to find a desired output through the use of the 'brute force' approach. It works by starting at the root of the tree and then follows a consistent path until the end of the path within a 'leaf'. When the leaf is evaluated into the correct output the algorithm stops and the value is found. If an incorrect node is found the process is repeated by going back to the parent of the leaf and following and checking another node. This is done until the full tree has been traversed and there are no correct nodes or if a correct node is found and the recursion stops. [4]

This is the backtracking code I've used within the 8Queens concept program:

```
private boolean backtracking (int col) {
   int row = 0;
   if (col == boardSize){
      return true;
   }
   else{
      boolean attempt = false;
      while((row < boardSize) && !attempt){
         if (check(row,col)){
            row++;
         }
```

[4] http://www.cis.upenn.edu/~matuszek/cit594-2012/Pages/backtracking.html

```
            else {
               board[row][col] = 1 ;
               attempt = backtracking(col+1);
               if (!attempt ){
                  board[row][col] = 0;
                  row++;
               }
            }
         }
         return attempt;
   }
   }
```

This algorithm uses the current attempt to see if this passes and if it doesn't pass then the attempt is withdrawn and another attempt is used. Although this means that, in the worst case, the algorithm needs to perform O(n^k) times. This number (if n) is the number of unassigned values within the grid and k is the number of possible values, k = 9 in the Sudoku backtracking algorithm. This, given the 'right/wrong' board could provide very poor results.

1.4 Improving the efficiency of backtracking

Backtracking is a very poor method to solve CSP's, as it checks every possible value for every single variable. This is inefficient and doesn't take into account the other assigned values or the other variables domains within its scope. The algorithms discussed in this section will improve backtracking and reduce the number of checks made.

1.5 Arc Consistency Algorithm[5]

Arc Consistency is different because the domains are reduced and there are propagates of each state. The algorithm is used to simplify the constraint satisfaction problem by continually iterating through the constraints and their implications of the variables one at a time. The difference between the generalised arc consistency and arc consistency is the use of non-binary constraints using tuples of variables instead of a pair.

In order to increase the efficiency of solving the Sudoku the domains of each variable could be reduced, as every constraint within its scope (that has a linked value and the domains) could be compared and reduced if needed. This will reduce the possible values that backtracking needs to perform, as it only uses values that are possible within the domain. If these are done early then the algorithm starts off with fewer values to process. Restricting the arc needs to be implemented as soon as possible or at least very early on in the process.

The algorithm below is displayed with a 'line-by-line' commentary:

Generalized arc consistency algorithm[6]

1: Procedure GAC (Variables, Domain, Constraints) [7]

2: Inputs

3: *V* // a set of variables

4: *domain*// a function such that *domain (X)* is the domain of variable *X*

5: *C* // set of constraints to be satisfied

6: Output

7: //arc-consistent domains for each variable to cut down domains that increase efficiency.

8: Local

[5] A filtering algorithm for constraints of difference in CSP's by Jean-Charles REGIN Pages 362 -367

[6] http://artint.info/html/ArtInt_79.html

[7] http://www.youtube.com/watch?v=Ge-XiX_tP4c - Doug Fisher's video on Generalized Arc Consistency

9: D_X //is a set of values for each variable X

10: *TDA* // is a set of arcs

11: for each variable X do

12: $D_X \leftarrow domain(X)$

13: $TDA \leftarrow \{\langle X,c\rangle \mid c \in C \text{ and } X \in scope(c)\}$ // For all the variables create the scopes, each variable within the domain creates the scope.

14: while ($TDA \neq \{\}$) // While the TDA set is not empty, meaning all the constraints have not been satisfied. When they have stop.

15: select $\langle X,c\rangle \in TDA$; // For each constraint within the TDA

16: $TDA \leftarrow TDA \setminus \{\langle X,c\rangle\}$;

17: $ND_X \leftarrow \{x \mid x \in D_X$ and some $\{X=x, Y_1=y_1,\ldots,Y_k=y_k\} \in c$ where $y_i \in D_{Yi}$ for all

18: if ($ND_X \neq D_X$) then // if the new domain is not the same as the old domain due to it being reduced.

19: $TDA \leftarrow TDA \cup \{\langle Z,c'\rangle \mid X \in scope(c'), c'$ is not $c, Z \in scope(c') \setminus \{X\}\}$ // Within the TDA set now that the domain has changed it needs to be considered again, so that it can be checked, due to the new domain changes within a constraint, that is in it's scope.

20: $D_X \leftarrow ND_X$

This algorithm prevents redundant boards being created, as the given domains are reduced to prevent propagation. Instead of trimming the tree the process has prevented unnecessary items from being created. Therefore, the constraint will increase the efficiency of the program.

This algorithm is used to increase the performance of backtracking, and does this effectively due to pruning the domains and searching redundant values. This efficiency boost on time is $O(v^2 d^2)$, where v represents the variables that are not assigned, and the d is the number within the domain. This will be discussed in section 1.7.

It has been implemented within the program and the code is featured the showcase section 6.

1.6 Dynamic Variable Ordering

This is a process that uses the most constrained variable to be the first one that's utilised. Its purpose is to allow the dead ends to be discovered as early as possible. Also, without needing to be considered separate from other variables, this effective heuristic cuts down the search space. [8]

Dynamic ordering means that the choice of the next variable used will be dependent on the state of the search. This is found using forward checking, which is the opposite concept of backtracking. It means that the current domain is dependent on the previous and current constraints. The reason for this is to use the best possible variable in order to increase the efficiency.[9]

The concept of dynamic variable ordering is to compare all the available variables and to control which constraints will be compared first. Using the variable that has the most constraints does this. If the number of constraints is high then it is likely to produce the highest number of fails when tested, as it allows the most constrained. Its function is to get as many of the failed tests done in the early stages so that they do not have to be replicated later on in the process. This will improve the efficiency when searching and creating the rest of the board. [10]

This could benefit the Sudoku Solver, because it'll allow the board states to be compared and searched in an effective way. Therefore, the states that will fail are found as early as possible. It will be used when the next cell needs to be populated within the grid, using the tightest constraints on the cell to produce the failed states. Please see example below:

[8] www.ics.uci.edu/~dechter/courses/ics-275a/spring.../chapter5-09.ppt slides 26-31
[9] http://ktiml.mff.cuni.cz/~bartak/constraints/ordering .html
[10] http://www.dcs.gla.ac.uk/~pat/cp4/papers/95_14.pdf

As an example, this can be implemented using a queue that will compare all variables with their current scope variables and the more that are assigned; the higher the selected variable will be placed. This, with a backtracking algorithm that will recursively check the values for the most restriction values within its scope, allows the algorithm to store all the variables in a list, so that the most likely to be assigned will be first and there will be more restrictions available to impact on the next variables.

The increase in efficiency for DVO, in its worst case, can be the same as normal backtracking. If a CSP was created where each variable had the same amount of constraints and the domain contains the same values then it would not affect the complexity.

1.7 Complexity, NP Hardness and the Big O Notation[11]

Due to the nature of the Sudoku I will be creating search algorithms that will find the possible values to satisfy the constraints and this could in some cases be searching by providing every possible value within every unassigned cell. This will have a poor time complexity and thus modifications will be made to backtracking to reduce the complexity. [12]The reasons problems are defined within classes are to assess the kind of scalability it will need. For example if a problem is within P it can be computed within polynomial time meaning the number of steps included are reasonable to the problem.

A problem can occur within the NP Class if a solution of the problem can be given and checked within polynomial time. This allows a set of problems that are not polynomial, but can be computed and their validity measured. When a problem is NP Complete it means that a problem within the NP class can be transformed into the current problem and this can be done in polynomial time.

[11] Discrete Mathematics & its Applications, 6th Edition. by Kenneth H. Rosen. Page 193
[12] Time and Space, Designing Efficient Programs. Adrian Johnstone & Elizabeth Scott. Chapter 5,Page 68 – 75

1.7.1 Why Is Sudoku in the NP Complete Class

Any way to solve the Sudoku involves using every possible combination of values; this means that the Sudoku grid size increases the values to be considered exponentially. Sudoku grid size of 9x9 is not within the class as the NP Complete class contains decision problems that have different size n and if n increases in size the computations required to solve that problem increase heavily. This means a board size of n x n[13] is within the NP Complete Class. However to prove that a that Sudoku boards are NP Complete the Latin Squares problem[14] needs to be considered discussed in chapter 2.2.2, This problem is within NP-Complete and can actually be incorporated into a Sudoku board since it contains the same rules just in a smaller scale, due to this an entire Sudoku board can be composed of multiple Latin square problems within polynomial time[15], this then concludes that Sudoku falls within the NP Complete class as well.

1.7.2 SAT[16]

Once a constraint network is created there are then many possible values for each variable, let alone for each variable to satisfy its own constraints from the other variables. A SAT problem is the concept of giving values to the variables that all return true, this is done with the backtracking algorithm that enters the values 1-9 all of the possible values and goes through one by one and recursively iterates through them until they are all returned true. SAT is within the NP Complete set and is the method used within the same algorithm I am performing on the Sudoku to solve it, i.e. backtracking. This means that Sudoku of an nxn size board is within the NP Complete class.

1.7.1 Larger Sudoku boards

Creating a larger board as discussed: for example solving a Sudoku double the size of 18x18 squares; would pose a larger problem to the backtracking algorithms I'll be implementing. The size of the space will increase exponentially. Thus I will use Dynamic

[13] http://www.math.cornell.edu/~mec/Summer2009/Mahmood/More.html
[14] http://www.cs.ox.ac.uk/people/paul.goldberg/FCS/sudoku.html
[15] Complexity and Completeness of Finding Another Solution and Its Application to Puzzles by Takayuki YATO and Takahiro SETA Theorem 4.6 The problem of partial Latin square completion is ASP-complete. www-imai.is.s.u-tokyo.ac.jp/~yato/data2/SIGAL87-2.pdf
[16] http://www.cosc.canterbury.ac.nz/csfieldguide/student/Complexity%20and%20tractability.html

Variable ordering or GAC algorithm. The larger boards will take a lengthier amount of time to compute. They are not effective with a standard 9x9 board with a hard board that proves harder to solve. (See hard board results in chapter 5.3.4).

1.7.1 Big O Notation

Programmers use the big O notation to specify the number of visits, in order, n analyses the performance of an algorithm. The use of the O notation is used to describe how the run time scales in relation to the input of the algorithm. An example of a Big O Notation is O(n!). This actually means, that for every possible element of n, it is explored.

Analysis of the algorithm complexity is within the evaluation section. (See 5.3 for complexity of algorithms).

Chapter 2: **Background Theory**

2.1 Literary Survey

2.1.1 Chapter 1 Resources

Professor David Cohen. The Complexity of the Constraint Satisfaction Problem. ISG-CS Mini-Conference Slides

http://www.rhul.ac.uk/computerscience/documents/pdf/csisgmini-conf/dac.pdf

This presentation gave the basic understanding of the Constraint satisfaction problems and a brief description of the complexity involved. There was a very clear definition of a CSP on slide 14 that aided my chapter on constraint satisfaction, as well as the understanding of the project I was about to undertake.

http://technet.microsoft.com/en-us/library/aa214775(v=sql.80).aspx

Contains a definition of deterministic and non-deterministic functions that allow me to confirm a definition for my broader understanding.

http://www.cs.ubc.ca/~kevinlb/teaching/cs322%20-%202009-10/Lectures/CSP3.pdf

Contained a very good example of the theory behind Arc Consistency and constraint networks as to what needs to be included within them. This further linked to what I needed within the arc consistency, what it could do to improve the backtracking algorithm and reduce the time complexity due to reducing all of the domains.

http://www.cis.upenn.edu/~matuszek/cit594-2012/Pages/backtracking.html

A very extensive description of the backtracking algorithm, within the description is the basic a layout of the algorithm, but it also involves a debugging section that helped to actually apply the algorithm as well as the description of the search tree.

A filtering algorithm for constraints of difference in CSP's by Jean-Charles REGIN Pages 362 -367

Gives a very detailed and technical report on a CSP and the algorithms involved with reducing the domains. Gave a very detailed review of the generalized arc consistency algorithm and how to implement it. The Zebra Problem was a good example and I was able to relate what I was currently trying to achieve with a relatable CSP, I found that helpful for understanding purposes.

http://artint.info/html/ArtInt_79.html

Contains a very good description of the Generalised Arc Consistency with a very good and clear algorithm as figure 4.3, and a network diagram as figure 4.4. This was the algorithm featured within this report as it was the clearest for me as well with the description.

http://www.youtube.com/watch?v=Ge-XiX_tP4c - Doug Fisher's video on Generalized Arc Consistency

This video gives a further description of the same algorithm and network diagram as in the http://artint.info/html/ArtInt_79.html source; this gave a very long and well-explained annotated version.

www.ics.uci.edu/~dechter/courses/ics-275a/spring-2009/chapter5-09.ppt slides 26-31

A PowerPoint presentation that includes a very detailed description of all the algorithms that I will implement, including backtracking, forward checking, GAC and DVO, as well as displaying the algorithms it also describes their complexity.

http://ktiml.mff.cuni.cz/~bartak/constraints/ordering.html

Discussed variable ordering to allow a definition and its relation to backtracking. This was helpful to expand on variable ordering for my broader understanding.

http://www.dcs.gla.ac.uk/~pat/cp4/papers/95_14.pdf

A tutorial on a constraint programming that was a very interesting to read and showed very good examples that related to the 8 queens problem. This also discussed a lot of the

further needed information about algorithms and the variable ordering. Complexity and Completeness of Finding Another Solution and Its Application to Puzzles by Takayuki YATO and Takahiro SETA Theorem 4.6 The problem of partial Latin square completion is ASP-complete.

www-imai.is.s.u-tokyo.ac.jp/~yato/data2/SIGAL87-2.pdf

An extensive report on the current proof of the Latin Square problem belonging to the NP Complete Class, used very formal definitions and was hard to understand, however it does prove the polynomial time ASP reduction & proves the SAT is ASP complete.

2.1.2 Chapter 2 Resources

P, NP, and NP-Completeness by Oded Goldreich pages 151-154

The optimal search algorithms for NP chapter gave a clearer understanding of the time complexity within a search algorithm and this was relatable to my current project.

Time and Space, Designing Efficient Programs. Adrian Johnstone & Elizabeth Scott. Chapter 5, Page 68 – 75

Explanation of the background of the NP sets and how they relate to some algorithms, I was able to get a further understanding of the use of backtracking algorithm and why it needs to be minimised within large data sets.

Algorithms and Complexity 2, CS2870 By Gregory Gutin February 16, 2013. Page 93-97

Much like the previous document, however it gives a much wider range of NP – complete problems and also included a description of 3-SAT and its definition. The main advantage of this source was that on page 97 it explains the stages of proving that a problem is NP-complete.

http://www.cosc.canterbury.ac.nz/csfieldguide/ComplexityTractability.html

Detailed report that has good interactive applets that aid the explanation of the traveling salesman problem, this then goes on to explain the rest of the problem and involves a very simplistic but easy to understand concept of algorithm time complexity.

http://www.math.cornell.edu/~mec/Summer2009/Mahmood/Home.html
Discusses the math's behind Sudoku with an interesting chapter on a 4x4 that describes the possible boards of just a very small board size, describing the problem of "the number of distinct 4×4 Sudoku solutions is 4!×2×2×3=288".

http://www.cs.ox.ac.uk/people/paul.goldberg/FCS/sudoku.html
Discusses the theory of why Sudoku nxn is within NP complete, using the Latin Square problem, which makes up a Sudoku differing slightly with the constraints. This was helpful to read.

2.1.3 Chapter 3

http://www.theguardian.com/media/2005/may/15/pressandpublishing.usnews
A background of the Sudoku puzzle and how it came about, it was interesting to read about the past of Sudoku to lead into the methods used by humans to solve them.

http://www.stolaf.edu/people/hansonr/sudoku/explain.htm
http://www.sudoku-solutions.com/solvingHiddenSubsets.php#hiddenSingle
http://www.decabit.com/Sudoku/NakedSubset
http://www.decabit.com/Sudoku/XWing
http://www.angusj.com/sudoku/hints.php#sordfish
http://www.decabit.com/Sudoku/XYChain
http://www.decabit.com/Sudoku/Techniques
These sources all contain the methods of human Sudoku solving. They allowed me to further understand the methods that could be used with limited computation. They contain a simple explanation of what needs to be done to find the current value using the specific scope, these were much more restrictive as they are not general and sometime require the scope to specifically placed to be relevant.

http://arstechnica.com/features/2005/05/gui/ last Accessed: 31/10/13

The history of the GUI included the basic information needed to give an introduction for GUI.

Heuristic Evaluation, Usability Evaluation Materials by Darryn Lavery, Gilbert Cockton and Malcolm Atkinson at the Department of Computing Science, University of Glasgow http://www.dcs.gla.ac.uk/asp/materials/HE_1.0/materials.pdf

A self applied form that provides a series of questions to self criticise a GUI and if all of the questions are true then the GUI fulfills what it is designed for in a board context, however I needed to still specify my own objectives and for it to fulfill those.

Cay Horstmann Big Java Forth Edition Page 629-631

The big java book describes the use and application of a Stack and also describes a Queue. This chapter gave a very detailed description, as well as the code to apply the data structure.

Cay Horstmann Big Java Forth Edition Page 652-663

Again the big java book included a chapter on hash tables that allowed me to understand what their function was as well has how to program the different implementations. This was very helpful, as I needed to be able to use a simple hash set to be able to filter the domain.

http://en.wikipedia.org/wiki/Heap_(data_structure)

Describes a heap as a tree structure as well as its application as a priority queue that was used within the program to order the variables.

http://www.javacreed.com/swing-worker-example/

Essentially a tutorial on Swing Workers that was needed when the GUI became unresponsive and ideally there needed to be another thread to handle to process of performing the algorithm and updating a board.

http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html <last accessed 7/11/2013>

Oracles guide to the layout system within swing for the GUI window builder pro, this gives a definition of all the layouts and also includes the code to be able to implement it. This was important to create the board containing JLables. The box layout was used as the main layout.

http://docs.oracle.com/javase/tutorial/uiswing/layout/box.html

Box layout contains a function called glue and it gives an example of this that allows a background to the application of glue that constrains the objects within the view at another object at a minimal distance and doesn't exceed a maximum distance.

http://www.tutorialspoint.com/design_pattern/flyweight_pattern.htm

A tutorial of the flyweight pattern and what it needs to be relevant to a problem, how to implement it, this website includes a detailed example of code that includes steps, this was very interesting to read about as before the program was created meant that it allowed the concept to be included within the design.

2.1.4 Chapter 3

http://en.wikipedia.org/wiki/Tic-tac-toe

When researching the tic tac toe game this website gave both a simplistic and formal definition, but it also describes the search tree involved for each board and the ways to reduce the boards by pruning and the duplicate boards.

Algorithms in a Nutshell, by George T. Heineman, Gary Pollice, and Stanley Selkow pages 217- 223

This book gave a very clear and detailed description of the Alpha Beta Pruning algorithm that was used in conjunction with the mini max algorithm; there was a description of the algorithm and how it will effect to time complexity within the application of the pruning method.

2.1.5 Chapter 4

http://docs.oracle.com/javase/7/docs/api/java/util/Observer.html

This was used as the basic oracle definition and application of an observer from one class to another this source also describes the update method.

http://www.sudokudragon.com/sudokutheory.htm

Provided a very hard to solve board that I used to test the algorithms on harder boards to see what kind of time difference there would be. Also had some interesting notes on complexity.

CS2800: Software Engineering Notes - D. A. Cohen

The presentation gave an outline of the software engineering standards that are expected within the current industry. These are then what I have been taught and followed throughout the process of creating this project.

2.2 Techniques used by human Sudoku solvers

Around the world Sudoku puzzles are solved by millions of people, there are many ways that people can solve it by using their basic logic or more complicated processes that are proven to aid and move a step closer to a desired completed board. Users will often only use the very simple methods and will struggle with anything harder as they will need to use the other methods, as simply scanning will not provide an answer.

2.2.1 Background

How Sudoku became a well known and widely played puzzle game starts with its creation that began in 1783, when Leonhard Euler the Swizz mathematician created “Latin Squares” [17] (which will be discussed later). That is the foundation of the grid that contains a Latin Square that is divided into nine and provides some of the limitations that are the constraints within Sudoku. It became accessible to most people and was published as a

[17] http://www.theguardian.com/media/2005/may/15/pressandpublishing.usnews

puzzle game, first in America and then later within the Times Newspaper where it became a hit due to its simplicity and ability to become addictive.

2.2.2 Latin Squares

As already mentioned, Latin Squares is an n x n grid containing numbers that are only allowed to appear once on each row and column.

Here is a Latin Square of size 6. It is possible to see here that each row and column only has one of each number.

0	**1**	**2**
1	**2**	**0**
2	**0**	**1**

This allows the grid to then be used in the same way as the Sudoku, but with a distinct change of using a full 9x9 grid, with 9 sub-grids that must contain the numbers 1-9. This changes the arrangement a lot. It is not possible for the grid to be a true Latin Square, as it cannot have a normal sequence instead it is further limited, and by this it does make the puzzle significantly harder.

It is also important to mention the 8 Queens program here, as there is a correlation to the Latin Squares and an 8 Queens problem, where in that case, a Queen must be placed in a cell where no other queen is on the same row, column or diagonal line. Here is an example of a Sudoku:

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

As you can see, the grid does not replicate numbers within columns or rows as well as within each individual sub-grid. This is what makes a Sudoku, and these constraints allow the publishers to part fill some of the grid with premade numbers. This is then possible to be completed and must be done in the way it was designed, to allow the correct constraints to be satisfied.

2.3 Techniques used by human Sudoku solvers [18]

To use the puzzles that are published in books or newspapers there needed to be different levels of Sudoku and providing fewer predefined numbers did this. Allowing the users to find the puzzles harder, as conventional methods of solving would often not be enough means that other strategies would need to be used.

Methods used by humans to solve Sudoku's range from simplistic to more complicated, here are some of the techniques:

2.3.1 Using Pencil marks

Not exactly a technique, however it does allow the user to add their own workings out to each cell to make it easier to use. This is when the user puts in temporary values of what the cell could be at that time and will change a often during the solution process.

2.3.2 Crosshatch Scanning/Naked Single[19]

The most simple method, using all the other numbers in the current grid to see if any can be filled due to the limitations of only one number can be placed within that cell. In that case it could be using the restrictions from the column, row and sub-grid, meaning there could only be one number that could go in that cell.

[18] http://www.decabit.com/Sudoku/Techniques
[19] http://www.stolaf.edu/people/hansonr/sudoku/explain.htm

2.3.3 Hidden Single

Much like the previous but uses the constraint that the row, colour and block must contain numbers 1-9. This could be the case, if there were already some numbers defined within a row, column or block and the last must be a number that is not within the set currently. Most human users will use these single methods. [20]

2.3.4 Row & Column Checking [3]

When a number is possible, and there is only one instance of it within a block, as well as the column and row that it is placed, this means that no other block can have a number within it that is the same, on that row. This limitation to the rows and columns can aid in reducing possible number options.

2.3.5 Elimination of Subsets

When a collection of cells have the same candidate numbers (possible numbers) but a pair or more have only a specific candidates that only appear in those cells all other candidates are not considered from these cells due to these cells being the only ones that could possibly contain those numbers, so the other candidates are meaningless and are discounted. [21]

2.3.6 X Wing

Four cells that contain the same candidate, if their positions are within 2 rows and two columns then this allows the X Wing technique to be used, because the 4 cells cross over in terms of rows and columns. Their candidates are then eliminated, as they would be unable to have the same number, as they are in line. [22]

2.3.7 Swordfish

A technique much like X Wing, however it is a bit more advanced, as it uses three rows and columns. This is a way of eliminating large numbers of redundant candidates, if all

[20] http://www.sudoku-solutions.com/solvingHiddenSubsets.php#hiddenSingle
[21] http://www.decabit.com/Sudoku/NakedSubset
[22] http://www.decabit.com/Sudoku/XWing

three rows have multiple cells, that contain the same candidates within the same column then they are eliminated. [23]

2.3.8 XY Chain

When there are a group of cells that are all linked, meaning they are all within a sequence such as:- the first is on the same row as the second and shares a candidate. As well as this, a third cell is within the same block and this also contains a shared candidate with cell two. This creates a chain and can continue. If the chain is created, and the end point of the chain is the same, then the candidate from the endpoints can be removed [24]

2.3.9 Comparison of Methods

These techniques are very useful, however they will not be able to be used every time. When the puzzle is solved, it will depend on what the layout of a Sudoku is and can only be used when they are available, often determined by the pre-set cells due to the complexity of the Sudoku. The harder the puzzle the more likely one of these techniques will need to be used so that it can be completed.

2.4 How Programs Solve Sudoku's

2.4.1 Backtracking (Recursive)

Much like a human, a program can check the basic constraints that are within the Sudoku and check for singles using a version of the crosshatch scanning by looking at the rows and columns and checking. To do this involves checks for the candidates within the constraints for that cell, and if it has none then it moves to the next cell. If a candidate is found it is entered into that cell temporarily until it is confirmed. This then is recursively executed until the grid is full of the numbers and all constraints have been satisfied, at which point the Sudoku is solved.

[23] http://www.angusj.com/sudoku/hints.php#sordfish
[24] http://www.decabit.com/Sudoku/XYChain

2.4.2 Exact Cover

This process is a very effective way to complete a Sudoku puzzle by creating a matrix containing all of the data needed to complete the puzzle. The matrix is broken up into four sets of 81.

- The first set represents one of the cells that are within the Sudoku, so there are 81 and is used to show that a number has been placed within that cell.

- The next set is defined by the 9 rows and the possible numbers inside of them (1-9) this allows each row to be defined by any of the numbers it could possibly be.

- The next set is the same for the 9 columns and the possible numbers within them (1-9). The columns will not be set to the correct number using the possible values.

- The final set are those labelled by the 9 boxes and the possible numbers within them (1-9), as each sub-grid needs to contain one of each number and this process allows all to be tried if needed.

When a number is placed, it is represented in all 4 of the correlating places within the matrix. In this way, each entry can be clearly positioned where it is, with each row having a cell for each possible number and a search tree is used to find the missing numbers. [25]

2.4.3 Brute Force

The brute force algorithm uses the process of going to each cell and assigning it a number – 1, for example and checks the value against the constraints. If it could be true, then that cell is set to one and the process moves onto the next cell, where the same process happens. If the constraints do not permit the use of the number considered for that cell, then the number is increased to 2 and the check is made again. If after all possible numbers have been considered, then the cell is left empty and is set to a priority of 2. This means that it will need to be given the same treatment when the next pass is made. This process is run until the last number is found.

[25] http://www.ams.org/samplings/feature-column/fcarc-kanoodle

2.5 Humans vs. Computers

Sudoku is used as a puzzle game, however humans do struggle with the puzzle, due to its wide range of possibilities and restrictiveness due to the constraints that provide the rules of the game. Humans however, are generally slow at completing these puzzles, especially when compared to the time a computer program takes to complete the same grid.

While the processes used by humans are slow and less effective, there is not much that can be changed because of the limitations on how much we can mentally compute within such a small time, and at the same time. The scanning technique requires the user to look around the grid to find a cell that could be a number, but does potentially waste time, as it is possible that this process might not prove effective in all cases.

For a human to advance and be faster complete harder Sudoku puzzles they need to know the techniques that will be needed to be able to find the possible numbers and be able to place their values in the cells. Techniques that wouldn't be initially thought of by a player would be X Wing; it uses a very straightforward process that makes sense, however when the user is solving the puzzle with a simplistic approach they would not consider this.

Computers on the other hand are much more effective time wise, with their completion of the grid. However, in saying this, there are a variety of algorithms that can be used- backtracking and exact cover being examples and efficient, and would allow a Sudoku to be solved as long as it was a possible grid.

These algorithms are a lot quicker to complete the puzzle, however this is due to the ability to create large matrixes to hold the data, or the extra processing speed to continuously check each value for every cell. These processes are only as effective as the speed that the algorithm can process all of the data that it needs to. In backtracking, this is very little data but has to be replicated many times. With exact cover it is the whole matrix that needs to be computed.

2.6 Related Programs

At the start of the project, research was made into existing Sudoku solvers to give me inspiration and discover what is needed within the program. I found that there are a very large number of programs to choose from, thanks to the popularity of Sudoku. As well as this, there are many methods of selecting a Sudoku grid, and inputting values. The algorithms to solve the puzzle are sometimes just included within a solve button, while others allow a method to be selected.

2.6.1 http://www.solvemysudoku.com/

(See appendix 2) figure 1: Image of Solve My Sudoku website.

The key features within this solver is the Sudoku grid, that allows input by keyboard input, and has a solve button that allows the board to be solved. A very bright orange background surrounds all this. This is the minimum a Sudoku solver needs to work and this is what my project should do within its basic framework.

While the context of the programme is not known, and it may be that this was an early version, there are many disadvantages of this program, the most being the lack of functionality. For example it is not possible for the user to specify which algorithm to use, as well as any way to input boards other than human input. The colour scheme is not attractive, and the brightness of the background distracts from the functionality of the solver.

2.6.2 http://www.sudoku-solutions.com/

(See appendix 2) figure 12: Image of Sudoku-Solutions website.

Unlike the previous solver, this one is further complex, with many options for the user, so that the experience is much more engaging and allows different methods to be used when solving the board. The interface is very clean, with the board clear, and with a tabbed panel where the different methods of solving are. This is a very neat method of containing a lot of options within a small space, and if my program gathers as many methods of solving Sudoku, then this could be an option.

It contains pre-set boards that the user can use and solve as well as edit (using the grid that allows users to edit). There is also a 'load and save' function that would be very beneficial for the user and then means the same board is repeatable. This is then saved into a text file so the user can store it. The boards all have a rated difficulty and even allow the users to choose harder boards to solve, as well as find out their own boards' ratings.

Overall, this Sudoku solver is what I get most of my inspiration from, as it contains everything I would like my project to do and will allow there to be a clear and concise layout. As well as the kind of functionality that is within this program, there is also a large 'how to use' section that is useful but which I would like to integrate straight within my program rather than have as a separate body of text. These considerations are then projected onto my version of my GUI & program design in chapter 4. Designing the program. See GUI Design in chapter 4.4.

2.6.3 http://www.sudokuwiki.org/sudoku.htm

(See appendix 2) figure 12: Image of the Sudoku-Wiki Sudoku website. This program provides a different outlook on the algorithms to solve the Sudoku. It shows a dynamic list that selects the current method of finding the value, allowing the use of the human Sudoku solver styles to be viewed very effectively. These are the methods discussed within chapter 2.3. This was then used as inspiration for my design for the program and this is included within chapter 4 as the design section.

2.7 The Graphical User Interface

The Graphical User Interface is an essential element to a program that's aiming to be user friendly. The user interface will allow the program to show a window or a series of windows, allowing the user to see all the data that can be output, whilst also permitting the user to input the data. This has become a common process in computer technology and is used by all programs to create an accessible interface for all.

The history of User Interfaces for programs (operating systems in particular) has changed radically since Douglas Englebart displayed the first Interface in 1968[26]. The project involved implementing a display of everything a user would require, whilst making the interface instantly (visually) understandable. It also aimed to allow users to contribute their desired inputs, which outlines the key advantage of the GUI.

2.7.1 Displaying Inputs/Outputs

This practice also needs to be followed within the GUI for the Sudoku Solver. It involves finding the simplest way to display the inputs and outputs, as well as transforming the dull and monotonous appearance of data into a visually interesting and accessible Interface.

For this project the user will need a very simple layout involving a function that allows the buttons to be displayed in a simple and ordered way. The Sudoku grid is going to be created in a recursive way that will allow each cell to be created individually, which means they will all be equivalent. However, some of the design questions concern what I'll be using as the user enabled part of the cells, i.e. how can a user put a number into a cell?

The whole structure of the grid and user interface will be made in a recursive way to enable the user to edit the grid. By filling the grid with buttons that can change colour when clicked on by the user will show they are editable. The user could then type in the desired number input by using the keyboard.

Another method is to similarly make the buttons clickable to show an editable state. The user will then be able to use some numbers displayed on the screen to enter into the buttoned cells within the Sudoku grid. This could be a more advantageous method than my previous point because it restricts the user, only allowing them to place limited numbers (1-9) in the cells. By programming the numbers for the solver it will prevent certain errors from occurring, which would be more likely to arise by the user using a keyboard for manual input

[26] http://arstechnica.com/features/2005/05/gui/ last Accessed: 31/10/13

If the button method proves an insufficient strategy then there's a way to allow the user to pick up and drag a frame that represents a number, which they can then drop it into the cell using an event handler. Therefore, by enabling the user to select pre-programmed numbers in the cell will provide a more efficient user-friendly system.

2.7.2 User Friendliness

For a user interface to be effective to the user it needs to provide every aspect a user needs to complete the task the program is designed for. The design process is the key to analysing the user's needs and requirements; the effectiveness of the design to fulfil those requirements is the measurement of usability.

To satisfy the specification allows user interface to satisfy functionality needs. However, to provide a user with the best possible application interface to increase functionality, other examples include the appeal of the application within it's layout and colours.

2.8 Heuristics

Using a resource[27], which represents the process of evaluation of a user interface, these heuristics can be found on Heuristic Evaluation (Nielsen and Molich, 1990; Nielsen, 1994). I will summarise the relevant heuristic and relate them to the GUI of the Sudoku Solver.

2.8.1 Status Of The System

This is to inform the users of the program's inner workings, in terms of the time constraints. For example, if needed there could be a loading bar on a splash screen as well as a progress bar when the user requires the puzzle to be solved.

The system differs from a real life experience. The system was originally a game played with pen and paper, which allowed the user to make notes, and edit their own mistakes. Whilst mistake handling is possible, there are functions that can be

[27] Heuristic Evaluation, Usability Evaluation Materials by Darryn Lavery, Gilbert Cockton and Malcolm Atkinson at the Department of Computing Science, University of Glasgow http://www.dcs.gla.ac.uk/asp/materials/HE_1.0/materials.pdf

implemented that *are* possible in real life. One of these achievable processes could be the filling in of squares.

2.8.2 The User Experience

This is achieved by allowing the user to save and load previous puzzles. This acts as a help function to assist the user in solving the puzzles by providing a few visual steps.

2.8.3 Consistency

This is vital to the process it handles. But mostly, the appearance needs to be constant and clear, where the buttons need to be equally sized (especially the ones within the grid) to maintain a uniformed structure.

2.8.4 Error Prevention

It is important to prevent the program from having errors and to implement safeguards as a precaution in cases they may occur. If the user will input a number higher than 9 into the grid then it will result in an error. However, if the safeguard were implemented into the program, the user would only be able to choose a button between 1-9, which would prevent an error from occurring.

2.8.5 User Relevant Information

Everything the user needs to know should be visually clear, and the trivial or complicated data should be kept from the user. This will enable the user to easily focus on the things they want to. For example, there could be time and date stamps on each saved Sudoku puzzle, so they'll be identifiable for the user. There could also be a smaller visual example of what that puzzle actually looks like for the user's viewing.

2.8.6 Design Accessibility

I aim to make the user experience as simple and enjoyable as possible by preventing the user from experiencing long processes delays whilst using the Sudoku solver, which could easily be made into quick and manageable functions. This can be achieved by allowing user-limiting usage by giving them pre-programmed buttons and inputs to fill the cells in

the Sudoku grid. They could also use the keyboard, as it will allow faster adding of integers or a 'button like' tab to move to the next cell.

2.8.7 The Design

It is vital that only necessary and important information should be displayed to the user, which will prevent them from getting confused and inundated with large quantities of data and functions. For example, a program could be given a specific structure. The display needs to be minimal and uncluttered, which will limit non-vital information and only display it at relevant times.

2.8.8 Error Handling

Error messages can be used to explain to the user what the fault of the program is instead of it crashing, which can allow the user to correct their error, if they caused it.

2.8.9 User Assistance

A help function will be implemented to explain the game and the user interface that can assist a completely new user (with no understanding of the program or the game) on how to work and use it. Using temporarily displayed boxes that contain useful rules and instructions about what the user needs to do, could achieve this. For example, a drop down box could be used to show the game rules and processes. An alternative could be the display of pre-written manual. However these are not the most modern or practical assistance tools, because users often want to get started on the program as soon as possible or expect a visual tutorial.

2.9 Data Structures

Data structures are concepts and processes for handing the data within a system and each provides its own benefits and reasons to be used. Using these structures allows programmers to use them to increase performance within their programs, which could include processes that allow effective searching and sorting. However, each type has its disadvantages.

2.10 Examples Of Data Structures

To explain data structures and what they are I will use the example of an array, which is a basic way to store data. It is a fixed size, therefore, only a specific number of elements can be within it, and the data is then held. However if the size of the array needs to be increased, the array has to be drained of its elements and re-created in the right size. There are other alternatives to using an array, for example, array lists, which are detailed later in my report. The functionality of a list (that helps the array problem) is that it doesn't have a specific size and it adapts to the size of the data as it's put in.

The example above proves that sometimes data structures need to be used in place of others to provide functionality, which otherwise cannot be there. Alternatively, it will provide a much easier way to write the program with the data structure that is best suited. In the case of this array, when the size of it will be changed throughout the program, the array list can be used. This will be treated the same way by using get and set methods:

```
ans = list.get(index);
```

This is instead of an array, which is done by using:

```
ans = array[index];
```

It would cause problems for the user for the program to be without this structure because the old array would have to be replaced and the size altered, whilst also repopulating it. However, this solution allows the program to work without those issues and this is what the data structures are for.

2.10.1 Linked Lists

Another example of a data structure is a linked list, which can be used instead of an array list. It gives similar functionalities that allows the set methods, delete and element functions to be implemented. However, it does not provide a function for getting a specific item without traversing through the list. Using this as a linked list, over an array list, is appropriate, provided it's not used to source a specific element. It's a much faster way to add and remove elements by simple traversing. Therefore, it highlights how

important choosing the correct data structure is and having to know what the program will be doing to decide what is needed.

Throughout my concept programs and main project I will be using many different data structures to allow my program to work effectively and correctly. Therefore, the right structures have to be used and well executed. The concept programs have provided the foundation of what is needed with the final program and each has highlighted one or more structure.

2.10.2 Tree Structures

A tree structure conceptualizes that all data is linked and can be traced via traversing down the tree. Its method is to create by nodes that contain data within the object, which are then stored within the tree. Each node is connected to the next because it stores the next node within the current.

The use of a tree structure is to allow the input data to be easily searched or found, which can be done by adding the nodes that contain the data set and traversing though them. This can reduce search time, therefore each operation requires O(log(n)) time when n is the size of that data set. This data structure also allows efficient removal and insertion allowing the tree to be traversed to add a new node.

Dynamic Trees allow the process for creating a tree during runtime, which will be subject to constant change. It can be added to after the process of creation and this will allow the tree to be made larger. It can also be cut when needed and by using the board states that are no longer required it is possible to trim that branch and dispose of the unnecessary leaf. This highlights the features within the tree of a tree that allow nodes to be added, searched for and removed. These features are valuable to tree functions and this also allows limitations to be placed. For example, when a node is removed, it is known and recognised, and the tree will remake itself to shift the other nodes around the tree, if needed.

The advantage of using a Tree is that the data set is easily entered into it, which (once within) can be sorted and compared to another. It does not matter if the tree is

unbalanced because the dynamic tree allows the tree to be traversed quickly in O(log(n)) time. Therefore, it does not lose its functionality due to the use of the dynamic tree methods.

This process is more flexible in comparison to the arrays because when a new object is added, the tree is sized to the current number of nodes instead of a set number. For example, an array is set to an integer value, and when an object is removed or added a temp array has to be created to add the new values, so the new array is re-created to correct size. However this is built dynamically and will never be smaller or larger than needed.

2.10.3 Stacks

Stacks are an abstract data type, meaning that they are defined with a set number of operations. In this case, pop, push and peek allows them to be categorised as abstract, which is a form of simplifying their concepts, because they can be limited and described with these functions.[28] They are a way of storing elements in an ordered fashion, because there is a strict order to the stack on how elements can be inserted, or taken from the stack. This can be beneficial when a program needs to be able to store elements, but only when dealing with one at a time or when storing that element for later.

Simple commands of push and pop essentially input the element into the array one at a time. It is a 'last in/ first out' data structure, which only allows the top element to be taken from the stack. The 'push' method simply adds the element to the stack and the 'pop' does the reverse where the element is then removed.

The logic in using a stack is due to its limitations, actually helping the functionality of a program to be automated. For example, there is the 'Towers of Hanoi' problem, which uses a stack to push and pop the elements or disks in this case.

[28] Cay Horstmann Big Java Forth Edition Page 629-631

2.10.4 Queues

A Queue is linked and relates very closely to a stack. However, it is created using a linked list and the operations it uses differ slightly. The add operation adds the elements to the tail of the queue and the other operations remain the same, peek and remove. This allows the reverse functionality between a stack and a queue, which neither is able to do without the operation. The stack cannot return its bottom element until that element is next and the queue will not return.

Priority Queues include elements that have an assigned number, which relates to the order the elements can be arranged in. Adding each element with its priority number, gives the order to the data, which is output. It allows the items to be stored in order when they are added to the queue. The queue will then extract the minimum number in the stored elements when the q.remove() method is called.

This will be used to store the data, which will be assigned priority. This can be achieved by using a process which scores the most effective next move, so the board can be scored and stored as a priority queue. This means that only the highest score will be an output, which could provide a quick and easier way than comparing each board.

2.10.5 Heaps

Heaps are structured differently to trees because each sub tree is higher than the root. Yet, there are no restrictions on the value of each node and its position. In the case of a binary heap each parent only has two nodes and in other heaps there are restrictions on the amount of children each parent can have. Heaps are related closely to priority queue and these are defined as the abstract data type of a heap.[29]

A heap is populated by adding a node to the end of the tree, demoting the parents' slots if it is larger than the element to be inserted. This is done to move up the node, so that at the end, the new node will be in a position where both of its children are smaller than its element. However, this example has been putting the smallest element at the root, which can also be the largest element as the root grows.

[29] http://en.wikipedia.org/wiki/Heap_(data_structure)

In order to remove the element at the top of the heap, you use the function that will take away the root node and move the latest node into the root. However, this causes problems as it likely that the heap will not match. It will have to be fixed by resorted it, which is achieved by adding a new element.[30]

An advantage of heaps is their ability to be versatile by using different data types. For instance, it is possible to use an array or an array list to store the elements, using the layers of the heap to be stored in the arrays. This increases its efficiently as the elements are set in their representative locations, which are predefined.

2.10.6 Hash tables

Hash tables are created using the hash function, which creates a code for objects that are different. For example, if there are many elements within a data set that are likely to be reproduced within it, then it is worth using a hash table that uses the hash function to create a unique code. It is important to remember that in some cases there are exceptions, but they are not common. Yet more commonly, the function will be used within the hash table to be a position within an array that is limited to the size a user specifies. If there is a duplicate, then it will be stored within a linked list with the other hash coded elements.[31]

To find an object within the hash table each element has to calculate its hash code and reduced modulo to the size of the table because this provides the location within the table. Each element becomes within the location check if they are equal to the given element. If found then x is within the set. To delete an element the same process is repeated. However, if it is found within the set it's removed.

The advantages of hash tables are their rapid increase in efficiency. This means (within the best case) it takes O(1) time to add, remove and delete the hash table elements. This provides a huge advantage to any program that has a large data set and any set of related data that needs to be found within this.

[30] Cay Horstmann Big Java Forth Edition Page 652-663
[31] Cay Horstmann Big Java Forth Edition Page 652-663

It is possible to use this within the Sudoku solver to identify each board/game state that is stored in the hash table. This is a much faster process to retrieve the boards than using a tree or any other data structure.

These data structures will be a key part within my final program. They will aid with the fundamental aspects of handling the data, whilst storing it in the most effective way possible. They'll also assist in outputting that data in the best way possible and (in some cases) in a system that allows it to be automated within the program, which will enable it to remain consistent and constant. These are implemented in Chapter 4.9.

2.11 Swing Worker [32]

This program will provide a GUI that displays the information required to the user, while an algorithm is computed. This can be done using a swing worker, as it manages the use of multiple threads to perform tasks in parallel (in this case the GUI update), and the computation of solving the Sudoku. Without a swing worker, there would be issues where the program becomes unresponsive while the computation was made. This is due to one thread creating a stream of computation throughout the program that cannot deviate.

Swing worker is an abstract class that allows the programmer to simplify the use of threads so that it is not directly handled, but instead is to just be able to include what needs to be computed within that section. If this were not done, then threads would have to be created and then terminated at the correct times. Swing workers are much more similar and effective for algorithmic computations. This was implemented within the Sudoku Solver and is described in (See Chapter 6.6).

2.12 Layout Managers

Layout managers are used to prevent user interfaces from changing dramatically when the window size is reduced or increased. They are needed so that the placement of a button or object position is not affected when the window is changed. There is a

[32] http://www.javacreed.com/swing-worker-example/

possibility that the button can be lost behind the window frame and a hidden part of a user's view. Another problem could be organising the buttons in a way that allows them to be changed, and my project will be focusing on the grid and group layout.

Swing will automatically use the absolute layout, which means it will allow the object to be set statically, whilst setting the position of the object using integers to specify location. However, this can cause problems when resizing the window later on in the process.

2.13 The General layouts

A brief summary of the general layout I could implement in my Sudoku Solver is as follows.

2.13.1 Boarder Layout

The Boarder layout is used for the simple allocation of components to the areas within the window. It will include five main areas; the title top bar (page start), the 3 main information blocks (that allow the object to be set to the line start), centre and end and the bottom page end with the last bar. This is all achieved by using the following code[33]:

pane.add(button, boarderlayout.PAGE_START);.

2.13.2 Box Layout

The Box layout is used to display buttons in the same axis whilst making their centres or a set position inline. This allows buttons to be in a row or within the same line. The box layout arranges all the components inside in an ascending order and equal size, which is the desired size of the object to fill the space. The following code sets an object to a box layout:

pane.setLayout(new BoxLayout(newPane, BoxLayout.X_AXIS));.

[33] http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html Last accessed 7/11/2013

2.13.3 Flow Layout

The Flow layout puts the components within a line, following one after another. It is used to allow a series of buttons that are tightly organised and follow the same flow. This is achieved by adding a new object to the flow layout.

2.13.4 Group Layout

The group layout can be used to group components (like buttons) together by connecting the content, which involves adding them into groups. This is completed by using different groups for items and generic settings, which affect all of the items in that group. In this example, there is a content pane from a user interface built by window builder. It creates a group that uses the.addGroup function to add items, which group and present it using the .addGap function.

.addGroup(contentPane.createSequentialGroup() //creates a group that follow in a sequence.
.addGap(100) //the size of the gap between components.
.addComponent(btnSolve, GroupLayout.DEFAULT_SIZE, GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)// adds the button and puts it into the group layout using default sizes.

2.13.5 Grid Layout

The grid layout is used in this project as a way of keeping the buttons in a visual grid. This is done by adding components to the grid layouts 2D array, which allows it to display everything in the correct position, including the specification of the grid location. To create the grid, a new instance of the layout is made and then the components can be added to it.

GridLayout sudokuGrid = new GridLayout(0,2);
sudokuGrid.add(button("1"));

This method proved the best way for me to create a Sudoku grid. It involves filling in a 2D array of buttons, which the user can utilize to enter and select the integers used to play the games.

2.13.6 The Features Of Layout Managers

There are some extra features offered by the layout mangers, which allow the user to customize the visual appearance of the program. These methods are used within selected managers:

[34] Glue is used within layouts such as the box layout, allowing the user to add in a glue object between two components. This provides a gap between the components that can be replaced with a flexible link, which is unlike a rigid link that is unable to change. Unlike the glue function that allows the components be kept at a distance, this function allows the layout not to be affected if they are moved to other positions.

There is a function that allows the alignment of the components to be set to enable the layout to be viewed in a different way, which is done by:

```
button.setAlignmentX(Component.LEFT_ALIGNMENT);
```

Layout mangers are more commonly used because they provide more advantages than absolute placements. It serves in the design stages and creating the GUI, but also assists the resizable windows in the program. The layout managers allow the buttons to be dynamically changed if needed so they remain the same size and within relative distances to each other given the boundaries.

Layouts allow provide as much of a visual aid in terms of grouping the components and allowing them to be set quickly and within the correct placements showing them in the order they are added as well as positioning them with the correct glue in between each component.

2.13.7 Advantages and Disadvantages

One of the biggest advantages of layout managers is that they provide a simple template for each window. The templates can be exactly the same on each window and (no matter the size) there will always have the same positioning. This helps to create a consistent and well-presented user interface with minimal effort as the layout manager can handle a lot.

---

[34] http://docs.oracle.com/javase/tutorial/uiswing/layout/box.html

However, there are also problems that occur when using them and these limitations concern what each layout manager can actually allow. There are different managers that can be used for many distinct layouts. Yet the use of one could hinder the Developer, as there might need to be a use to keep a component somewhere that the manger does not permit. This can be resolved with absolute positioning but then the Developer loses the benefits of the layouts resizing capabilities.

2.14 Design Patterns

Design patterns provide the structure for a program, which is done by giving a concept or design a layout that allows it to work correctly. A design pattern does not specify code but is used to simplify a complicated problem by spreading it into defined classes or methods, which are predetermined by the pattern. There are huge varieties of patterns and many different designs for the same problems. This is due to the few limitations that are in place, as it just needs to work for the programmer and allow a fix for a problem, which often helps to make the code more understandable with a set layout the pattern, defines.

Each pattern will define a structure. It will have to be designed, often in conjunction with other classes and objects, whilst also having their inheritance defined to the programmer. Therefore it is made easier by knowing what needs to be made. They are designed in a specific way, so that if they do specify an object or class they will be open to change. They will not need to be set exactly because they are defined through each program being built for a different purpose.

Some of the design patterns that I have researched are linked to what I'll need within the Sudoku program. (See Chapter 4.1) There will be several design patterns incorporated within the program. They will all be for their own individual reasons and playing on their unique strengths.

2.15 Using Different Design Patterns

2.15.1 Model View Controller

There is a user interface, that will contain the buttons, and this will be linked to the rest of the program. It is necessary to have a design pattern that will allow a separation of the user interface and its event listeners, which allow it to be handled independently to the other program functions.

The Model View Controller design pattern is split into three different parts, they are separated to provide the layout of the User Interface to be clearer so that there are not many methods within one class implementing an interface and relaying information from the other program straight to the view.

The view is that interface code that is generated to provide buttons and a view for the Interface, this will also contain listeners to allow button presses to be recognised within the program.

The view is linked to the controller, which takes the information from the view that a button has been pressed and activates the corresponding listener event that is a method within the controller, this allows the distinct separation between the code for the GUI and the code that is implemented within the program. The controller is linked to the view and the model.

The model implements the action that is requested by the button press, the view is not linked to the model instead the controller is the “middle man”. This allows the model to take the inputs from the controller, and then implements the rest of the program to retrieve the outputs that can then be parsed back to the view.

The Sudoku has a view that will contain JLabels with key listeners and event listeners. This amounts to a lot of possible instructions for each button that is within the controller to prevent clutter and allows a simpler way to parse data to and from the user interface to the program to be calculated and returned. In this case, the imported board needs to be solved when the “Solved” button is clicked. The controller will register the

click and run the solved method that will parse the necessary data to the model. The model will run the CSP part of the program and return the solved board.

2.15.2 Flyweight Pattern

The flyweight pattern creates new objects, but ones that differ from each other. Similar objects are not created, as they don't need to be. For example, within a Sudoku grid at the beginning of the game many of the cells are not given a value until the user sets one. This means, at some point, there are only a few objects that will be set with values that differ from others. Due to this change there will only be 10 possible values, null-9, which allows the pattern to use each number as a pathway to create the cells by only storing the position of each cell. This benefits the program because it is more effective at reducing the number of objects created and decreasing the memory needed, which also increases better performance[35].

Flyweight is designed by having a reference to every object that could be considered a flyweight object. For example, in the Sudoku puzzle there are only ten variations of what each cell can be. This means that the flyweight will have the ten different objects stored within a hash map. Therefore the only data needing to be stored is each cell that equals those objects. This will be used when creating the constraints considering the variables and domains.

The Sudoku solver is a puzzle that requires 81 instances of an object. In this case each cell within the grid will have to be created and this will represent the numbers that are input by the user. It is necessary for each of these objects to be duplicated so they can be handled in same manner throughout the whole program.

2.15.3 Composite pattern

Alternatively the Composite design pattern groups' objects and treats them in the same way as one would be treated. This could benefit the handling of the variables as there could be many and a lot with the same values such as 0, as that is their initial state. This

[35] http://www.tutorialspoint.com/design_pattern/flyweight_pattern.htm

allows there to be a group that can change, for example, rows, columns and objects with the same value can be grouped, which allows them to be checked against constraints.

This pattern's process is to put the objects into a tree where they can be added and removed from the group by using simple methods. However, this does limit the functions, yet still allows there to be a link between the objects and their values. With this process comes the ability to put the cells into trees.

2.15.4 Strategy pattern

The Strategy pattern allows the program to use a variety of independent algorithms, which can be applied to any object. It provides a structure that each object can be handled by. The desired methods could be helpful in the project to restrict unnecessary checks taking place. This could reduce the time taken to complete constraints.

The Sudoku solver will be able to use different algorithms to solve the puzzle and this pattern aids the implementation of different algorithms, which are essential. Each constraint could be used in a different method and there can be a runner method that allows the object to only use the constraints it needs, which prevents them from all running coincidently.

Within the Sudoku solver there are many data structures that are used in several ways to It would be beneficial to have a pattern that would allow the separation of the data structure from the logic, which would allow it to be independent.

2.15.5 Singleton pattern

There have been a lot of ways to handle and manipulate an object but there are design patterns that allow easy and effective ways to create objects. It is possible to incorporate a design pattern that creates objects but enable more effective ways to do so.

For example, the singleton pattern creates a class that can create an object every time it is invoked, which allows the process to be very quick and easy. It means that creating an object is simply calling the method. It is easy to then limit the number of objects, as the amount of them depends on how many times the class is invoked.

To link this to the Sudoku program will allow the correct number of cells to be created. Allowing other design patterns to be incorporated into this could be necessary to create the objects, using the flyweight pattern. For example, when it creates its objects it could use a singleton, meaning it's possible to get the benefits from both patterns.

2.15.6 Advantages and Disadvantages

Design patterns give a wide range of functions and assistance throughout a project, from the backend of the program to the GUI. This can really push a programmer to try new patterns to improve their programs. They can be used repeatedly within projects, as each pattern can be used for many different uses and reasons.

However there are problems with design patterns. There are so many, that for each problem, there will be a pattern that is linked and can be used for the situation. The problem here is that you have to find the right pattern, which can be quite labour intensive, as it requires research or previous knowledge of the patterns.

When working within a team each person will have their own experience with different patterns. Within a group there can sometimes be too many ideas for design patterns that could actually harm the programs design instead of help it.

These were implemented within the Design chapter 4.

Chapter 3: Design & Implementation

The design of the project has been constantly changing as the concept programs have highlighted new routes and increased my confidence with the ability I have to program something without being afraid to branch out and try new techniques. To use patterns will be a large part of my designing stage and the research involved has allowed my program to be structured that way.

3.1.1 Programming Structure

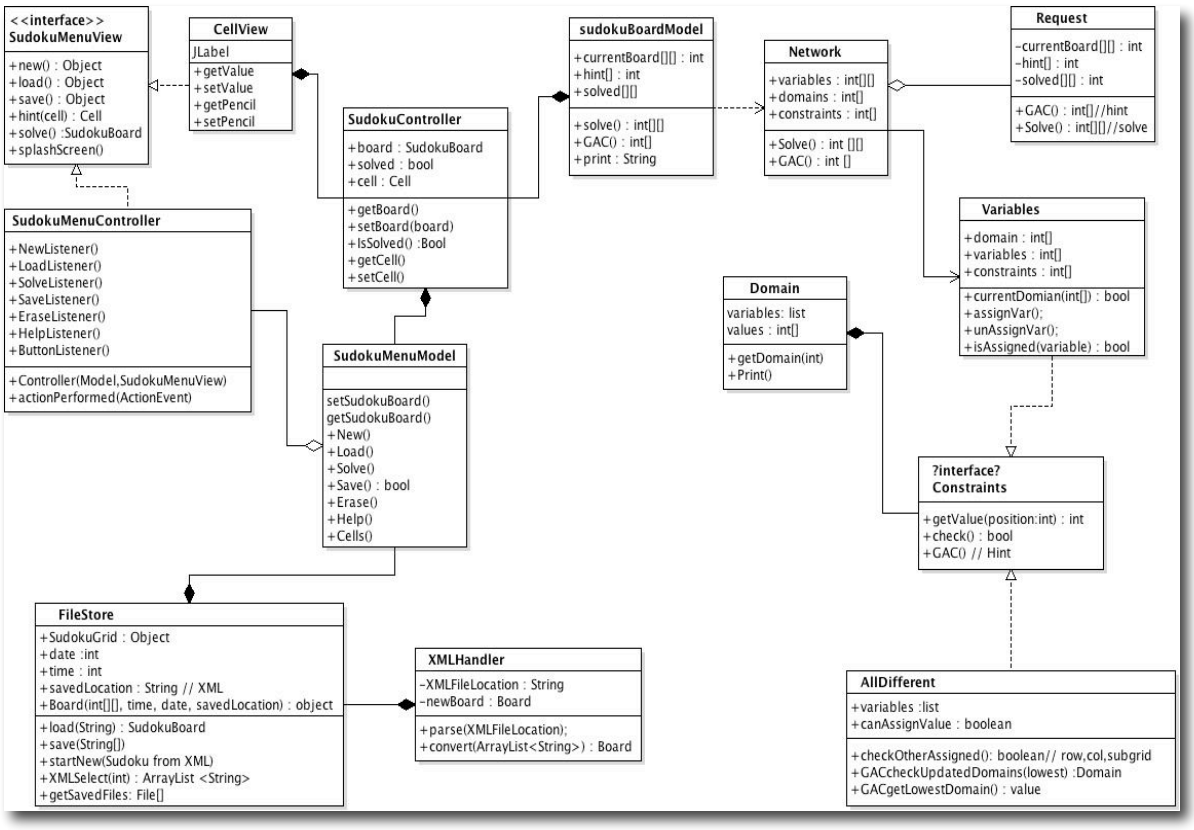
Design patterns are chosen to help with the programming and design of the Sudoku Solver, they each provide useful for the use they are implemented for. Including them within the program will be done at the design stages when a UML diagram will be made. They will help the process of coding by making it understandable, as well as helping the performance within the system by making it as effective as possible.

A model view controller will be used within the view for the menu, this is because of all of the ways the view needs to send a receive data, it would be more effective to allow a standard way of controlling what is being passed to and from the view. In this case the board solving and updates need to be controlled with the buttons on the menu and the boards will be updated, and this pattern will aid in the control. Another MVC will be used (of sorts) for the Sudoku board that will allow the Sudoku cells to be a view, as they will integrate directly within the view's grid and the board itself will allow all of the computations needed, while the Sudoku Board Model retrieves the information requested.

The Strategy design pattern will be included, as it is perfect for creating an interface that will allow the different algorithms to be used through a constraint interface. It is designed to create a network that will be ideal for passing a board to a constraint to be solved. This is done using the Constraint class which each algorithm implements and allows the network calls that determines the correct algorithm to be called to create an instance of that algorithm.

Designing the Sudoku program started with a series of UML diagrams that describe the project and its method of doing so, using the classes that are intended to provide every function that is needed as well as mapping out the strategies and to be able to view what needs to be made within the programming stages.

This is the UML diagram that I have created to layout the program and this has allowed me to view what needs to be programmed in order to have a fully functioning program. There is a description of each class below.



3.2 Description of Classes

3.2.1 <Interface> SudokuMenuView Class

This class will be created using Google's Window Builder Pro that will allow me to quickly implement everything that is needed like buttons and creating auto-code that speeds up the process, it has added functions of adding layout managers. The use of the View is to give the user everything they need to interact with the program, by using actionlisteners

and keyListeners. This could be from the Sudoku Grid where the labels represent the label values. As well as this there are buttons that provide functions that interact with the either the cell values by erasing the values, or loading a new board from an XML are just examples.

3.2.2 SudokuMenuController Class

The controller is used to provide a route to go to implement the listeners actions that are desired, for example if the program wants to display a value in a cell it needs to tell the model that is parsed though the controller and then to the view to be displayed. This process of keeping the listener actions within the controller allows the view to be less cluttered and contain only methods that are needed or provide a function within the view.

3.2.3 SudokuMenuModel Class

This is the link from the program and its inputs and outputs to the MVC that handle the view, this is used to provide the actions to get the values or implement the correct methods if a button is pressed, for example the load button is pressed and a XML date is selected to be loaded, the model passes the date and will call the FileStore class load function.

3.2.4 FileStore Class

File handling is a part of the program that needs to be implemented by the buttons on the view and the forms that allow the user to select the date of boards to be loaded, however this array is used to simply what needs to be taken into account by using methods for each button call.

The load method requires the string of a chosen XML file which was saved, this allows the correct XML to be loaded and also it then does not need to be stored within the program making the program more efficient as it isn't holding many saved files within its own memory.

Save method turns the current board displayed in the Sudoku grid in the view and stores it as a string that can then be passed to the XMLHandler that can export that string of integers into a single XML file.

StartNew loads a random pre-set board that is stored in a folder separate from the saved files, using the XMLHandler to import the file and then display them on the View creates this. getSavedFiles returns an array of the current files that are within a folder that are saved by their date, this can then be presented to the user to be selected when a board is requested to load.

3.2.5 XMLHandler Class

Designed to parse the XML file and use the <Sudoku> set </Sudoku> placement to be able to correctly identify the placement of the required integers, this would allow the XML to be quickly scanned and only the Sudoku headers will need to be identified to retrieve the set. This will then be converted to a string and given back to the FileStore class that has requested it.

The Build method will take an array list containing items that are within the Sudoku, these will then be grouped with a date of saved in the form of a string, this would allow the file name to be the string date and the data within will consist of the saved board that is stored within an xml.

3.2.6 SudokuController Class

This class will create a board object that consists of Cells that contain the integers that put together the board. These integers will be saved within cell objects that are created within the CellView class, and each board will contain 81 of these. The controller is used to set the board cell values, as well as retrieve them.

3.2.7 CellView Class

Links the cells to the view as these are created within the view and are set using the controller. This class allows the text of the labels to be set and the process of setting them. The Cell view observes the model so, when the model is updated, so can the

CellView class, this allows a change whenever it is needed so that the board will also be updated.

3.2.8 SudokuBoardModel Class

Models for the boards implement the methods that can be used on the board, so for example this would allow the algorithms to be launched by using GAC or solve. The board can be printed so that it can be viewed when testing.

3.2.9 Network Class

A virtual network is created to deal with the constraint satisfaction aspect of the program that just needs to be parsed the board to be used and then the applications of the algorithms can be applied, this can be done using the methods that will use the Generalised Arc Consistency to supply a next move. The solve method will provide the answer of the Sudoku Puzzle.

3.2.10 Request Class

Uses the request from the view and specifies which aspects of the network need to be considered.

3.2.11 Variables Class

Uses the variables that are the cell values from the SudokuBoard, these values are entered by a user and saved to a new board that is parsed to the variable class where the values of each cell become the variables that are able to be checked if they are assigned or not using a method, as well as the ability to change the assigned value so that they can all be checked, these are accessed by the Constraints class.

3.2.12 <Interface> Constraints Class

The constraints class is used as an interface between the domain and the variable as there is a given variable that will take a value and the domain is the list of possible values that the variable can have, the constraint takes the value and the domain and makes a constraint from it, meaning what possible values the variable can hold.

The constraint class parses its constraints from the AllDifferent Class, where a list of variables is checked to be true and if it is possible to assign the value that is the only possible value for that variable from the domain.

3.2.13 Domain Class

Takes the given variable and displays the values that it could possibly hold due to other factors, for example within the same row if there are already values, they will not need to be within the domain and will not be included. This is then given to the constraint class to deduce the possible values from the constraints.

3.2.14 AllDifferent Class

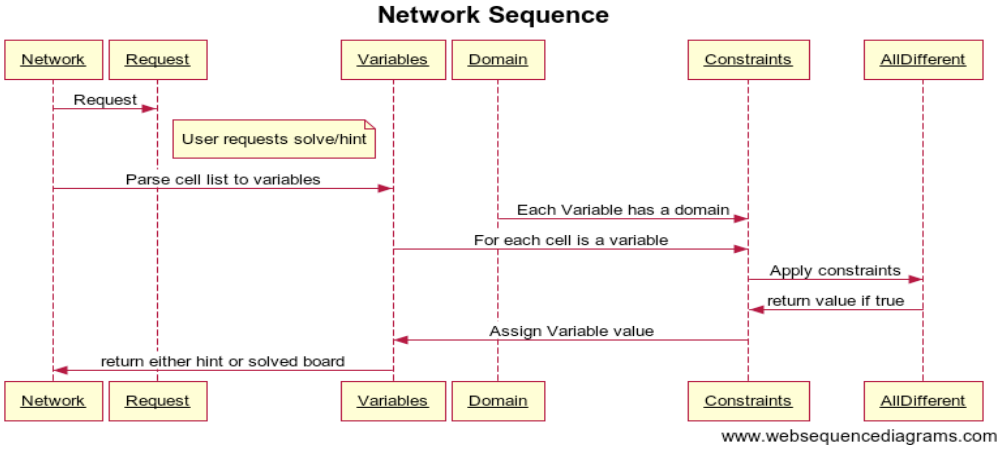
Takes the variables and checks that the values within them are all different, so there are not any duplications within the scope. If there are, the value for the cell is wrong and there has been an error within the board.

3.2.15 The Network

When a request is made the network is traversed, it takes the given board and uses the constraints that are taken from the classes that make up the constraints class, using the domain and variables, this is then compared to the AllDifferent class that applies the constrains, i.e. that all the values within the scope are different values. These values are then returned if true and then assigned at the SudokuBoardModel that connects to the network.

3.3 Sequence Diagram

This sequence diagram provides a visible representation of this.



3.4 Design Research

Analysing other Sudoku Solvers was done in the background theory (Chapter 2.7) where some other Designs and functions were criticised, and I was able to relate what they have and what my program will do in terms of look and functionality. However, they were not the only source of design research. The concept programs that were created in the first term that encouraged me to try new design techniques and new design patterns, were included within the program as I had a good knowledge of how they worked, and how to implement them.

3.4.1 Influences From Other Programs

http://www.solvemysudoku.com (See appendix 2) figure 1: Image of Solve My Sudoku website. This Sudoku solver basically describes what I did not want my program to look like. It only contains the bare essentials, but did benefit me in the importance of a good-looking program that will stand out but not be off-putting for users. This is why I decided to include an image as the program header to include a more striking but simplistic GUI.

http://www.sudoku-solutions.com (See appendix 2) figure 2: Image of Sudoku-Solutions website. This program gave a good indication of what was needed within a program and what kind of styles I could try to implement, for example if I had a lot of

different options with limited room then it would be possible to use a tabbed section. However, in the end, that was not necessary as there was room for everything that was needed, as I was not constrained by a size of the window as long as it was smaller than the monitor.

http://www.sudokuwiki.org/sudoku.htm (See appendix 2) figure 3: Image of Sudoku-Wiki Sudoku Solver. This program was slightly more complex than the previous. It includes a large list of methods to find the next value with some of the methods discussed in the human solving chapter 2.3. A feature that was very helpful was the use of the domains that were shown and updated throughout the implementation of solving the Sudoku, as well as a 'take step' feature that just showed one value being positioned. There is a way to select boards that correspond the algorithms that are needed to complete them, that I thought would be very helpful, as this creates boards that are harder and would not be solved by other methods.

However a problem with this type of Sudoku solver is that it's geared much towards the user in solving a Sudoku instead of using a simple backtracking algorithm, which is not very effective, but this uses the actual methods humans' use. It's effective but it's not necessary within this program due to the processing power available.

3.5 Graphical User Interface Design Concepts

Technical drawings have been made to draft the appearance and feel of the User Interface. I completed these by using different styles and concepts, which would aim to increase the Sudoku solver's functionality and the accessibility of the interface.

Technical Drawing (One Figure 1)

This is a very simple design and was the original, just to give an idea of what kind of inputs and outputs there will be, and allow them to be visually displayed in a simple format. This design used simple buttons that would be linked to event handlers and through them methods to act on the desired function.

The grid for the Sudoku uses a very simple recursive grid layout that puts the buttons into a correct format. This keeps the design very simple and the desired input for the numbers would be via the cell view.

Technical Drawing (Two Figure 2)

This design includes different features, for example, a note box which will be an editable box that allows the user to put notes into it if needed, there are also the 1-9 buttons that allow the user to select a cell and input the number from this sequence.

Technical Drawing (Three Figure 3)

This design implements the drag and drop function, as well as another window, which allows the user to save multiple puzzles. When the load button is pressed it displays a window viewing the saved puzzles, which allows them to be selected and loaded. For added simplicity this can also be achieved within a drop down box, which is illustrated in the technical design drawings.

3.6 Design Implementation

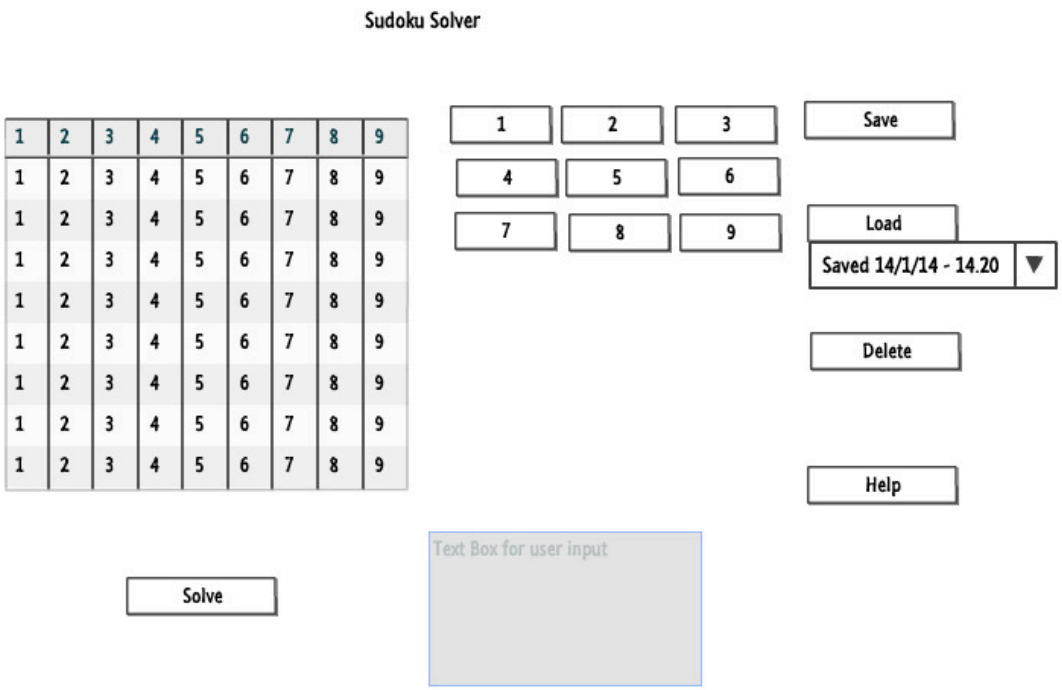
Each of these designs has been replicated within window builder to show and create a useable experience. It saves time by implementing this structure because it can display and test the essential features within the Sudoku solver, as well as assessing their practicality. Therefore the best examples of the features used, which were the essential design elements in the technical plans, can be displayed within the final project.

To create the design elements and provide an easy to build on process I have used a model view controller design pattern with the GUI. This will allow the methods and functions to work without being implemented into the design code, which promotes new functions, as they will be separated and processed independently though the controller. Therefore the project takes what it needs from the program and the computations and will output them.

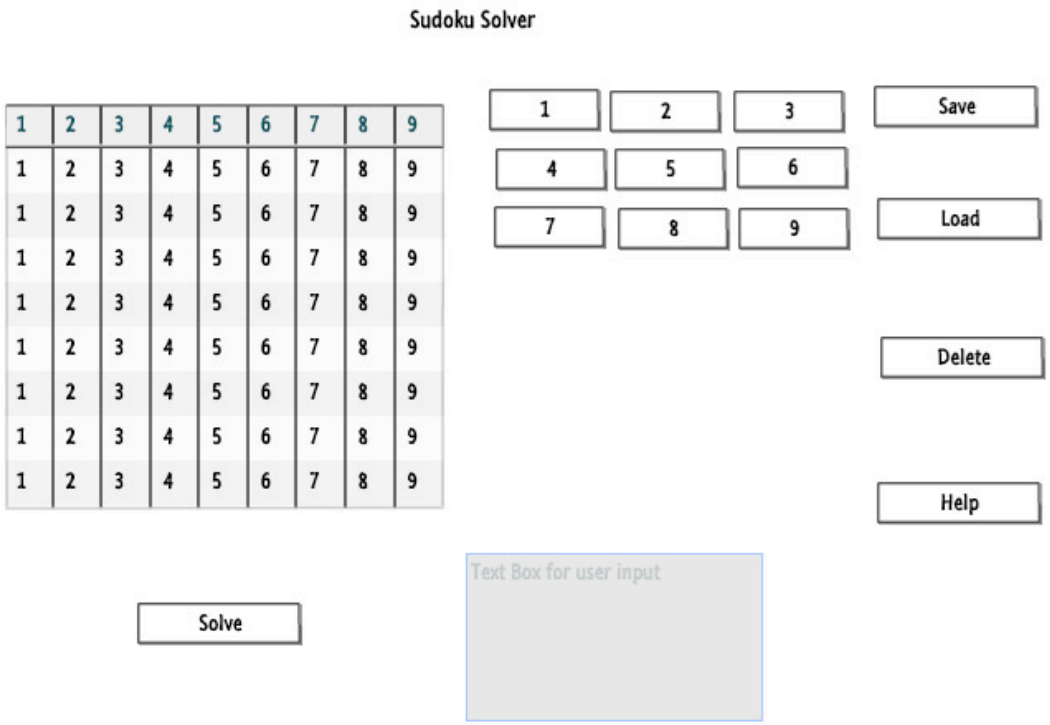
The concepts and designs for the user interface have been developed to create a user-friendly and accessible alternative to the current plain structure of buttons, and large one window interface. By using a task bar as an alternative to buttons, and by keeping the window tight (only displaying the Sudoku grid and keep the options available in a hidden menu), this will make the design cleaner and the Sudoku solver simpler to operate.

3.7 Evaluation Of Designs

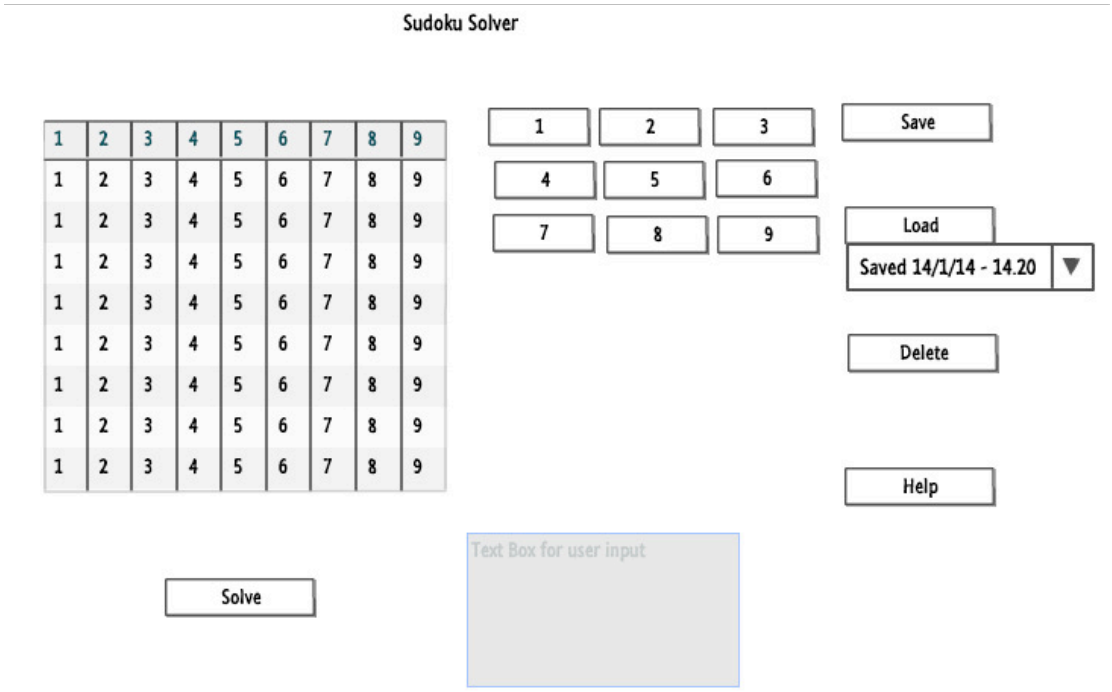
After completing these heuristic trials it has encouraged me to make several changes within the design stages of my final project. There will need to be a few more functions to allow the user to get the best experience. The previous designs have been altered due to the heuristics. Therefore, they allow the designer to take a step back from what they want from a system and the real reason it is made. I will be making these changes to my final design to incorporate them into the final report.



Technical Drawing (One Figure 1) _____*63*



*Technical Drawing (Two Figure 2)*_____*64*



*Technical Drawing (Three Figure 3)*_____*64*

3.8 Using Layout Managers

During my experience of using a layout manager I found that when adding another component into the later stages, you risk changing the rest of the layout. For example, within a button system, I featured 6 buttons that were perfectly arranged with correct spacing in-between. Later, when I added another button and an object next to the previously displayed buttons, the spacing between the buttons changed, which left the display looking dishevelled and untidy. The buttons were shifted much lower than expected. However this was easy to remedy by changing the code and the gaps.

Within the GUI program concept I have only used two of the layout managers, the grid and group layout. However this will not be the case within the final project, as I will be expecting to use a few others to enable the program to look consistent and accessible for users.

3.9 Using Data Structures

I applied some of the data structures covered theory section that discussed within chapter 2.9 to help within the algorithms to store the information and manipulate them. Firstly the List data structure was used extensively throughout the project to hold the Cell Views, variables and domains. This allowed a common type of DS to hold information that meant transferring was easy.

Within the DVO and GAC algorithms I used a queue that had a compareTo method within the variables class that allowed the queue to hold variables but be ordered by the compare method that uses the domain size as a comparison for the most constrained variable.

```
public int compareTo(Object otherObject){
        Variable other = (Variable) otherObject;
        if(!(this.domain.domainSize()==9)){
```

```
if (this.domain.domainSize() < other.domain.domainSize()) return -1;
if(this.domain.domainSize() > other.domain.domainSize()) return 1;
```

To create the domain, all of the values within the scope of a variable need to be compared and any duplicates have to be reduced so that a domain will only provide the possible values and not every value within the list of constraints, so a domain needs to have duplicate values removed by a hash set. This allows the value to be put into the set and return all of the values without the duplicates that can then be removed from the domains. This was done in the allDifferent class when the values are input into the domain.

3.10 Concept Programs

A concept program made was a Tic-Tac-Toe game; the reason for this was to be able to understand how to program a game using a game tree structure, to have a basic knowledge of artificial intelligence and to understand the concept of using search trees and the understanding of how large they can get from a small board like Tic Tac Toe.

3.10.1 Game Tree

Game trees are a system of tree that store game states. In this case it starts with an empty board and each token that is placed can be placed In 1 of 9 squares that are within the 3x3 grid, this allows the game tree to start with an ‘empty’ board at the root and have 9 options with the depth 2, this contains the options where the first players token has been placed in each of the 9 possible squares.

The next depth allows the second player to be placed in of the 8 remaining squares that the first player did not place the token. Creating a board of each option does this placement, where the token could be placed and then they are all scored. To which the best scored board is the next best placement.

The disadvantage of storing all of the possible boards is that a lot of them will be redundant and are not needed due to some of them being the same there are other reasons to filter the other boards, for example getting rid of the redundant boards that are used when a parent has ended due to it resulting in a win or draw and then the boards after do not need to be stored.

3.10.2 Boards

The number of possible boards calculates as 3^9 is the number of boards that can be considered because there are 3 states to each cell (X, O, Empty) and there are 9 cells; this number is then totalled to 19,683 boards.

However, the number of possible games is the number of possible ways to place the tokens within a board, this is calculated as 9! Possible games as there are 362,880. To reduce this number factors have to be taken into place to stop the board production if there is finishing sate, i.e. someone wins.[36]

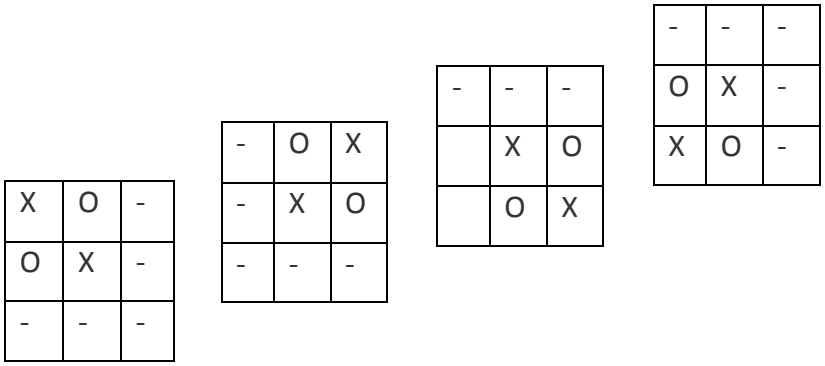
3.10.3 Duplicate Boards

The way to reduce the game tree is to access what is needed and what can be reduced. Obviously cutting out the redundant boards is possible as well as boards that are duplicates of each other, there is a way that a board is the same as there are not many possibilities. The first is to think of each board as a symmetrical cell, where when the values of cells around the board it is possible that another board will be the same as that, and this will cause the duplicated board redundant. The placements of the values within the board can be moved.

These four boards are cases where they are all the same, and when rotated they will give the same case in terms of scoring and anything else. To reduce space a hash

[36] Steve Schaefer (January 2002). "MathRec Solutions (Tic-Tac-Toe)". MathRec.org

table could be used to identify all the boards, and each is given a unique ID number, so that when a board like this what is identical it will only be included once and not stored several times.



3.10.4 Symmetric Boards

A way of reducing boards is by comparing new boards to the already considered boards and the chances they are the same this is symmetric, and an example of this would be if a board only had 3 values, both X's are placed in the top left and top right corners and the O is stored in the centre cell. The board is symmetric, as the board has the same placement on both sides; this means two boards do not need to be stored independently, as they are the same.

Once these factors have been considered, it changes what the numbers of possible games are, as the symmetries and rotations cause the number of games to reduce to 26,830.

3.10.5 Positions

Each position within the board will have a preferred positioning, to be able to simulate the AI of the computer player, as it cannot just add in the top left most cell and that be the best position. It needs to be done in a way where each move is given a preferred position, and following this set of rules does this:[37]

10. Win- Complete a 2/3 line

[37] http://en.wikipedia.org/wiki/Tic-tac-toe

11. Block – If user is going to win then block last value
12. Split – create two rows with a 2/3 line
13. Block Split – block split
14. Centre – enter value in centre cell.
15. Corner- place in opposite corner of user
16. Empty corner – place in corner
17. Empty row – place in middle of the row

These rules allow the program to look at the whole board and decide what is the most effective move to be placed and this way increases the chances of winning.

3.11 Algorithms

To provide a next value to be placed within the board, an algorithm is used to find the right value by checking the current board and returning the best option. There are several ways of doing this and they range in accuracy of the best possible value.

3.11.1 Brute force

This algorithm will check every possible value, by using the processing power of a computer to do so. Every possible value is checked by using the value and compares it to the constraints. If it satisfies, then it's allowed to be placed within that cell. If not, then the next possible value is considered and this is recursively done until the right value is found. This will allow all of the possible boards to be created, and as long as the correct constraints are specified it will allow the algorithm to alter the results, and it will not accept the wrong values.

3.11.2 MiniMax

Used to calculate the best possible move for the computer and the player, it scores each of the moves by using the current player that is taking their turn, and they are given the priority by choosing a cell that is the highest score.

The MiniMax algorithm can be described by a list of instructions:

1. If game is over, return the score of current player
2. Else Get list of new games for every possible move
3. Create list of score
4. For each state add MiniMax result of state to the score list
5. If current player is X, return max score from list
6. If current player is O, return min score from list

This recursive algorithm uses the current player as the perspective for that initial run of the algorithm. It will return the move that is desired to provide the best move to beat the opponent. This however provides an unbeatable program due to tic tac toe being a game that if played properly without mistakes it should always end in draw or win.

To improve the MiniMax algorithm by being able to prevent or stop the opponent from winning, however, it is not within the current algorithm, as it does not consider to block or prevent the other player. This is why the depth is included to adjust the score by the depth, as each level is incremented by one to allow it to adjust the scores within the depths, making it more likely to choose the desired board.

3.11.3 Alpha-Beta Pruning[38]

While MiniMax calculates the player's best move with the opponent's moves within consideration, the MiniMax algorithm is quite naive and does not notice boards that do pose a threat to the current player, and the algorithm will continue to produce tree nodes connected to dead nodes that contain false boards as they do not provide a solution.

AlphaBeta will search for the best move but will stop when there is a state that is deemed to not matter to the rest of the search. This will be more efficient than the MiniMax algorithm due to it greeting the tree that extends further than it needs to, due

[38] Algorithms in a Nutshell, by George T. Heineman, Gary Pollice, and Stanley Selkow pages 217-221

to boards that are redundant. For example, boards that contain a win, and this will allow fewer boards to be created and reduce redundancy.

An example of how this algorithm can dramatically change the amount of processing that needs to be done to consider the game states is that a board of only two values:[39]

X	O	-
-	-	-
-	-	-

MiniMax: explores 8,232 game states

AlphaBeta: explores 47 game states

The checking of these game states actually provides the same answer, that the best possible value is to put X into the middle cell. However this is the best-case example where the values start with the best solutions, are the first to be checked, and this will not always be the case.

3.11.4 How this relates to Sudoku

The algorithms used within this project are creating a search tree that then can be reduced to find the values that can be placed and this can also include the best possible value that can be scored. This relates to Sudoku, which could be solved, using the brute force algorithm discussed above but this can be considering a very large amount of values and considering a larger scale board with a Sudoku this would not be an effective method. However using a search tree that only the possible values were considered would shorten this.

This concept program was designed to further the understanding of search trees and how they become very large from a very small board, it means that when applied to Sudoku a search tree would be very large due to the size of the cells and constraints, as well as the possible values. So to allow the search trees to be acceptable to use it is

[39] Algorithms in a Nutshell, by George T. Heineman, Gary Pollice, and Stanley Selkow pages 223

essential that they are pruned like was used in the alpha beta algorithm that took the values that were not possible and excluded the later nodes that would not have been possible, this will be done within Sudoku by pruning the domains of the variables so that only possible routes are considered.

To be able to prune the domains will allow the CSP of Sudoku to be faster to solve even through a tree of all possible options will not be stored but it will be created and traversed to find the next value that will satisfy the constraints, the program will depend on the ability to prune the very large tree to only use the possible values and that will mean the algorithm is faster.

Chapter 4: **Refactoring**

During the design stages and throughout the implementation of writing the code, there were several changes that occurred that might have deviated from the plan. This was essential to build the program in a correct way or to overcome obstacles that may not have been considered until they actually had to be implemented.

4.1 User interface

The user interface was changed often to allow new ways and improvements to the program, allowing the functionality to be improved by changing the layout of the GUI placing the buttons within a different position and assigning them to different anchors. This allowed the window to be easily changed but to a more compact version, so that It was not a full screen size with wasted space. Instead everything is now closer and more compact to prevent over usage of space.

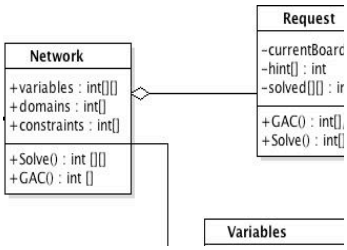
One of the factors that has changed is an extra button within the program as a help function, to allow the user a pop up text box, that contains a small set of instructions of how to play Sudoku as well as a hint on the controls, for example to enter a value into a cell simply click and then key in the desired number. However, this also includes small descriptions for each button that shows what it does.

4.2 Sudoku Grid

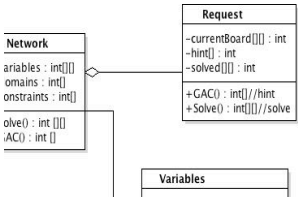
Originally, the program was designed to contain buttons within the Sudoku Grid. O starting the programming, however, I found that it was more beneficial to use JLabels, as they are more versatile with what can be done to them.

Due to using JLabels it was necessary to implement a Key listener that I wanted to have for each cell so that when the user typed the desired number and clicked on a cell, that cell would then be updated as text for the number that was pressed by the user.

4.3 Creating a network

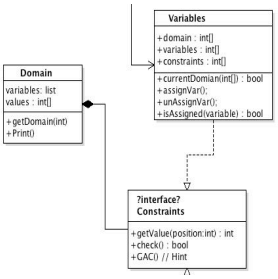


The reasoning to divide the program into different sections by using a network is that one section of the program can handle objects and user interface and the other, which only needs to know the current board values, can be separated. It is practical to involve a Network class that can separated these two functions and allows the program to simply parse the board and then the network will return the requested board.

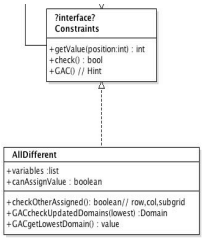


The network involves the classes that interact with the board in terms of constraints and the values assigned to them. Within the network there will be a class that allows the request of the user to include what algorithm needs to be implemented.

The main bulk of the network system is the constraints that are taken place on the board that are then reduced to a set of values assigned to the board as a completed board or board that contains a next possible value placement.



The constraints will be taken from the AllDifferent class that specify that each value has to be different that any of the others.



4.4 Sudoku board Model View Controller

To allow structure, the classes of a board that need to have both a class that contains the cell linked to JSwing objects such as JLabels, it is essential that there is a structure that allows each of the desired board functionalities to work. It was useful to implement a MVC structure that allowed the view to contain the boards' cells that are JLabels and contain the values that are represented within the board.

The board controller class is used as the main part of the board, that is where all the input data taken from the view is collaborated with the board and sent to the model to be parsed to the network. Without the controller, the view and model would not be able to parse information to each other as the controller links the two.

The use of the model in this case is to ready the board to be sent to the network, so that it is the current board that has been taken from the controller and this will then be sent to the network. This also needs to be able to handle the output from the network, as it will return a new board or value, which can then be parsed, to the controller.

4.5 Observer[40]

As within the program there will be a lot of objects that will be changing often, they will need to be observed with an observable method. Within the class where the changes are made, the observed object and the update method is called whenever the object is changed. This allows the class to know when there is a change within the object. In the case of the Sudoku, the observable object will be the cell that contains the values of the Sudoku.

4.6 Swing Worker

Within the program writing often problems occur that were unforeseen within the design stages. For example, I came across an issue with non-responsive JLabels that were

[40] http://docs.oracle.com/javase/7/docs/api/java/util/Observer.html

assigned the desired value, but were not updated to do so within the view. To resolve this, I used a swing worker (researched in the background theory Chapter 2.12).

Concurrency had to be introduced to the user interface, as it was essential that the program was also responsive to user updates even when making a calculation. The main reason needed other threads to allow the program to calculate what the cells values need to be within the board and these needs to be updated when they are changed.

The resolution to this is to use a swing worker that uses a worker thread to allow the calculation process to be made within the program, so that when it returns, the view can be updated to the current board state. This is an example of a swing worker that I used for the solve button.

```
class LoadListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
                SwingWorker<SudokuBoard, Void> worker = new
SwingWorker<SudokuBoard, Void>() {

                        @Override
protected SudokuBoard doInBackground() throws Exception {
                                String chosenDate = m_view.loadMenu();
//load the date from the selected by the user
                SudokuBoard board = m_model.Load(chosenDate);;
//load the board and return the new board.
                  return board ;}
                        @Override
                protected void done() {
                                SudokuBoard board = null;
                                        try {
                                                board = get();
//save value that was returned to a new board
```

```
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    } catch (ExecutionException e) {
                        e.printStackTrace();}
                    board.setBoard(board);
//set that board as the current board within the view using the SudokuBoard class to publish it.
                }
            };
        worker.execute();
// run the SwingWorker
```

The implementation problems that were encountered though using a swing worker within the project are in the Swing Worker Implementation section Chapter 5.6.

4.6.1 Swing worker problems & resolution

The function of a swing worker was discussed in Chapter 2.11, and this was then implemented within the program to allow the GUI to update while the program was running. However, while this did work for some time, I found that it was not effective for my program structure, as it did not need to be updated during runtime. Instead, I changed the structure of the program to allow the current board to be directly passed to the View, and then created an update that way, within the view. This meant that there was a problem when I tried to program a feature that updated the domains and values while the backtracking algorithm was working.

This problem was resolved using a swing worker within the controller that called for the algorithms to be solved, which meant a new thread was created and I was free to deviate from the way the 1 thread had constrained the program.

4.7 Structure changes

The structure was implemented in the way it was designed and this was done to keep the program consistent. However, I found that some structural changes needed to be made

to the benefit of my programming style and the ways that I could get the best functionality. I did not include the observable calls function that was the request, after creating it and deciding it was not needed. It was easier for me to create a string of calls from the listener within the view to the network that just calls the solve method. This process was easier for me to return the board so it could be shown.

No request class, observer removed

4.7.1 Updated UML

Throughout the program development, implementation methods were added that were not foreseen at the design stage, and they are included within this UML that shows the end product. I included more methods within the view and network classes that I had planned, however they were all necessary to give the program the functionality it needed.

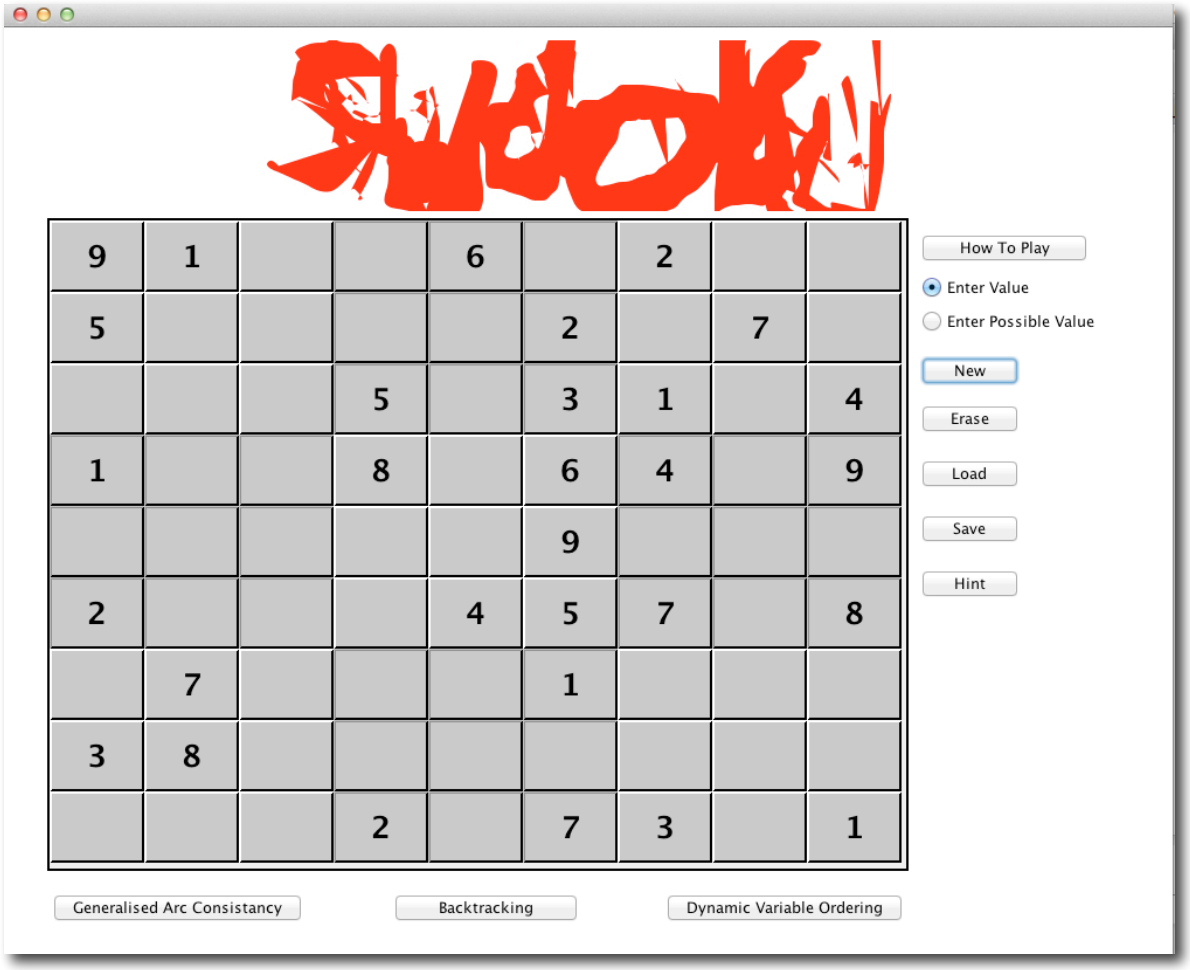
See Appendix 2, figure 4 – Created using Object Aid

Chapter 5: **Evaluation**

5.1 Graphical User Interface

The final graphical user interface shown below in figure 1 is the main menu, containing all of the possible features in user actionable buttons. The Sudoku grid also contains a Sudoku board of the CellView JLabels. This then allows the values to be displayed

Figure 1.

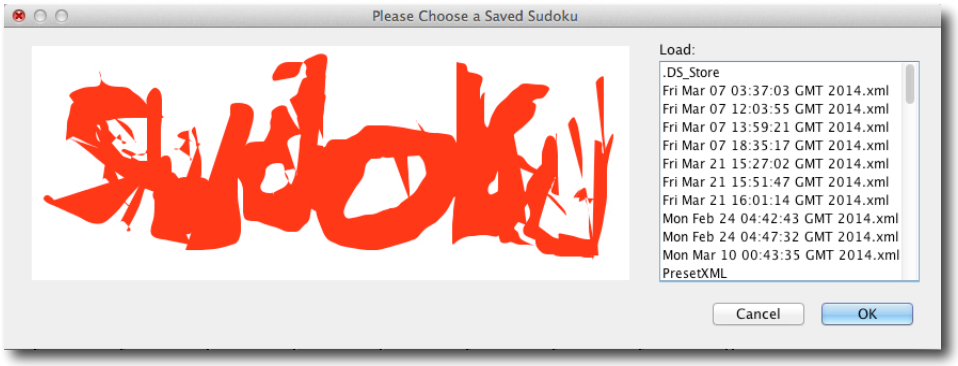


5.1.1 Load Window

The load window used to load a new board to the Sudoku from the menu on the right hand side will show the possible boards that have been saved, within the given file

directory that can be specified. Once loaded, these boards then become the current board, and thus can be solved or edited.

Figure 2.



5.2 Algorithm Results

A test was performed for each algorithm, that used the same input board, and the speed was started before the algorithm was called, and stopped once it was returned back to the method, so this does not include the display time just the computational speed. The board was a pre-set saved board that was stored and used again for each test so it is consistent.

Algorithm	Backtracking	Dynamic Variable Ordering	Generalised Arc Consistency
Test 1 (ms)	231	90	57
Test 2 (ms)	132	43	29
Test 3 (ms)	85	49	32
Test 4 (ms)	117	38	35
Test 5 (ms)	84	33	30
Average	129.8 Milliseconds	50.6 Milliseconds	36.6 Milliseconds

The varying speeds are due to the program only using a timer that computes how long it takes to return the boards. Other programs on the system that are running could affect

this easily. I will run the same experiment with a different board, but with no background tasks running on my machine to see if it will improve the results.

Algorithm	Backtracking	Dynamic Variable Ordering	Generalised Arc Consistency
Test 6 (ms)	608	112	81
Test 7 (ms)	613	50	53
Test 8 (ms)	613	63	41
Test 9 (ms)	612	48	55
Test 10 (ms)	615	60	42
Average	612.2 Milliseconds	66.6 Milliseconds	54.4 Milliseconds

This shows that using the different algorithms provides a faster computation and, more importantly, that Variable ordering and GAC are considerably faster than backtracking.

5.3 Complexity of Algorithms

5.3.1 Backtracking[41]

The concept of backtracking is to try every variable with every possible value until they all are correct, this is a naïve approach and the complexity for a Sudoku is O(possible values ^ unassigned variables) this algorithm is acceptable for most boards, however there are some very difficult boards within a 9x9 board where the length taken was not expected, i.e. one of the hardest found boards[42]

5.3.2 DVO

Dynamic variable ordering does not change the complexity of the backtracking algorithm in its worst case, it is just a practical method of making the most constrained value go first

[41] Time and Space, Designing Efficient Programs. Adrian Johnstone & Elizabeth Scott. Chapter 5,Page 72

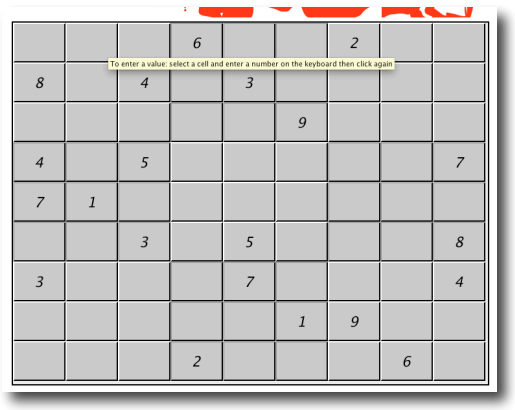
[42] http://www.sudokudragon.com/sudokutheory.htm

and this is more effective than normal backtracking, however given a board where all the values have the same number of constraints it would not be effective.

5.3.3 GAC

Reduces the domains and this means it reduces the number of branches it needs to visit by pruning them, this is also done to all of the constraining values within the scope of the variable. Its worst case complexity is O(constraints within arc * max size of domain^3), this is because the size of the arc is the number of variables within the scope of a current variable, this is multiplied by the maximum size of a domain as in the worst case this could be 9.

5.3.4 The hardest 'found' board results (Please see footnote 42):



Algorithm	Backtracking	Dynamic Variable Ordering	Generalised Arc Consistency
Hard Board	426828 ms	3963 ms	3083 ms

Backtracking performed poorly as this board was specifically designed to be hard for the naïve algorithms. However dynamic variable ordering was still a slow time but a much improvement on the backtracking, this is a very big difference of speed simply by choosing a constrained value. GAC performed the best as the domains are just reduced and it keeps on reducing those values are found.

5.4 Software Engineering Concepts[43]

Software Engineering is the process of creating a program in a predefined systematic way, that includes commonly accepted methods to increase the readability of the code as well as many other aspects that improve the stages of creating, implementing and testing of a program.

Checkstyle is used to as a standard format for code and is used to provide a consistent and commonly understood layout of both code and to filter out bad programming techniques, like spacing the correct amount when a code is formatted, as well as having the correct line and parentheses within conditional statements. It also enforces the use of Javadoc. The program has been set and formatted to comply with all Checkstyle standards.

Javadoc is a commenting system that is used to allow readability for the code that is written and each class is given a Javadoc description that is consistent throughout the project, as each method and variable is described and its meaning justified. This also is invaluable when there are teams working within one program as it allows the other members to be able to understand quickly what a method does, or the reason it is put in place. Javadoc has been included throughout the entire project where it will describe both the code implemented as well as the algorithms used.

To be able to link a team to the correct documentation, an SVN repository can be used to allow a simple way for each member of a team to access and edit the same documents, and they can be updated in real time. This process is also a backup system and a way of version control. My program has been consistently stored within SVN and havs been stored within branches that were designed for different implementations of the algorithms or large structure changes.

[43] CS2800: Software Engineering Notes - D. A. Cohen

5.4.1 Test Driven Development

The TDD technique is used to force developers to thoroughly test the code that is being written, where each part of the code follows a test that was created to fulfill an objective or desired function and the program has to be written to pass that test as well as any other tests that have been created before. This means for every test there is a method that passes that test, when applied to a whole program every aspect of the program is tested so that everything functions as it needs.

The method involved is to first create a test that contains the possible inputs and output, then when this is run it will obviously return false as no code as been written for it to pass, this then has to be passed by any means, meaning to write part of a program. Then the code needs to be refactored to provide cleaner code.

As a result of this every time code is written it is passing a test, has been checked that it is necessary and it is refactored before moving to the next test. This programming style is very in depth and provides a very good programming structure. However I is not every time effective, it can take a very long time to try and get all tests to return true if there have been changes to other parts of the code. But ultimately allows faster recognition of errors and problems within the code as when a test is not satisfied then it is only those outputs that are effect so can be located and resolved. I will use this method within my program.

TDD was implemented from the beginning of the project and allowed tests to be created throughout the program and has continued to allow the design and testing progress to progress in a ordered and correct structure. An issue I did have was when trying to update the board within the view I made a large restructure to the program that meant many of the tests had to be edited to allow them to pass. This was rather frustrating and I did stop testing within this stage as the program was changing and keeping tests to pass became impossible when everything was changing for the boards.

5.5 Critical Analysis Of Program

The Sudoku Solver produced throughout this project that can be found within the SVN archive was designed and programmed to fulfil all of the requirements. This has been completed and the program is working and fully engineered. However there are a few issues with the program that I was not able to change due to multiple reasons:

The cells can be entered into by the user, however this does not work instantly, it requires a click on any cell to activate the key listener so that it then allows the user to input a number on the keyboard that is then the number that can be assigned to a cell, however due to this limitation it requires a click, then a key input then another click on the cell. This does become frustrating and could confuse first time users. Due to the restrictions within swing the only place that consistently works when a label is clicked on to enter the number is the centre and this can be frustrating if a user missed an entry due to a miss click.

I would liked to have implemented more algorithms into the solver, however due to time constraints I only have the 3 defined algorithms of backtracking, DVO and GAC.

For a user to be able to save documents within their own computer they have to manually enter their desired saved file location within the program code. This I would like to change and might be something that is needs to be implemented before a presentation where a pop up box can specify the saved file location.

The program satisfies the objectives as they were followed closely within the design stages and within the programming:

Objective	**Evaluation**
Write a report and program of a Constraint Satisfaction Problem in the form of a Sudoku Solver.	There is an extensive report containing all of the background theory and results of testing, including descriptions of the algorithms involved.

Design the program using research from other existing Sudoku solvers and providing all the desired functions. To aid this design patterns will be included to increase the performance and structure of the program.	The design of the program was extensive and used the research that meant the program was designed in an effective and was based on the design patterns researched.
Create a Graphical User Interface using layout Managers that will provide all the functions and user interactions needed, this will include using layouts to create the Sudoku grid and allow a user input function to solve a Sudoku manually in addition to the algorithm functions.	The GUI was designed and made to include every function that was needed, it provides every function needed and looks simplistic but attractive.
Creating a constraint network to contain the CSP that will need to be solved, this will include variables, domains and constraints that will use data structures to provide effective way to store information and manipulate it.	The network class represents the constraint network that creates an object that is then applied to the constraint network to be solved. This uses many of the researched data structures.
To explore and implement new algorithms that will solve the Sudoku in a faster time and provide other functions that will allow the usability of the program to be increased, these will include:	The solving of the Sudoku uses different algorithms that provide to show the comparison of speeds.
Implementing a simple backtracking algorithm on the Sudoku to solve it, using the basic backtracking algorithm will then provide a foundation for time complexity	The backtracking algorithm is functional and solves the board.

improvements.	
Implement a dynamic variable ordering algorithm that increases the speed of the solution and a 'next best cell' function, using the priority queue data structure.	The dynamic variable ordering solves the board in an improved time to the backtracking algorithm and implements a priority queue.
Implement Generalised Arc Constancy to further increase efficiency to reduce the domains of the variables within a scope.	The generalised arc consistency algorithm reduces the domains of the variables within the constraint network and produces a solved board faster than the DVO algorithm.
As an extension I want to incorporate a save and load function to allow many boards to be saved if they are not finished. This will be done via an XML parser.	The boards are able to be saved to XML files as well as loaded as they are consistent it is possible to load keep the same format, this also includes the pre set boards that are loadable when the new button is activated.

5.6 Evaluation

This program has allowed me to understand constraint problems to a high level that has been interesting and throughout the project I have constantly wanted to know more. Additionally, it has tested my programming skills and report writing. I have achieved almost all technical parts of the program that I set out to achieve, however there were many problems and difficulties throughout the project and I had to problem solve my way out of them.

The main problem I had was how to design and implement a program for the first time. This was the first program that I have ever written that I had to design, so my understanding of the CSP grew throughout the programming meaning the code was restructured often to get another function to work. I had many issues with the GUI and getting the algorithm to display its values on the GUI as the algorithm was running. This required a swing worker, but I had some initial issues with this and it resulted in a program restructure to pass a board through the project back to the view when the algorithm was completed. This created difficulties when I implemented a function to the view, the algorithms values running until it was fixed.

Other problems were encountered when the programming was completed and my understanding of what I currently had to do was unclear, however making mistakes is no problem as the more mistakes made the most I had to fix them and ended up learning more. Again this might have been due to the early design stage where I was naive about what the project might actually need to function correctly.

More of the problems encountered are featured within the refactoring chapter 6.

5.7 Conclusion

Through this project, being able to design and program my own project program and continuously work on it throughout a year, there has been a large boost in my confidence in programming and my ability to be able to progress into a career in programming. The ability to program this in Java has allowed me to further my skills with Java, however I feel as though I missed out on an opportunity to learn a new language extensively, as some are designed for constraint programming, e.g. Haskell.

The ability to program this constraint satisfaction problem provides a practical example of the algorithms needed to solve the problem. A CSP can be solved programmatically by creating constraints and domains within each variable and providing the constraints return true. In the case of Sudoku it is the all-different constraint if satisfied the puzzle is solved. The algorithms that use the constraint network to perform a

search that will be pruned, meaning it will not consider the redundant values and improve the time complexity to solve the puzzle.

Overall, the project went well and I have created a constraint satisfaction problem in the form of a Sudoku solver program that performs the tasks that it was designed for and this has allowed me to create the correct documentation and perform tests on algorithms used to solve the CSP. There have been some of the more complex theoretical aspects that I have struggled with initially, particularly the complexity chapter that I had little practical knowledge about, but when the implementation of creating and programming the Sudoku Solver I grew to understand the time complexity of the algorithms.

Chapter 6: **Professional Issues**

6.1 Software Piracy

The use of the Internet to fulfil both educational and entertainment needs has allowed a huge collection of programs that are easily downloadable from websites without the need to pay or even question their legitimacy. This unfortunately is now how programs are downloaded, due to the large amount of freeware programs available many customers are reluctant to pay for a product.

The programs for Sudoku solvers are all over the Internet, most of which are websites that are purpose built within the page. However, the others are a mixture of small programs developed by students or people that are programming for fun, and allow their software to be used freely. Some companies provide the same or more complex programs and often require payment for the product

To download a program that is required is simple, and this can be done using a search engine that is designed to look at what words have been put in place within the website, i.e. the program description, to display the product that the search engine thinks best matches the search. This is done by using key words within the site and is called search engine optimisation.

6.1.1 Threats

Searches do not provide any safeguards in terms of the sites visited and it is possible that there are harmful programs within a site that can be found in a search.. When finding a program, for example a Sudoku Solver it is possible to download many solvers from the internet that are freeware, however these all could possibly pose a threat to the computer running it.

6.1.2 Illegitimate software

The programs that can be downloaded for free can sometimes be cracked versions of other developers work, thus infringing copyright and committing a crime, there are

many cases where program code is edited to be able to be used for free. However even when a program is legitimate, but edited, it does not mean the program is safe to run. Due to the program code changing it would be very simple to hide a virus within a program that is presumed legitimate.

Software piracy has been a massive issue within the games and entertainment industry as a whole, and there are very few things that a dedicated and skilled team would not be able to copy or at least bypass in systems to allow added functionality (for example with mobile phones that can be jail broken to allow 3rd party software on to the device that has not been verified by the manufacturer).

When free software is introduced to the Internet by the developer it is often posted to many sites to allow the product to be as widely accessible as possible. However, while some sites do allow downloads of the product, it needs to be verified, as it would be possible to create an edited harmful version and publish it to one of the download sites (and no one would question its legitimacy).

6.2 Prevention

When a file is downloaded there are safeguards protecting the computer, however when the user wants to open a file made by an unknown developer then, it will tell the user so and allow them to choose if the program should be run. If a user has downloaded a program often, they will just want to use it, so there is a possibility they will allow the program to run.

To prevent this taking place, companies promote the use of their own products and often enforce this by providing updates within their software that fix some issues, i.e. updating an IPhone that is jail broken will have to be done again, once the update is completed. The same is true with programs that are not from the original developer. They can have major flaws in security or functionality that need to be patched but are unable to do so.

Another prevention method is to use antivirus software, however this is not effective all the time, yet it is the best way to prevent most programs that contain a virus from being introduced to a computer's files. This is possible by allowing the antivirus access to the program file, where it will analyse the code within and look for hidden files that might contain harmful data. This data is then compared to the database of already know viruses and will warn the user of the impending threat.

Bibliography

Book Resources

Professor David Cohen. The Complexity of the Constraint Satisfaction Problem. ISG-CS Mini-Conference Slides

http://www.rhul.ac.uk/computerscience/documents/pdf/csisgmini-conf/dac.pdf

Discrete Mathematics & its Applications, 6th Edition. by Kenneth H. Rosen. Page 836

A filtering algorithm for constraints of difference in CSP's by Jean-Charles REGIN Pages 362 -367

P, NP, and NP-Completeness by Oded Goldreich pages 151-154

Time and Space, Designing Efficient Programs. Adrian Johnstone & Elizabeth Scott. Chapter 5,Page 68 – 75

Algorithms and Complexity 2, CS2870 By Gregory Gutin February 16, 2013. Page 93

Graph Algorithms and NP-Completeness. Kurt Mehlhorn. Page 171- 211

Heuristic Evaluation, Usability Evaluation Materials by Darryn Lavery, Gilbert Cockton and Malcolm Atkinson at the Department of Computing Science, University of Glasgow

Cay Horstmann Big Java Forth Edition Page 629-631

Cay Horstmann Big Java Forth Edition Page 652-663

Cay Horstmann Big Java Forth Edition Page 652-663

Algorithms in a Nutshell, by George T. Heineman, Gary Pollice, and Stanley Selkow pages 217- 221

Algorithms in a Nutshell, by George T. Heineman, Gary Pollice, and Stanley Selkow pages 223

Web Resources

http://en.wikipedia.org/wiki/Constraint_satisfaction_problem

http://arstechnica.com/features/2005/05/gui/ last Accessed: 31/10/13

http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html Last accessed 7/11/2013

http://technet.microsoft.com/en-us/library/aa214775(v=sql.80).aspx

http://www.tutorialspoint.com/design_pattern/flyweight_pattern.htm

http://www.cs.ubc.ca/~kevinlb/teaching/cs322%20-%202009-10/Lectures/CSP3.pdf

http://www.cis.upenn.edu/~matuszek/cit594-2012/Pages/backtracking.html

http://artint.info/html/ArtInt_79.html

www.ics.uci.edu/~dechter/courses/ics-275a/spring.../chapter5-09.ppt slides 26-31

http://ktiml.mff.cuni.cz/~bartak/constraints/ordering.html

http://docs.oracle.com/javase/7/docs/api/java/util/Observer.html

http://www.dcs.gla.ac.uk/~pat/cp4/papers/95_14.pdf

http://www.cosc.canterbury.ac.nz/csfieldguide/student/Complexity%20and%20tractability.html

http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/SIGAL87-2.pdf

http://www.math.cornell.edu/~mec/Summer2009/Mahmood/More.html

http://www.theguardian.com/media/2005/may/15/pressandpublishing.usnews

http://www.decabit.com/Sudoku/Techniques

http://www.stolaf.edu/people/hansonr/sudoku/explain.htm

http://www.sudoku-solutions.com/solvingHiddenSubsets.php#hiddenSingle

http://www.decabit.com/Sudoku/NakedSubset

http://www.decabit.com/Sudoku/XWing

http://www.angusj.com/sudoku/hints.php#sordfish

http://www.decabit.com/Sudoku/XYChain

http://www.javacreed.com/swing-worker-example/

http://docs.oracle.com/javase/tutorial/uiswing/layout/box.html

http://en.wikipedia.org/wiki/Tic-tac-toe

http://en.wikipedia.org/wiki/Heap_(data_structure)

http://www.dcs.gla.ac.uk/asp/materials/HE_1.0/materials.pdf

http://www.ams.org/samplings/feature-column/fcarc-kanoodle

http://www.cosc.canterbury.ac.nz/csfieldguide/student/Complexity%20and%20tractability.html

Complexity and Completeness of Finding Another Solution and Its Application to Puzzles by Takayuki YATO and Takahiro SETA Theorem 4.6 The problem of partial Latin square completion is ASP-complete. www-imai.is.s.u-tokyo.ac.jp/~yato/data2/SIGAL87-2.pdf

http://www.cs.ox.ac.uk/people/paul.goldberg/FCS/sudoku.html

http://www.sudokudragon.com/sudokutheory.htm

Multi-Media Web Resources

http://www.youtube.com/watch?v=Ge-XiX_tP4c-Doug Fisher's video on Generalized Arc Consistency

Appendix 1

My Diary, Plan and Time Scale

Date	Deliverable	Kept To
Term 1		
WEEK 1 (23-29/9/13)		
10/9/13	Start project plan	
WEEK 2 (30-6/10/13)		
3/10/13	Write specification And layout reports	3/10/13 Started
4/10/13 2pm	Finish Project Plan	4/10/13 Finished
5/10/13	Detailed Specification Written	5/10/13 Finished
5/10/13	Design and plan the GUI	5/10/13 Started Sketches were made and so were computer built ones
WEEK 3 (7-13/10/13)		
8/10/13	Program 1: Simple colourful GUI with an active button	8/10/13 Started 3 GUI's built within the SVN
8/10/13	Report 1: User Interface design for a Sudoku solver	10/10/13 Started
WEEK 4 (14-20/10/13)		
17/10/13	Finish Program 1	17/10/13 Finished
17/10/13	Finish Report 1	17/10/13 Finished
18/10/13	Report 2: The use of Layout Managers for resizable GUI applications Solvers.	21/10/13 Started
17/10/13	Program 2: data structures including, for instance, dynamic trees and priority queue, populated with large random data sets.	22/10/13 Started
WEEK 5 (21-27/10/13)		
27/10/13	Finish Program 2	29/10/13 Finished

27/10/13	Finish Report 2	29/10/13 Finished
	If feeling up to date start main project design	(Few days behind did not)
WEEK 6 (28-3/11/13)		
28/10/13	Program 3: Game tree for Tic-Tac-Toe	29/10/13 Started
28/10/13	Reports 3: Design Patterns.	1/11/13 Started
WEEK 7 (4-10/11/13)		
10/11/13	Finish Program 3	1/12/13 Finished, had problems implementing the algorithm.
10/11/13	Finish Report 3	20/11/13 Finished
WEEK 8 (11-17/11/13)		
11/11/13	Program 4: Eight Queens using Backtracking	11/11/13 Started
13/11/13	Reports 4: Constraint Satisfaction, particularly consistency techniques.	21/11/13 Started
WEEK 9 (18-24/12/13)		
24/11/13	Finish Program 4	1/12/13
24/11/13	Finish Report 4	1/12/13
WEEK 10 (25-1/12/13)		
25/11/13	Report: Complexity, NP hardness and the big O notation	25/11/13 Started
25/11/13	Report: Techniques used by human Sudoku	Have Not Completed
25/11/13	Catch-up Week finish needed	
25/11/13	Start final project if possible	Have Not Done
1/12/13	Finalise reports & programs	1/12/13 Completed
WEEK 11 (2-8/12/13)		
4/12/13 2pm	Term 1 Reports deadline	
7/12/13	Prep for review	

WEEK 12 (9-13/12/13)		
9-13/12/13	December Review Viva	Completed Review
Holiday	Final project coding and design	Design made and the design report is started
	Outline of report	Outline made
Term 2	13/1/14	
14/1/14	Continue final program design	Program starting to form and is making good progress, GUI and UML diagrams are made
14/1/14	High-Level Design	Completed
22/1/14	Final report draft start	The report is much like the handed in on the interim, as I did so much in the first term and concentrated on the reports. This was quite extensive already.
1/2/14	First Prototype Program	Program is underway, slightly delayed by a week, basic functionality, and problems displaying the update boards.
22/2/14	Draft Report Finish	Draft report handed in and feedback returned that will improve on the current report.
10/3/14	Finish program coding Final Deliverables 1. The program must have a full	1. Has an object-orientated design. 2. Used engineering

	object-oriented design 2. A full implementation life cycle using modern software engineering principles 3. The program will have a splash screen and two other user interaction screens. 4. The program will have a Graphical User Interface that can be used to generate, load, save, solve and help with Sudoku solving. 5. The report will describe the software engineering process involved in generating your software and describe interesting programming techniques and data structures used (or able to be used) on the project.	principles. 3. Splash screen made, needs to be correctly working, main function screen and the loading menu. 4. Completed 5.Almost completed, just need to be restructured and corrected.
12/3/14	Suggested Extensions Save results of time to database to have a leader board (Store puzzles as xml) allow random Sudoku puzzles playable by user Solve puzzles user puts in solve in different ways using different algorithms	Extension Completed
13/3/14	Review notes from Algorithms and complexity for final reports	Notes reviewed and the book, P,NP, and NP-Completeness by Oded Goldreich to improve the relevant section of the report that I struggled to understand.

15/3/14	Testing of code	Code tested and refined tests.
20/3/14	Finish report and proof read	Proofing of report is done
20/3/14	Finalise code and create working file	Code finished and check styled ready to add to a tag and put everything within the branch.
22/3/14	Specification of Testing Procedures	Testing of the program needs to be completed and all bugs need to be found.
26/3/14 2PM	Final Report	Handed In
26/3/14 2PM	Final Project Program	Handed In
27-28/3/14	Project Demo Day	This has been moved into the exam period.

Appendix 2

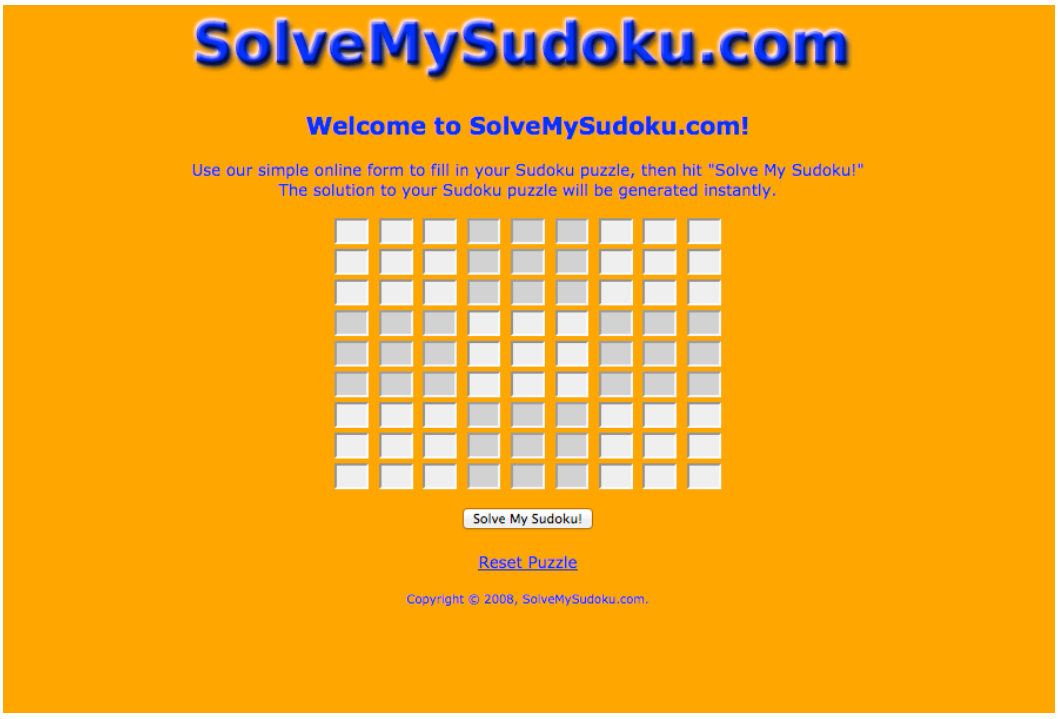


Figure 3.

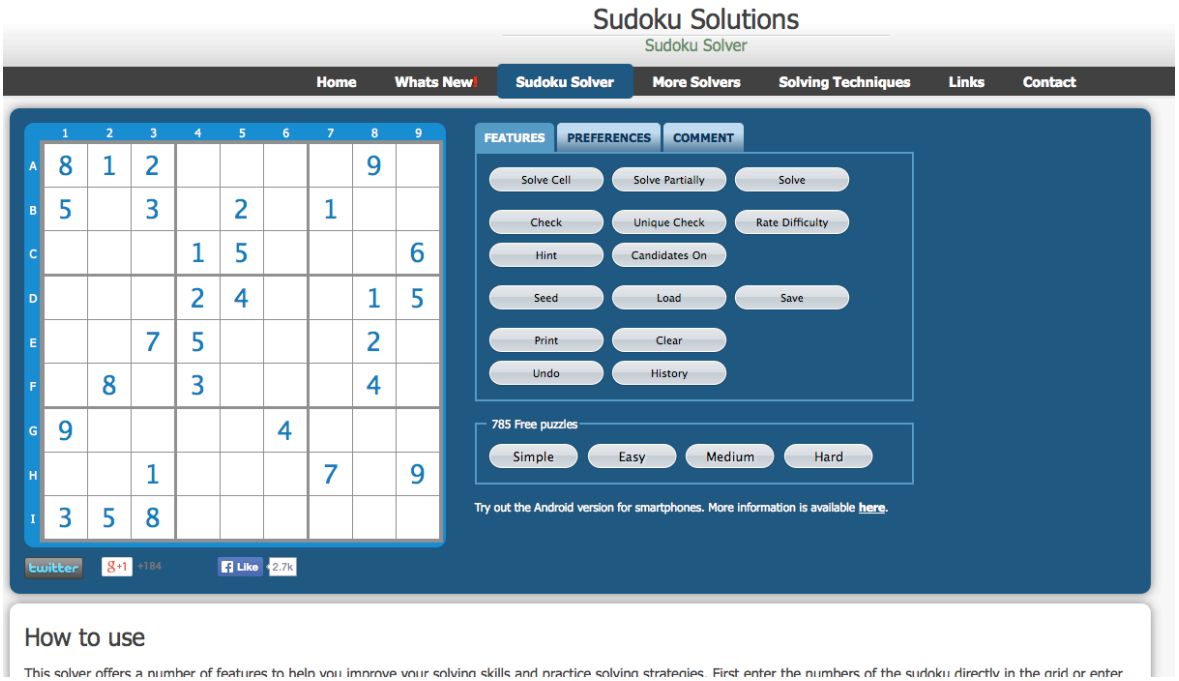


Figure 4.

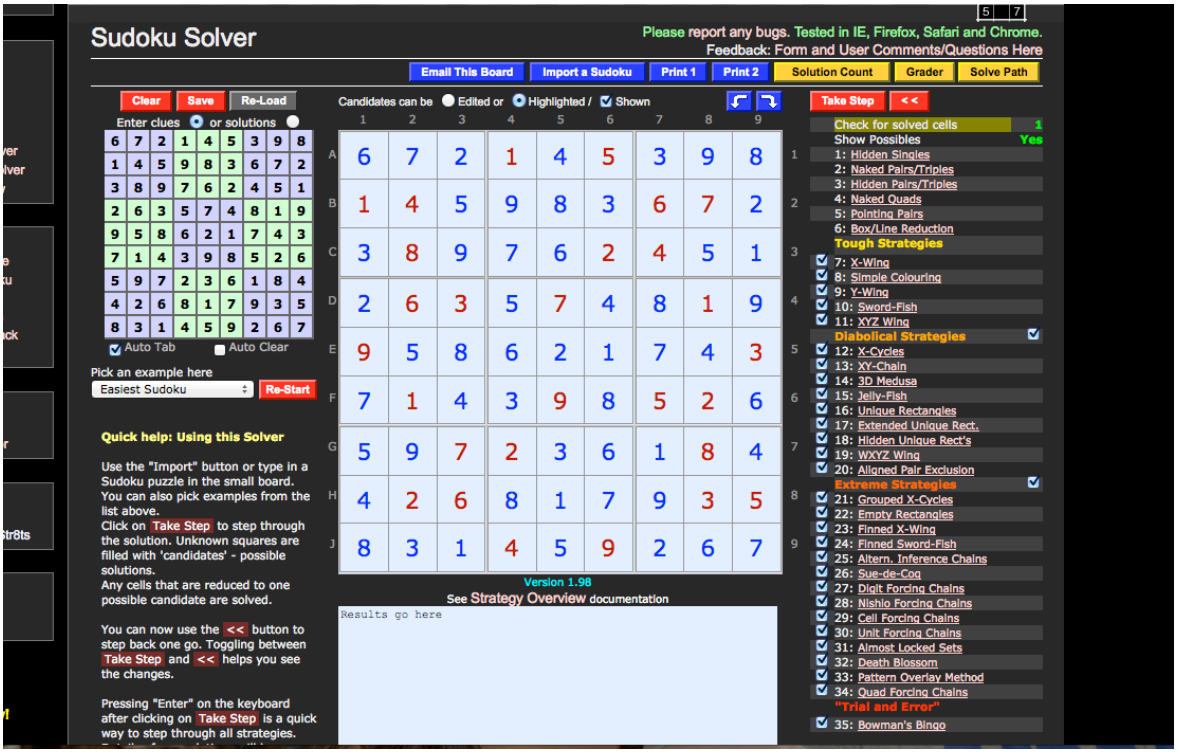


Figure 5.

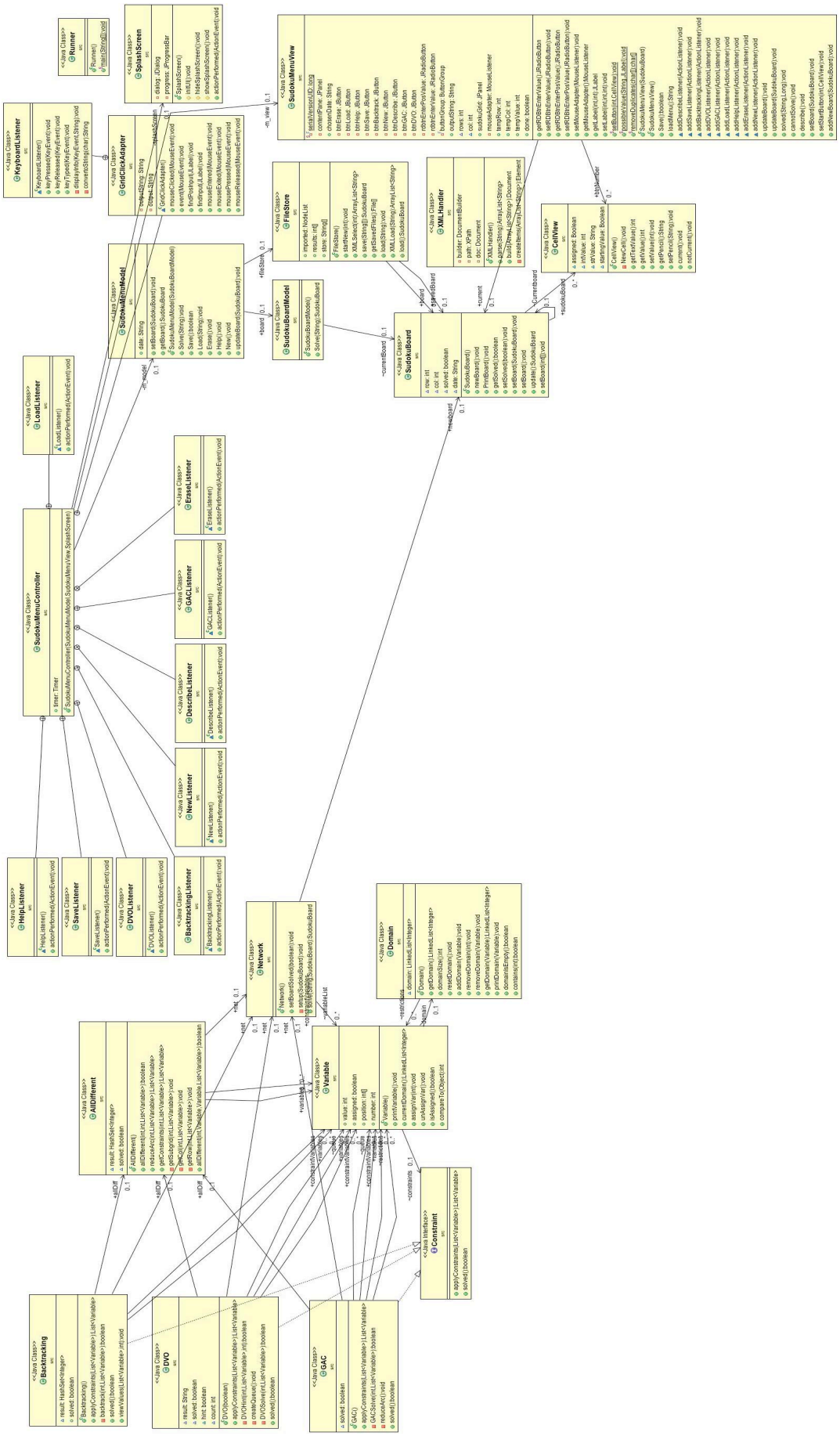


Figure 6.