Interim Report

Solving Sudoku with Artificial Intelligence

December 7, 2016

Submitted by:
Student ID: 4230462
Caitlin O'Sullivan
School of Computer Science
psyceo@nottingham.ac.uk

Supervised by:
Robert John
Professor of Operational Research
School of Computer Science
Robert.John@nottingham.ac.uk

Contents

1 A Brief History **3**

2 Introduction **3**

3 Solving Methods **4**
3.1 Summery of Problem 4
3.2 Brute Force 4
3.3 Refined Brute Force 5
3.4 Backtracking 5
3.5 Dancing Links 6

4 Project Specification **7**
4.1 Functional Requirements 7
4.2 Non-Functional Requirements 8

5 Design **9**

6 Implementation **9**
6.1 Algorithms 9
6.2 Image Recognition 10
6.3 User Interface 10
6.4 Testing 10

7 Progress **11**
7.1 Project Management 11
7.2 Contributions and Reflection 12

1 A Brief History

Sudoku is a puzzle that is relatively modern in origins, first appearing in its modern form in Dell Magazine in 1979.[2] It was moderately popular and Dell Magazine continued to publish Sudoku puzzles (called Number Place in its publications) among other puzzles over the following years. It was not until the puzzle made its way over to Japan where it adopted the name Sudoku, meaning Single Number. It was there that the puzzle really took off, fulfilling the role similar to that of the crossword puzzle for a language where the alphabet is ill suited to crosswords. Due to the massive success of Sudoku in Japan it was brought back over to America in 2005 and since has been hugely successful. Dell Magazine continues to release the puzzles on its web page [3], now under the name Sudoku.

2 Introduction

Sudoku consists of a board that is 9 squares by 9 squares. Within this, the numbers 1-9 have to be arranged such that there is only one of each number in every row, column and 3x3 box. Typically Sudoku puzzles are presented to be solved when partially filled in. The minimum number of clues within the Sudoku grid is typically assumed to be 17 clues[1], however it has not been proved that a Sudoku with fewer clues cannot have a unique solution. All valid Sudoku must have only one unique solution.

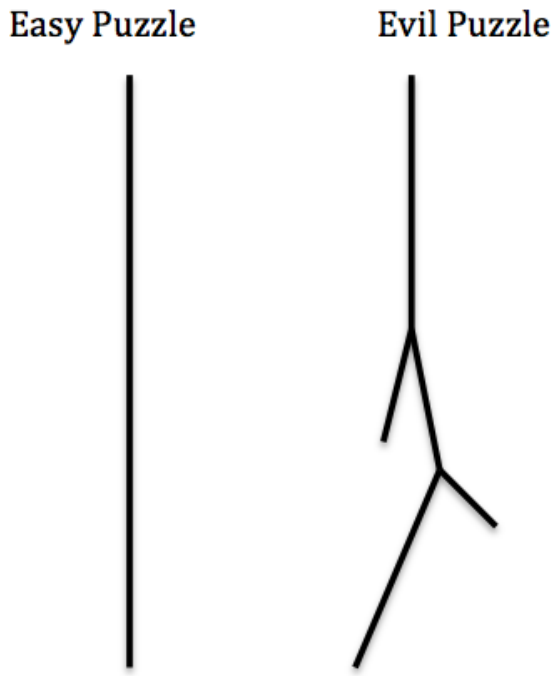

fig:Sudoku Branching
Figure 1: A diagram to show the different branching that takes place on an easy Sudoku vs a hard Sudoku

An easy Sudoku often has very little need for complex solving strategies, as most of the numbers can be filled either because they are the only solution in their row/column/square or because the solution can easily be spotted through some other means. A hard Sudoku often involves very complex moves and thinking ahead in order to solve the puzzle. The hardest Sudokus are those which have branching, where there are no clues but the solver will be forced to make a choice of two or three numbers to place in a square, where one number will lead to the solution and the other number will lead to the puzzle running itself into a dead end. These hard Sudoku are the most difficult for humans to solve as it involves thinking through the next several moves after the branch until they realize that one choice is better than the other. The diagram shows that for an easy Sudoku at each step of the solution there is only one possibility for the solution from start to finish. However for a difficult Sudoku there can be multiple places where the next step is not obvious and the solver has to make a choice between two or more numbers. As there is only one

possible solution to a valid Sudoku then there must be some sort of backtracking in place to allow the solver to try one solution and discard it if it is not a viable solution.

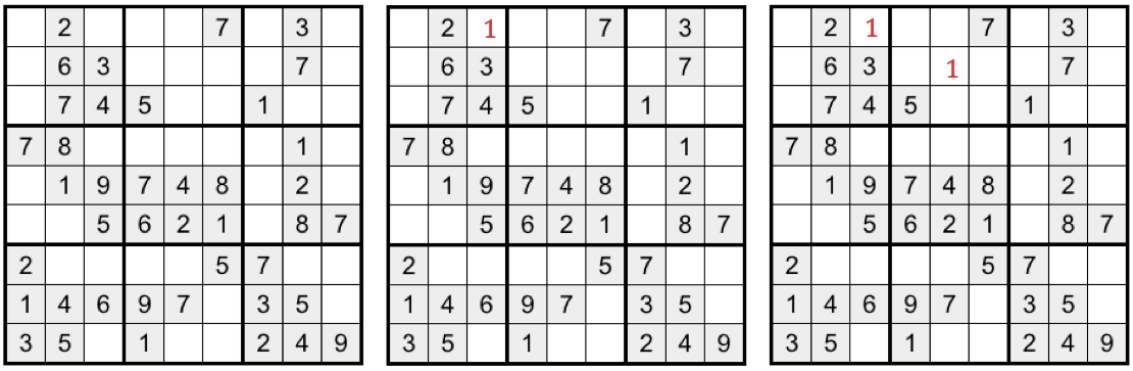

fig:Sudoku

Figure 2: On an easy level Sudoku each step necessarily leads to another possibility being narrowed down

Humans and computers typically solve Sudoku puzzles in very different ways. While a human solver may be limited to the most logical next step, for example on the diagram when a '1' is placed in the first box, it narrows down all the possibilities in the next box until there is only one place that the '1' can go in that box as well. A computer is not limited to making these logical jumps, which can be very expensive to evaluate, as for the computer it first requires searching the whole board for squares that only have one possibility, then placing a single value in them, then re-evaluating the whole board for places where the possibilities have been narrowed down. Rather than doing this expensive process the computer can instead do the relatively cheap process of generating a full solution and testing if it fits. This is a brute force solution. Compared to the complexity of the logical steps that a human will take the brute force solution is much simpler and is a algorithm that the computer can complete with ease but a human would greatly struggle with. However this simplicity comes at the cost of time and efficiency, and so faster solutions can be employed for a computer.

This project focuses on creating an artificial intelligent agent to solve any given Sudoku puzzle that is given to it. This will then be compared to a number of non-AI strategies based on the speed and efficiency of the different solutions.

3 Solving Methods

3.1 Summery of Problem

Sudoku is part of a set of problems that can be defined as Exact Cover. An exact cover is a sub-collection S^* of S such that each element in X is contained in exactly one subset in S^*, this can be Pentomino tiles in a grid, the N-Queens problems or a Sudoku. Sudoku is an exact cover problem as all the numbers 1 through 9 have to be in a subset such that the numbers are in one subset but not in another subset; when the rows, columns and boxes are viewed as subsets of the grid. Common features of the set of exact cover problems include the fact that they are all NP-Complete.

3.2 Brute Force

Solving a Sudoku with brute force is possible and it is the simplest way to go from an incomplete problem to a complete one. A brute force solution applied to a Sudoku grid will simply take every empty space on the grid and place a number in it. If the solution isn't correct, which will happen more often than not, then all the squares will be emptied and a new combination of numbers will be placed in the grid.

Brute force solutions have the advantage of taking about the same time to solve any Sudoku, regardless of the difficulty level. Due to the way the solutions are generated, there are no logical steps to take upfront as the grid is simply filled with any number from one to nine, the only logical

evaluation in this solution is at the end when the final solution has been generated and the board can be evaluated as valid or not.

However, there is a critical downside to the brute force algorithm, which is the overall speed of the solution. Due to the fact that the puzzle is only evaluated once a full solution has been generated there is a lot of unnecessary calculation that takes place. For example if the first square is blank then the brute force solution would place the number 1 in that box, only after placing all the other numbers in the Sudoku and evaluating it will the brute force solution realize that it is incorrect, while a more logical solution will see immediately see that placing the number 1 in the first box was wrong the brute force solution will attempt every combination of solutions with the 1 in the first box until moving on to try a number 2 in the first box. There are a total of $6.671x10^{21}$ [4] possible, valid, solutions for a blank Sudoku, a brute force solution will attempt to step trough all of these solutions and more besides with the aim of finding the one possible solution for the given grid. However, this is by no means the fastest solution or the most efficient, and more elegant solutions are absolutely possible.

3.3 Refined Brute Force

It is worth noting that it is possible to refined the brute force solution, by taking into account at the creation of the board that only certain values will ever be valid in certain units. A Refined Brute Force Search is one in which the number of calculations that the brute force has to perform is reduced. On a normal Sudoku the number of possibilities that the grid has to search is equal to the number of possible values in a unit multiplied by the number of units that have to be compared. Ordinarily there would be nine possible values, but by giving each unit a list of possible values that it can hold upon the creation of the board we can narrow down the number of possible solutions that we have to evaluate greatly.

For example in a given Sudoku the only possible values for the first square may be [5,8,9] this instantly reduced the number of possible solutions that need to be search by 66%, and multiplying in all the other reduced possibility sets further reduces the number of possible solutions that need to be checked. This means that the refined brute force solution can be magnitudes faster than a standard brute force solution.

However a refined brute force search will still suffer from all the same problems that a regular brute force search has. This includes an inefficient use of resources and slow computational time. Although the number of solutions that need to be checked is reduced, compared to the brute force solution, there are still many wasted solutions generated, which, on top of the validity checking that is also needed, makes the algorithm less efficient than others that check the solution as it is being produced rather than at the end.

3.4 Backtracking

For a more elegant solution than brute force solutions there is backtracking. A solution based on backtracking would be simple to implement and is guaranteed to find all solutions in a bounded amount of time. Backtracking is similar to trial and error, such that it tries out each different number in a square, until it finds a match or a fit that doesn't immediately trough up any errors, and then it moves on. It does this until it finds a solution or encounters a square in which no number can fit. If no number can fit in a square then the program must have made a mistake previously and so it backtracks to where the mistake is and tries a different number there. This is where the method gets the name 'backtracking' as the algorithm is constantly trying out different solutions and backtracking when mistakes are found. Backtracking is faster than a brute force solution, in part because unlike a brute force solution the backtracking algorithm does not completely throw away the past progress that it has made.

Backtracking is becomes very fast on problems such as the eight-queens problem, crosswords, and Sudoku because of the asymmetric testing that can be carried out. That is, it is very fast to prove that the solution is wrong, as only one square tested has to be proved incorrect for the whole solution to be proved incorrect. Knowing this it is easy to program a break into the system such that if an error is found the rest of the solution is immediately discarded without being tested. Note that it is not the progress that is thrown away, but rather the rest of the testing is halted, as a Sudoku is perfect and having one error proves the whole solution is invalid. This is known as an early stopping variant, and is used to increase the efficiency of the Sudoku Solver used in this project.

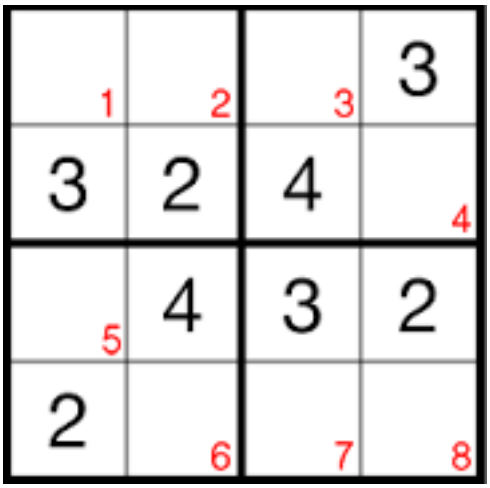

Figure 3: fig:Simple Sudoku
Applied Backtracking on a 4x4 Sudoku.
The blank squares have been labeled 1-8 in red for convenience

Example of Backtracking on a 4x4 sided Sudoku Grid
Position (1) set to 1
Position (2) set to (1 not valid, no valid moves)
Backtracking -> Position (1) set to 4
Position (2) set to 1
Position (3) set to (1 not valid) 2
Position (4) set to 1
Position (5) set to 1
Position (6) set to (1 not valid) 3
Position (7) set to 1
Position (8) set to (1 not valid) 4
Solved

3.5 Dancing Links

Dancing Links is a method of solving exact cover problems, that is both backtracking and depth first. It is a variation of the backtracking method that is above, and it is especially suited to problems where there is lots of backtracking and editing list[5]. The backtracking solution described above has lots of overhead required for keeping track of the various lists. This overhead can include the list of possibilities that have already been tried or generating new list of possibilities when backtracking and maintaining the board to check it at various intervals. The computation for this is obviously fairly large, as the computation for inserting an item into an arbitrary place in a list in O(n).

The Dancing Links Algorithm hinges on the fact that to remove an item from a list the operation can be described by the equation below.

$$L[R[x]] \leftarrow L[x], \quad R[L[x]] \leftarrow R[x]$$

However, if the resulting 'mess' is not cleaned up then the operation to put that item back in the list can simply be described as the calculation described below.

$$L[R[x]] \leftarrow x, \quad R[L[x]] \leftarrow x$$

This means that while the operation to remove an arbitrary item from a list is still O(n), the operation to place the item back into a list after it has been removed becomes O(1). While this may at first seem limited in its usefulness, it greatly reduces the cost of backtracking, which places items back in the list of possibilities. Using dancing links can greatly increase the speed of a program where evaluating the solution requires lots of list manipulation.

Due to the reduced cost of adding and removing items from the lists the calculations for backtracking become faster and more efficient and there is less work being done to write and

re-write the same values in memory. This algorithm can be applied to Sudoku as each unit has a list of possible values that con be placed in it, as the other units around it are filled in and evaluated the possibilities in each unit is effected, removing more and more until only one value remains. However when backtracking each value needs to be re-populated into the field and this can be costly to calculate which values need to be updated and to work out what the change it. Dancing Links would allow the cost of backtracking to be reduced as it would be a simple matter of adjusting the links in the list of possibilities to reflect the changed values. This algorithm would reduce the amount that needs to be kept track of at each stage of the calculation, and would make adding in removed values more efficient.

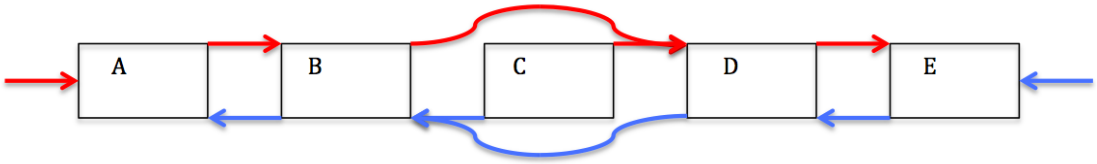

fig:Linked List
Figure 4: This diagram shows that by moving the links from C it can be removed from the list without having to delete it from memory

However due to the language that has been chosen for this project, Java, there is a critical flaw in the execution of this method. In Java the act of garbage collection would see that nothing is holding a reference to the Value of 'C' and would therefore assume that it is not being used. This would then be cleaned up and suddenly the Dancing Links Algorithm would begin to fail. The Dancing Links Algorithm relies on the linked list not being 'cleaned' and the pointers and links between objects constantly moving and shifting depending on the state of the list. With Java's garbage collection thrown into the mix there would need to be a list of items that maintains pointers to the objects that the dancing links list is working on, therefore introducing a separate list to keep track of the original values. At this point the second list that has just been introduced defeats the point of using dancing links in the first place as the point of it was to reduce the complexity of the program and the amount of accessing arbitrary items in a list.

A more appropriate language for coding this solution a language such as C++ as it does not have the presence of garbage collection to interfere with the items that have been 'deleted' from the list. The program could iterate over the list without having to worry that the hidden values are being moves and deleted, despite them still being needed. The advantage of Dancing Links, the fact that no clean up is required, is completely defeated by Java which handles all the clean up for the programmer, which is the opposite of what is required for Dancing Links. While the dancing links algorithm is an elegant solution to the problems that solving a Sudoku presents it is inappropriate to implement it in Java as the language itself does not lend itself to this particular solution.

4 Project Specification

The projects functional and non-functional were laid out in the project proposal, however a more refined version has been included in this report with details on

4.1 Functional Requirements

The following functional requirements for the project describe the metrics that the project will be measured by to ensure that it is completed to a high standard by the end of the project.

1. **Language** The language requirement on this project is Java, as it will be turned into an android app.
2. **Algorithms** Users will have a specific range of algorithms to chose from when solving the Sudoku Puzzle.

   2.1. **Brute Force** A standard Brute Force solution algorithm option

2.2. **Refined Brute Force** A refined Brute Force solution based on the possible values that can be placed in a unit upon creation.

2.3. **Backtracking** A solution that places values in units, re-evaluating and backtracking where necessary when the value is false or no values exist.

2.4. **Dancing Links** A more efficient Backtracking solution based on moving pointers rather than editing list.

3. **Data Access** The app will allow users to use the camera to capture a real life Sudoku board and play it in app.

3.1. **Camera Access** Users will be able to allow the app access the camera to take and store an image

3.2. **File Access** Users will be able to allow the app to access image files to read from an existing image

4. **Image Recognition** The app will be able to read a Sudoku board from an image and represent it as a playable board in the app.

4.1. **Sudoku Board** The app will be able to recognize a board in a given image for images that contain a clear and well formed Sudoku Board.

4.2. **Read Numbers** The app will be able to read numbers from the image for images that contain a clear and well formed numbers within a Sudoku board.

5. **User Interface** There will be a user interface that a user will be able to solve Sudoku's on.

5.1. **Sudoku Board** The user will be able to interact with a 9x9 grid.

5.2. **Operable Units** There will be units that the user can place the numbers 1..9 in, not other input will be permitted, and multiple numbers will also not be permitted.

5.3. **Inoperable Units** Units which are pre-populated with clues will be inoperable and the user will not be able the interact or change these units.

6. **Interact-able Buttons** On the user interface there will be a number of buttons that the user will be able to interact with.

6.1. **Generate Solution** There will be a button on the screen that will generate a full solution for a given Sudoku puzzle on the screen.

6.2. **Hint Button** There will be a button that puts a number on the Sudoku board as a hint for the user, the hint will not overwrite a number that the user has placed on the board, nor will it overwrite one of the immutable numbers. The hint will be an operable unit that the user will be able to edit if they wish.

6.3. **Clear Puzzle** This button will clear all operable units on the board, inoperable units will not be affected.

6.4. **New Puzzle** This will clear all the units on the board and generate a new set of immutable units.

Notes: It is important to note that requirement 1.4 to implement a Dancing Links solution could not be completed due to the limitation of Java as described in section 4.4 of this document.

4.2 Non-Functional Requirements

The non-functional requirements of this project will be used to measure how much the project has committed to its original intended purpose.

1. **Efficiency** One of the key ideas of the project is to create an AI Sudoku Solver that is supported on a mobile device, this means that the program needs to be efficient in order not to tax the phone's CPU.

2. **Performance** Related to the previous point the program needs to be well performing and fast, because of this if the Algorithm used is not efficient then it is rendered pointless as users are unlikely to be satisfied with waiting a long time for the solution to be calculated. The performance of the Sudoku app on the mobile device is critical for creating a good user experience, there should be no noticeable lag when changing activities and the algorithm should be performed quickly and efficiently.

3. **Stability** The program should be highly reliable, this includes not freezing on the user in the case that the solution to calculate the problem fails. The program should handle errors and long waiting times in a graceful manner, giving the user the option to cancel the operation.

4. **Availability** The application should run without the need to access the internet. All the algorithms that are used in the program should be solved at the time by the device, and should not need to rely on an external server to perform the calculation.

5. **Usability** The program should be easy and simple to use, all functionality should be self evident and require a minimum of explanation to the user. This will be achieved by keeping the number of activities and buttons to a minimum, and by only having vital functionality included in the app.

6. **Documentation** The program should be well documented throughout, with thorough explanations in key areas such as the algorithm classes and other critical sections. This only needs to take the form of comments within the code, as due to the usability requirement external comments will not be needed as the program will be self evident.

5 Design

Early in the design process of the Sudoku solver the decision was made to make it a mobile app. This design choice was made because by its nature a Sudoku is a fun puzzle to solve to pass the time. By having the Sudoku Solver paired with a Sudoku app the solver has more of a purpose and the app can be provided with a built in solver for when the user cannot solve any more and wants a hint, or when the user wants to confirm that they have the right solution. This allows for further additional features in the future of the app, such as taking a photo of a partially solved Sudoku and using image recognition to place it in the app.

As the app was designed with a phone app in mind the coding has been done in Java to facilitate being made into an android application. Though Java has the disadvantage of being somewhat slow as an operating language this can be worked around from the users perspective, and provided the algorithm used to solve the Sudoku is efficient then the result should not be effected too severely by the language. The major advantage that Java has is the ability to use it on android phones, which is the operating system on more than 80% of smart-phones [6].

6 Implementation

This section covers the design and implementation of the elements of the project. This has been broken in to the a section on the design of the application, a section of the algorithms used to solve a Sudoku, the image recognition that is used to read in pictures of a Sudoku for image processing, and finally the last section is on the user interface and the design choices that were made in regard to it.

6.1 Algorithms

The project is to first and foremost develop an algorithm that can be used to create a solution to a given Sudoku. Because of this the initial work on the project was made on the back end, to make the various different algorithms that can all be compared and contrasted against each other. At the point of writing most of the algorithms that were described above have been implemented and the key methods of each have been included in the description.

The first algorithm that was implemented was the brute force search, partly due to the fact that it was the simplest to implement on the system. Though the brute force search was the first

to be implemented it has not yet been run successfully to completion as the calculations take so long and the solution is so inefficient.

6.2 Image Recognition

A Sudoku can be read from an image by using image recognition and various algorithms to detect the edges of the Sudoku square and the numbers within it; this information can then be used to create a representation of the Sudoku. The application can access both the camera and the image folder on the phone, once the user grants permissions, in order to get the sample image. In order to detect the Sudoku in the picture the application must first threshold the image, it does this because the edge-detection algorithm works best on simple gray-scale images. The thresholding is done with a variable algorithm which takes the average luminescence of the surrounding pixels in order to decide whether any given pixel should be black or white. By using a variable thresholding algorithm the app can cope with images that have different brightness' or patches and lighter and darker areas.

6.3 User Interface

Progress on the user interface has not yet begun, however there are the basic designs in place for the user interface that can bee seen below.

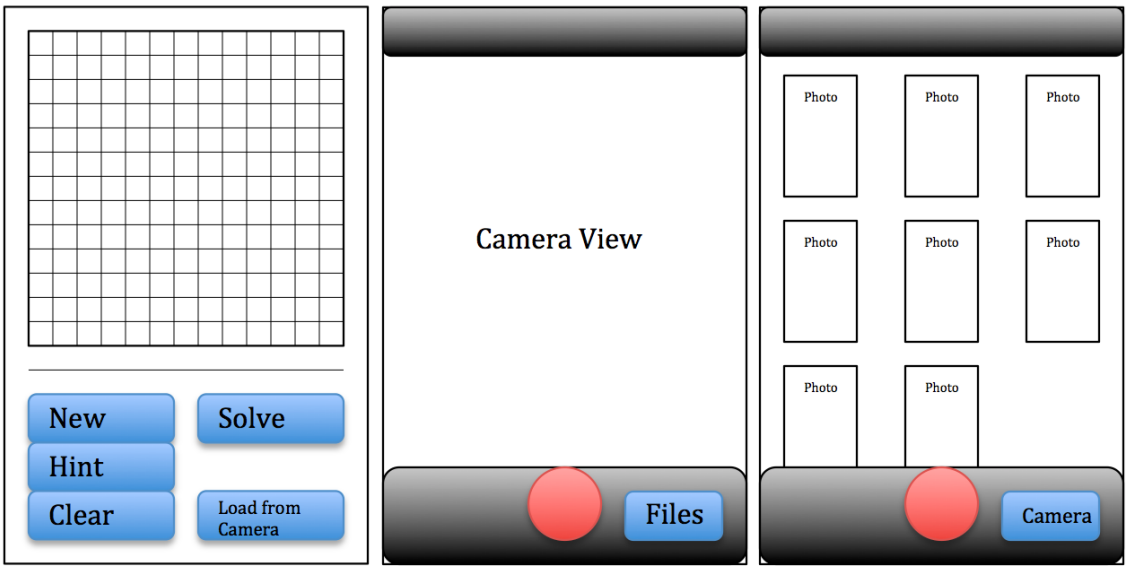

fig:User Interface Designs
Figure 5: These images show the initial designs for the user interface

These designs are made to place emphasis on the main part of the program, namely solving the Sudoku on the screen. The user has access to a menu for doing things like solving the Sudoku and reading in an image of the Sudoku for the application to solve. These designs have not yet been subject to a feasibility study and so are likely to change by the end of the project.

6.4 Testing

Testing is an important part of this project as the Sudoku needs to be given back completely solved in order for the project to be considered a success. In this project testing has been built into the time plan and is completed at every major stage of the project. The tests for the algorithms are simple JUnit tests, this is because all that is needed to confirm at this stage of the project is that when an algorithm is given a solvable Sudoku puzzle it will return a completed solution. There are also tests in place to evaluate unusual edge cases, such as the original puzzle not being solvable. It is possible for the original puzzle to be unsolvable in the case where an image is read into the program by the image reader, however a mistake has been made on one of the numbers that is not immediately obvious. In cases such as these it is required that the application handle errors gracefully in order to give the user a pleasant experience when using the application.

Additional testing of the user interface will be done with the Usability test. Participants will be asked to perform a few tasks on the application and their feedback on the project will be taken into account for future changes to the user interface of the system. This study will also be used primarily to evaluate the intuitiveness of the layout. However, the study will also be use to find errors and bugs that were not spotted during the initial testing. Together the usability study and the unit tests will remove the obvious bugs in the program and so the Sudoku solver will meet the requirements to be as reliable and bug free as possible.

7 Progress

This section covers the progress that has been made this far in the project and a critical evaluation of that progress.

7.1 Project Management

This project was designed to be completed in a linear fashion, with all progress on the backend, algorithms and image handling, made before the front end, user interface. The reason for this style of progress is to make sure that the more technically difficult parts of the project are fully completed by the end, as if there are any unforeseen difficulties or bugs then there will be plenty of time to fix them before the final deadline. This will be to detriment of the front end, however, in the case that there are issues with the back end of the program then it will be more useful to have a functional program that meets all the requirements and specifications, than a program that looks pretty but doesn't meet as many of the specifications.

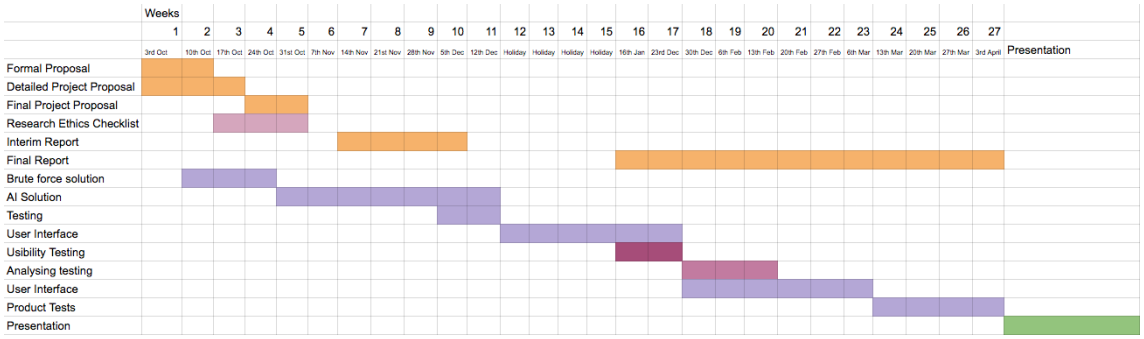

fig:Gantt Chart
Figure 6: This chart shows the original time plan for the project

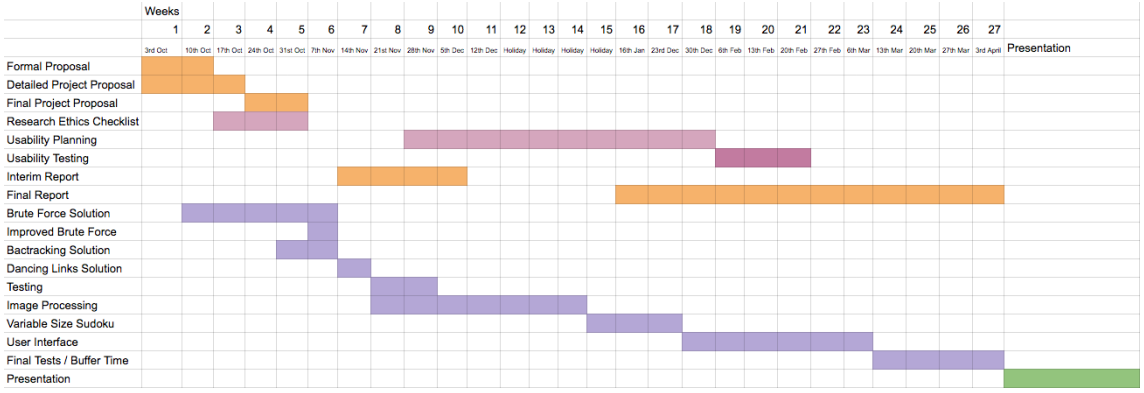

fig:Updated Gantt Chart
Figure 7: This chart shows the changes to the project and the updated time plan for the rest of the year as ten weeks into the project

At the beginning of the project the time was the Gantt that can be seen as Figure 5 However it quickly became apparent that the burden of the project was far less that had been anticipated at the beginning of the project. And so at the time of writing the Gantt chart for the project has been adjusted to Figure 6.

In terms of time management the project was severely overestimated, this is in part due to not enough reading of the subject before the beginning of the project. Upon research of existing solutions and other algorithms that can be adapted to the needs of the project, it quickly became obvious that the Sudoku Solver was not ambitious enough. The original plan has evolved to reflect this and has changed to become more detailed and ambitious in its design.

7.2 Contributions and Reflection

The scope of this project has increased to include image processing and there is an additional task at the end of the project to enable the algorithms to work on any size Sudoku. These additions have been made both to increase the difficulty level of the project, and also to add more useful features to the project. By adding these extra features to the project hopefully it will be a more fulfilling project that has gone above and beyond the original specification.

The speed at which the project is progressing is very strong and the quality of the work that is being carried out is good. This project stands at a good place at this critical halfway point, with a large amount of work already completed, including most of the work in the original brief. By adding new features and requirements to the project it can be made more engaging for both the developers and the final users. However while the progress of the project has been good so far, with the added burden of more features it is important not to get complacent and to continue working on each element of the project in a considerate fashion.

References

[1] Gary MacGuire, 1 Jan 2012, "There is no 16-Clue Sudoku".
arxiv.org/abs/1201.0749 Retreved 27 October 2016

[2] David Smith, 15 May 2005, "So you thought Sudoku came from the Land of the Rising Sun ...".
The Observer. Retrieved 27 October 2016..

[3] Dell Magazine.
`pennydellpuzzles.com/`

[4] Bertram Felgenhauer, Frazer Jarvis, 25 January 2006, Mathematics of Sudoku I
`afjarvis.staff.shef.ac.uk/sudoku/felgenhauer_jarvis_spec1.pdf/`

[5] Donald E. Knuth, Stanford University, 15 November 2000, Dancing Links
arXiv:cs.DS/0011047 v1

[6] Gartner, Press Release, 15 May 2016, Table 2
ahttp://www.gartner.com/newsroom/id/3323017