Final Year Project Report

Full Unit – Interim 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

December 02, 2013

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 Where other published and unpublished source materials have been used, these have been acknowledged.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

Student Name: Thomas Paul Clarke

Date of Submission: 04/12/2013

Signature:

Table of Contents

Abstract

This Interim report consists of a colaberation of reports that have been made thoughout the first term. Each report has a subject matter that is related to the final project which will consist of a Sudoku solver using constraint satisfaction. These reports also discribe concept programs that have been created in order to demonstrate key algorithms in both the programs and the reports.

SVN Archive

https://svn.cs.rhul.ac.uk/personal/zwac016/CS/Projects/Submissions/ZWAC016/

Project Aims & Objectives

My aim is to provide detailed reports that will help to explain and gather the required knowledge I need on key concepts, which I will be using within the Sudoku Solver and concept programs.

These programming techniques will be made within the first term:

1. Graphical User Interface
2. Layout managers
3. Data Structures
4. Design Patterns
5. Constraint Satisfaction
6. Complexity, NP Hardness and the Big O Notation

To illustrate and explain the programs I'll be making, which will help me to practise the advanced programming of the algorithms that might need to be used within my final project.

These techniques will also be made within the first term:

1. User Interface
2. Data Structures: Hash Tables, Priority Queues, Dynamic Trees and Stacks
3. 8Queens
4. TicTacToe

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 Alphabet 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	
Holiday	Final project coding and design	
	Outline of report	

Chapter 1: **Graphical User Interface**

Graphical User Interfaces are of fundamental importance to a program that is 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 the first Interface was displayed in 1968 by Douglas Englebart[1]. The project involved implementing a display of everything a user would require, whilst making the interface instantly (visually) understandble. It also aimed to allow users to contribute their desired inputs, which outlines the key advantge of the GUI.

1.1 Displaying Inputs/Outputs

This practise also needs to be followed within the GUI for the Sudoku Solver. This 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 acessible 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.

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

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 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

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.

1.2 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

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 if needed, there is 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.

1.3 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 of 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) will keep the options available in a hidden menu, which will make the design cleaner and the Sudoku solver simpler to operate.

1.4 Heuristics

Using a resource[2] 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.

1.4.1 Status Of The System

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

1.4.2 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.

1.4.3 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.

1.4.4 Error Prevention

It is important to prevent the program from having errors and to implement safegaurds 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 safegaurd was implemented into the

[2] 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

program, the user would only be able to choose a button between 1-9, which woulprevent an error from occuring.

1.4.5 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.

1.4.6 Design Accessibility

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

1.4.7 The Design

It is vital that only neccassary and inportant information should be displayed to the user, which will prevent them from getting confused and unindated with large quanities 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 relvant times.

1.4.8 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 it was caused by them.

1.4.9 User Assistance

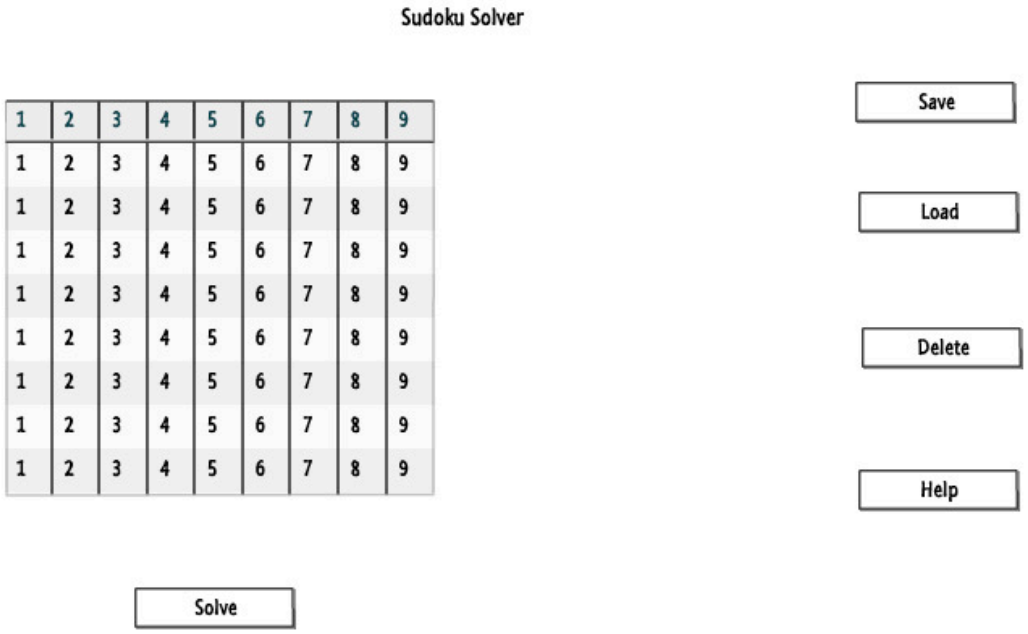
A help function will be implemented to explain the game and the user interface, which can assist a completely new user (with no understanding of the program or the game) on how to work and use it. This could be done by using temporarily displayed boxes that contain useful rules and instructions about what the user needs to do. For example, a drop down box could be used to show the game rules and processes. Or an alternative could be the display

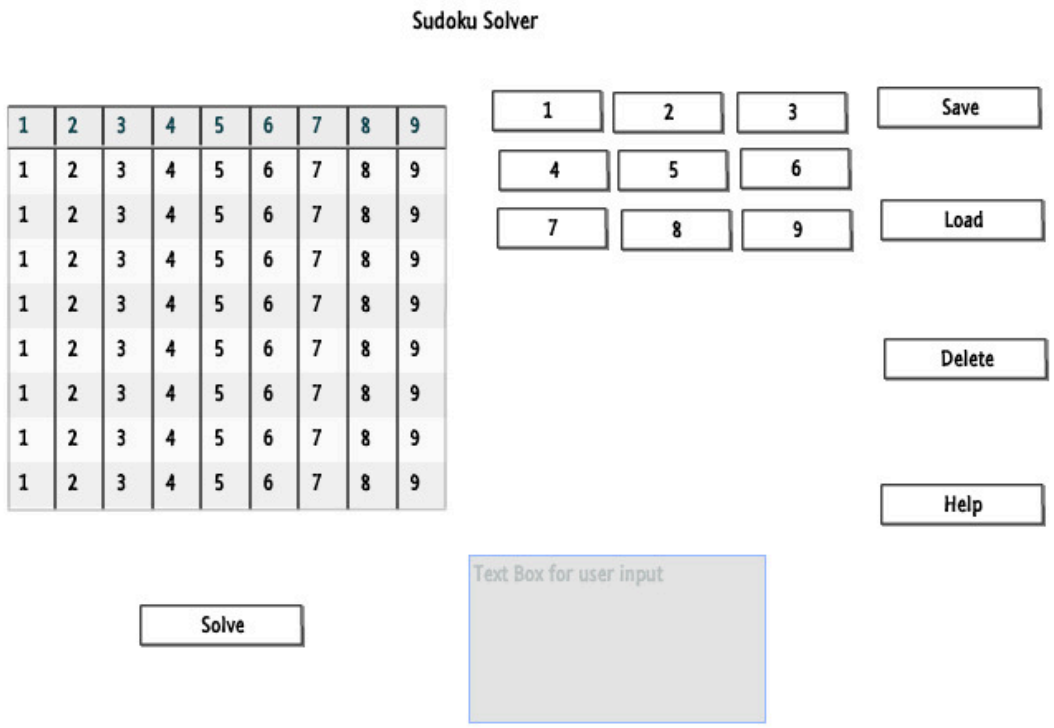
of pre-written manual. However these are not the most modernistic or practical assistance tools because users often want to get started on the program as soon as possible or expect a visual tutorial.

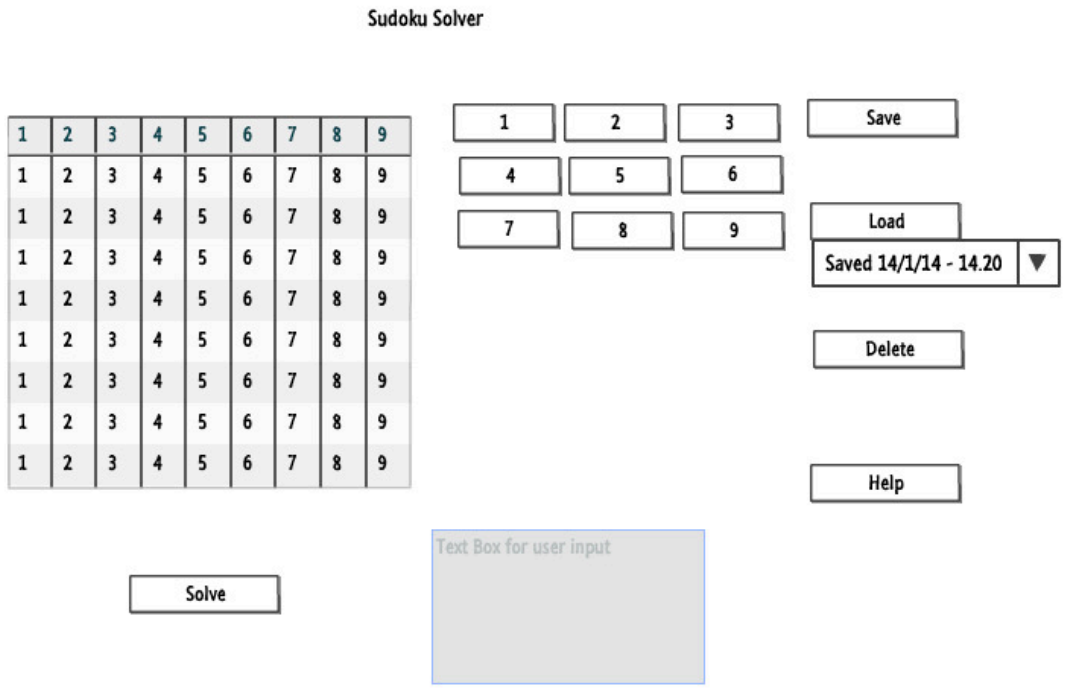
1.5 Conclusion

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 _____ *9*







Chapter 2: **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 effected when the window is changed. There is a 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.

The layout managers work (despite there being many different strands of managers) by using code to dynamically position the object that will be held there. Grouping, setting and linking components keep the objects in a relative position.

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.

These types of managers are used to help the user in many different situations. For example, my project will be focusing on the grid and group layout.

2.1 The General layouts

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

2.1.1 Border 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[3]:

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

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

2.1.2 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));.
```

2.1.3 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.

2.1.4 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.
```

2.1.5 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.

2.2 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:

[4] 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.

The layout system is very easy to use for an experienced *and* new programmer. It is possible to add the component to the layout that allows it to connect with the controls. However it is also possible to use programs like Window Builder Pro that allow the user to

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

create the layout using a faster and more visual experience where the layout restrictions can be visible instantly.

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.

2.3 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.

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.

2.4 Conclusion

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 a 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 mangers, 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.

Chapter 3: **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.

3.1 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. Although 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.

3.1.1 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 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.

3.1.2 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.

3.1.3 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.[5] 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.

This is done by using the simple commands of push and pop, which essentially to 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.

---

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

3.1.4 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 are 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. This is done by adding each element with its priority number, which gives the order the data 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.

We have these kinds of data structures to allow the abstract data types to simplify the coding process and provide useful limitations, which will only allow the specified commands to be done. They also enable the dynamic and faster input of elements than an array could provide.

3.1.5 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.[6]

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.

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

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

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.[7]

An advantage of heaps is their ability to be versatile within 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.

3.1.6 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 exception, 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.[8]

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 and a set of related data that needs to be found within the data set.

However the disadvantage of hash tables (in the worst case) is when all of the elements are given the same hash code and are all stored within the same linked list. This means that all of the search functions will still have to traverse between every element, which saves no time.

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

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

3.2 The Inclusions of the Date Structure

When programming my concept programs the data structures I've used and encountered for the first time have allowed new functionalities and more effective ways to increase the effectiveness of the program. Each data structure discussed within this report will be included within the final project, the Sudoku solver:

3.2.1 Priority Queue

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.

3.2.2 Hash tables

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.

3.2.3 Trees

These allow the functionality for storing the boards (that will differ from each other) and finding and traversing a tree will allow these boards to be found. There are also functions that will aid this, for example, the tree increasing in size whenever a new board is input dynamically. It is also possible to limit or cut a branch when needed to if the boards are then repeated.

3.3 Conclusion

These data structures will be a key part within my finial 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.

Chapter 4: **Design Patterns**

Design patterns provide the structure for a program, which is done by giving a concept or design of a program to solve a problem. A design pattern does not specify code but is used to simply 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 fix the problem, which often helps to make the code more understandable with a set layout.

Each pattern will define a structure. It will have to be designed (often) with links to 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 build for a different purpose.

Some of the design patterns that I have researched are linked to what I'll be needing within the Sudoku program. 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.

4.1 Using Different Design Patterns

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.

This is possible by using the MVC design pattern that stands for model, view & controller. These are all classes that will act with each other and break up the user interface part of the program. Acting upon the user inputs the event handlers (that are placed within the view) pass the input to the controller that (depending on the input) pass it to the model. This is the gateway to the rest of the program, which is the way that the design pattern

handles requests from the interface like a button. The view is used to display the interface and contains the code, which is linked to the event listeners. The controller is used to request information from either the view or the model, which is able to run the program and do the necessary calculations that the user requires.

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.

There are design pattern options to be used to obliterate this common issue. There will need to be some debate about which is most likely to benefit, the program or the programming difficulty. There are ways that these design patterns are able to work together.

4.1.1 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[9].

It 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.

4.1.2 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 objects within the system as

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

there could be many and a lot with the same values. This 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 thae 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.

The Sudoku solver uses constraints on the objects and these need to be implemented at different times. Often it would be effective to have a design pattern that would allow the implementation of the 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 coincidingly.

4.1.3 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 that could be helpful in the project to restrict unnecessary checks taking place, which could reduce the time taken to complete constraints.

This pattern gives the program increased flexibility and allows each cell to be treated independently with the algorithms it needs. It would also be beneficial to have a system that splits up by checking for the constraints within the program. This is because I've noticed within the concept programs that the code can become cluttered when there are several constraints.

Within the Sudoku solver there are many data structures that are used in several ways to allow the program to function correctly and by using the fastest methods to do so. However, there will need to be a significant amount of logistics [10] performed on each data structure. 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.

[10] http://www.codeproject.com/Articles/186185/Visitor-Design-Pattern

4.1.4 Visitor pattern

The Visitor pattern seapartes an object's logic and structure by splitting their classes and allowing them to be called when needed. This allows them to be independent and each object will be able to handle the logic that is needed at the correct time. In order to do this the classes are created for each algorithm and then the object is required to apply the logic if it's needed.

Within the Suduko solver program it could be beneficial to limit and monitor what is being called and how each cell is being influenced. This is especially during the design stage to see whether having separations will make the program simpler.

4.1.5 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.

4.2 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 in the future, 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.

Although, when working within a team each person will have their own experience with patterns and their way of programming. Within a group there can sometimes be too many ideas for design patterns that could actually harm the programs performance instead of help it.

4.3 Conclusion

I'm going to incorporate most, if not all, of the design patterns that I have described above. 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.

Chapter 5: **Constraint Satisfaction**

A constraint satisfaction problem is a specific conundrum that is best solved using strategies that relate to constraints. These are the limitations that are put on an object. The process of creating these objects is linked to what that object will be able to do to validate its value or a value it could take on. There are many ways to provide constraints. They are essentially a set of rules that the objects associated with it must conform to, which will provide the actions needed with the Boolean outcomes from these operations.

Use of constraint satisfaction problems are often linked to logic puzzles like Eight Queens and Sudoku. This is because the games rules are all about limiting what is placed in them and their location is due to collisions or duplicate numbers. There are many different ways to apply constraint satisfaction and each program needs to have all of the rules in place so it can be applied correctly.

A constraint[11] is split into two parts, the variables and the domain. The variables are the questions needing to be answered and, in these programs, it will be a location on a board that needs to be checked if it is a correct or a false state. The domain is the parallel part of the constraint and contains a set of values that are the allowed for the variables use. If the variables within the constraint are also within the domain then the constraint can be satisfied.

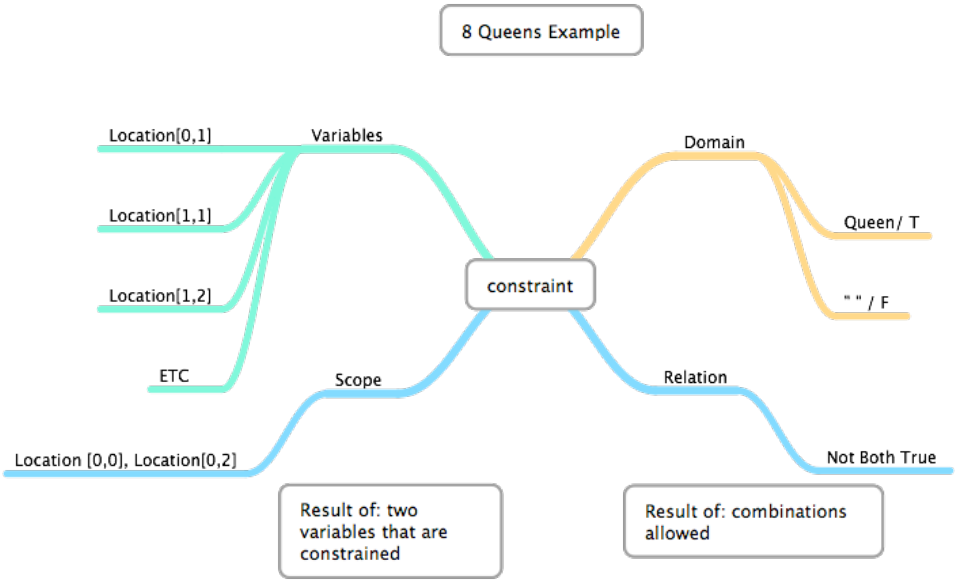
Each constraint has variables and in this example the variable is a Queen from the 8 queen puzzle. In this case there is a variable in the location the queen has been placed in, which now needs to be checked. The method is using constraint satisfaction by using the domain this is given with the variable and contains the value that the Queen can take. If it is satisfied it can be placed as a Q otherwise the cell will be left empty.

Within the constraint there is, what we call, a scope. It is the relation between two variables that are constrained, (coming from the tuple) which assigns the variables to the scope. When the two constraints are matched together we call it a scope and these are then constrained to produce the relation. The scope being currently used is utilized by the relation and it this relationship, in terms of their values, which allows them to combine.

[11] 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

5.1.1 Constraint Diagram

The diagram below shows this process related to the 8 Queen puzzle as an example to show the way P = (Variable, Domain, Constraint)



5.2 Constraint Techniques

Within constraint satisfaction there are techniques that can be used to increase efficiency. However, these are deemed deterministic, which means, that unlike a search, they will provide the same results each time. So given a particular input there will always be a constant output.[12] This proves to be very useful as it allows the computation to be made as soon as possible. [13]

To improve the constraint it is possible to select the objects that are no longer needed due to a variable being domain consistent. Meaning that within the domain of the variable there contains a value that is matched within the constraints. [14] Therefore this branch, and any others it continues to follow, is not needed because it's not valid. By pruning the domains as much as possible allows the efficiency advantage without creating impossible boards.

[12] http://technet.microsoft.com/en-us/library/aa214775(v=sql.80).aspx
[13] Discrete Mathematics & its Applications, 6th Edition. by Kenneth H. Rosen. Page 836
[14] http://www.cs.ubc.ca/~kevinlb/teaching/cs322%20-%202009-10/Lectures/CSP3.pdf

5.2.1 Backtracking

I have used a method called backtracking, which unlike forward checking and arc consistency, (discussed later in the report) does not have any propagation. This means that there is no reproductions of the states and it is the same board that is edited and deleted to find the desired state. Backtracking works by searching and if the outcome matches the desired output then it passes.

Backtracking is a recursive algorithm used to find a desired output through use of the brute force approach. It does this by starting at the root of the tree and following a consistent path until the end of that path within a leaf. When the leaf is evaluated into the correct output the algorithm stops and the value will be found. Or an incorrect node is found and then the process is repeated by going back to the parent of the leaf and following and checking another ode. This is done until the full tree has been traversed and there are no correct nodes on that tree, or that a correct node is found and the recursion stops. [15]

This is the backtracking algorithm I've used within the 8Queens 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++;
                        }

                        else {
```

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

```
                                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 does not pass then the attempt is withdrawn and another attempt is used.

5.2.2 Arc Consistency

However Arc Consistency is different because the domains are reduced and there are propagates of each state. The Generalised Arc Consistency 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 consistently and arc consistency is the use of non binary constraints using tuples of variables instead of a pair.

5.3 Arc Consistency Algorithm

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

Generalized arc consistency algorithm[16]

1: **Procedure** GAC (Variables, Domain, Constraints) [17]

2: **Inputs**

3: *V* // a set of variables

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

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

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**

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

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

$Domain_{ABC} = \{1,2,3,4\}$

TDA = {(A, A<B), (B, A<B), (C, B<C)}

11: **for each** variable *X* **do**

12: $D_X \leftarrow domain(X)$

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

$Scope_A = \{A,B\}$ $Scope_B = \{A,B\}$ $Scope_C = \{B,C\}$

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

Step 1:

Select = (A, A<B)

$Domain_A = \{1,2,3,4\}$

After Step 2:

$Domain_A = \{1,2,3\}$

TDA = {~~(A, A<B)~~, (B, A<B), (C, B<C)}

Step 3:

Select = (B, A<B)

$Domain_B = \{1,2,3,4\}$

After Step 4:

$Domain_{B2} = \{2,3,4\}$

TDA = {~~(A, A<B)~~, ~~(B, A<B),~~ (C, B<C)}

Goto Step 5: (line 18-20)

Step 7:

Select = (A, A<B)

$Domain_A$ = {1,2,3,}

After Step 8:

$Domain_{A2}$ = {1,2}

TDA = {~~(A, A<B), (B, A<B),~~ (C, B<C)}

Goto Step 9: (line 18-20)

Step 11:

Select = (C, B<C)

$Domain_C$ = {1,2,3,4}

After Step 12:

$Domain_{C2}$ = {2,3,4}

TDA = {~~(A, A<B), (B, A<B), (C, B<C)}~~

Goto Step 13: (line 18-20)

Step 15

TDA set empty, all must be reduced

17: $ND_X \leftarrow \{x \mid x \in \boldsymbol{D}_X$ and some $\{X=x, Y_1=y_1,\ldots,Y_k=y_k\} \in c$ where $y_i \in \boldsymbol{D}_{Yi}$ for all $i\}$

Step 2:

A = 4 (Did not pass test)

$Domain_{A2}$ = {1,2,3}

Step 4:

B = 1 (Did not pass test)

$Domain_{B2}$ = {2, 3, 4}

Step 8:

A = 3 (Did not pass test)

$Domain_{B2}$ = {1, 2}

Step 12:

C = 1 (Did not pass test)

$Domain_{C2}$ = {2, 3, 4}

18: **if** ($ND_X \neq \boldsymbol{D}_X$) **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 its scope.

Step 5

(A, A<B) with $Domain_{A2}$ = {1, 2, 3}

Compare

(B, A<B) with $Domain_{B2}$ = {2, 3, 4}

False - (A, A<B) $Domain_{A2}$ needs to be {1,2} Goto Step 6

Step 9

(A, A<B) with $Domain_{A2}$ = {1, 2}

Compare

(B, A<B) with $Domain_{B2}$ = {2, 3, 4}

True - (A, A<B) $Domain_{A2}$ is {1,2} Goto Step 10

Step 13

(B, A<C) with $Domain_{B2}$ = {2, 3, 4}

Compare

(C, B<C) with $Domain_{B2}$ = {2,3,5}

false - (A, A<B) $Domain_{A2}$ is {2,3,5} Goto Step 10 // This is repeated until

true - (A, A<B) $Domain_{A2}$ is {5} Goto Step 14

20: $\boldsymbol{D}_X \leftarrow ND_X$

Step 6

(A, A<B) $Domain_{A1}$ = {1,2,3}

return to top of loop

Step 14

(A, A<B) $Domain_{C1}$ = {4}

return to top of loo

Step 10

(A, A<B) $Domain_{A1}$ = {1,2}

return to top of loop

Step 10

(A, A<B) $Domain_{B1}$ = {2,3,4}

return to top of loop

5.4 Evaluation

I evaluated this algorithm to show what each step does and how it relates to some real values. This example shows how 3 different constraints are used on a domain and how they affect each one with the variables within it. The steps of the algorithm show how it is intended to reduce each domain, which are not then used within the next part of the program.

This can relate to the Sudoku solver: There are several different ways to create each board, store them and trim the branches of the boards that are not valid due to duplications. However this process can cause unnecessary processing, as each board will have to be tested until one returns false. Although the branch is trimmed it will still have been produced.

This algorithm prevents this by disallowing redundant boards to be created as the given domains are reduced to prevent propagation. This means there'll be duplicates when a tree of boards are created. Instead of trimming the tree the process has prevented them from being created. Therefore, the constraint will increase the efficiency of the program.

Another constraint propagation algorithm is the Dynamic Variable Ordering process, which uses the most constrained variable to be the first one utilized. Its purpose is to allow the dead ends to be discovered as early as possible. Also, without needing to be considered from other variables, this effective heuristic cuts down the search space. [18]

Dynamic ordering means that the choice of the next variable used will be dependant on the state of the search. This is found using forward checking, which is the opposite to backtracking. It means that the current domain is dependant on the previous and current constraints. The reason for this is to use the best possible variable in order to increase the efficiency.[19]

The concept of dynamic variable ordering is to compare all the available variables and 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

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

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

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. [20]

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:

Cell = variable
Column = {2,4,3}
Row = {1,2,3,4,5,6,7,8}
Square = {1,2}

```
For loop {
Constraint 1 = ( cell.value != row.cell[i] && cell.value != column.cell[i])
Constraint 2 = (cell.value != square.cell[i])
}
```

5.5 Conclusion

Constraint 1 has more constraints within it so the dynamic variable ordering will use the variable that has the most constraints. By using the constraints tests will be able to be run. Failing the tests means that the possibility of that option is ruled out, which (in this case) means that the most restrictive test, constraint 1, can be used and constraint 2 might not even be considered. [21]

Constraint satisfaction is essential within the Sudoku solver. It will enable the process of board handling, which entails the constraint finding the next cell by using a tree that contains the options. However this will have to be restricted by using these algorithms. It makes searching more effective by reducing the options, which only allows valid boards to be sourced.

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

Chapter 6: Complexity, NP Hardness and the Big O Notation

Each program has a runtime. Alongside this comes the complexity of the algorithms, the usage of the program, and how efficiently it does its job. There are cases where programs are not able to perform a task due to the constraints on time and the speed in which a problem can be solved.

6.1.1 Complexity

We explore and define the complexity of an algorithm to understand the amount of time it will take to solve a problem. It's important to follow this procedure in order to get an idea of how long the algorithm will take. This can be done using a timer, which can be used to time a small section of the algorithm. For example, a recursion, and this can be incremented by the amount of times the algorithm can be run in a worst-case scenario. [22]

Complexity is not an accurate value and consists of a multiplication of the scenarios. It uses this number to suggest a value to the user about how long the algorithm will take. This is also a way to view how long the computer will demand to complete the task. In some cases a faster computer can help this situation.

An example of this could be the traveling salesman problem.[23] It describes the scenario of a travelling salesman, who jumps between several cities. The program is used to find the shortest route to travel to each city, whilst walking the smallest distance. However, each city has to be checked against the others before each distance is calculated. This problem takes factorial time to be completed and 10 cities will take 10! Time.

6.1.2 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.

[22] http://www.cosc.canterbury.ac.nz/csfieldguide/student/Complexity%20and%20tractability.html
[23] Time and Space, Designing Efficient Programs. Adrian Johnstone & Elizabeth Scott. Chapter 5,Page 68 – 75

6.1.3 The Class NP

Within the project aspect the main aspect to look at is the problem large sets of possible nodes and the constraint such as time[24]. It is usually beneficial to find the best possible solution to a problem. However, this is not always possible, which is proven by the Traveling Salesman Problem. Therefore it is possible to categorise problems by using a threshold to limit the distance travelled between the cities. This limitation can be a used to enable the program to run more effectively than the threshold. If it is better then the decision problem then it is true, or else it is false.

There are problems that can be similar and their complexity can cause them to be unfeasible in order run in any practical amount of time. However, there are "tractable solutions"[4] that can be adapted to others, and this group is NP-complete. "Complete means they can all be converted into each other"[25] and contains the hardest problems within NP. To be within the NP there needs to be a polynomial time algorithm, which means that all the algorithms within NP have a polynomial time solution.

6.1.4 NP Hard

NP hard are a set of problems that are difficult to calculate. They are within the set NP-Complete but not within NP or P, which is due to the NP-hard set of problems being within the hardest NP-complete problems. The reason for having this set is to distinguish the sets that are hard to be solved. For example, the Traveling Salesman Problem is NP-Hard even when it's cut down to one node to be visited at once.

6.1.5 Sudoku is NP-Complete[26]

Proving that the Sudoku puzzle is within NP-complete[27] was documented in 2002. To summarise these findings, the puzzle contains a n2 x n2 grid that is split into n x n squares, which then contains n values. Each search is done on a value. In order to find the correct answer, if there was to be a search on every cell, there are a minimum of n-1 cells in each block that could be searched, which all have different n values. This could take n·n! Time.

[24] Algorithms and Complexity 2, CS2870 By Gregory Gutin February 16, 2013. Page 93
[25] http://www.cosc.canterbury.ac.nz/csfieldguide/student/Complexity%20and%20tractability.html
[26] Graph Algorithms and NP-Completeness. Kurt Mehlhorn. Page 171- 211
[27] http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/SIGAL87-2.pdf

This proves that the algorithm is hard. However there is a case in which the Sudoku size can be reduced for n = 3, which can then be solved by using a backtracking method[528], known as a Latin Square. This proves that it belongs in the NP class due its ability to be solved in polynomial time. I can apply this to the Sudoku Puzzle, as it can be a section of a grid. It can be performed n times, so it will now be O(n!), which is a vast improvement. Yet, it still proves that have a poor scalability.

6.2 Conclusion

The information I've discovered about the process of solving the Sudoku puzzles will now change the way the program will be designed and made. I have realised, by trial and error, that searching each cell to get the correct answer isn't practical, as there are too many iterations that will have to be made. It has also given me a secondary program to look into, which is the Latin Squares algorithm.

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

Bibliography

Book Resources

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

Cay Horstmann, Big Java: Forth Edition, Page 629-663

Darryn Lavery, Gilbert Cockton and Malcolm Atkinson, Heuristic Evaluation, Usability Evaluation Materials: The Department of Computing Science, University of Glasgow

Adrian Johnstone & Elizabeth Scott,T'ime and Space' in Designing Efficient Programs. 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

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

Web Resources

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

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

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

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

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

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

http://www.codeproject.com/Articles/186185/Visitor-Design-Pattern

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

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

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://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.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

Multi-Media Web Resources

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