EXPLORING SUDOKU ~ AN NP PROBLEM

REPORT & IMPLEMENTATION BY
CRAIG PULLAR

Table of Contents

INTRODUCTION

The aim put into a statement is *"to implement and compare various algorithm's performance at solving sudoku puzzles"*. In more detail it is to implement code which will generate solvable sudoku puzzles and implement different techniques to solve them and see which performs the best and why.

The goals of the project are as follows:

- Implement the following algorithms:
    - *Random Search* – This algorithm fills the blank spaces of the puzzle with random numbers until a solution is found. It is based around the fact that a computer can do this so quickly that a solution will be found relatively quickly despite the low chance of it occurring.
    - *Brute Force* – This algorithm goes through each point systematically placing a number in the them starting from 1. As soon as a conflict occurs in the puzzle, a conflict being where two numbers can't co-exist in the same row, column or segment, it will back track through the puzzle incrementing the values by 1 until the conflict is resolved.
    - *A* Search* – An algorithm which will cycle through all possible moves (depth-first) by ordering them according to a heuristic or scoring system to determine which moves are more likely to solve the puzzle quicker.
    - *Constraint Solver* – An algorithm for logically solving a puzzle, similar to how a human may approach the puzzle. It does it by identifying pairs of numbers which lead to blank spaces in the puzzle where only one number may exist.
- Implement a Graphical User Interface (GUI) for testing the various algorithms.
- Implement an algorithm for creating sudoku puzzles of varying difficulties.
- Compare the algorithms performance across puzzles of varying difficulty.

The intended audience of this project is the staff at Cardiff University, specifically my supervisor and moderator for the project. However the audience could also include members of the Artificial Intelligence (AI) community and game developers looking to implement a Sudoku game with the ability to solve and generate puzzles. The greatest

beneficiaries of the work done however will be myself due to how much I have learnt completing the project.

The scope of the project is mainly focused around AI work however it also includes game theory and user interfaces (UIs). Artificial Intelligence being the study of creating software and hardware capable of intelligent behaviour and user interfaces being the creation of windows allowing users to interact with software. However the UI is not the focus of the project and only an added extra so at its heart the project is an experiment in AI.

I will explain the approach of the project in more detail under the approach section of this report however I will summarise generally here. The project was approached by first implementing an algorithm to generate Sudoku puzzles which I could test, then I implemented all of the algorithms independently in no particular order and tested them against the puzzles generated by my own code. Once all algorithms were functional I brought them all together in a GUI which was designed to allow testing of the algorithms. Once completed I tested the algorithms using the GUI and gathered my results to draw my conclusions.

Assumptions made which the work was based on:

- The problem was not so different to solving other games which I had covered in the past and therefore similar approaches and algorithms could be used.
- That human techniques for solving Sudoku puzzles could be used in an algorithm, this brought to light some interesting results on the difficulty of replicating human techniques in code.
- The project would not require any special hardware requirements and could be run on your average modern computer.

The important outcomes of the project were to see how the algorithms compared against each other across the differing difficulties of puzzles. From this you could gather what kind of AI algorithms were suited for solving the problem as well as seeing whether or not the problem was comparable to other games. Another important outcome was to see if the varying difficulty of the puzzles had any affect on

which algorithm performed the best E.G. One algorithm performing better on a lower difficulty than another but performing worse on a higher difficulty in comparison to the same algorithm.

BACKGROUND

The project is based on AI theory so I will give you a little bit of a background on AI. As described earlier AI is the study of creating software and hardware capable of intelligent behaviour. An example of what is meant by this is a computer being able to hold a conversation with a human or playing a game such as Sudoku. AI's overall aim is to simulate human intelligent behaviour in computers. John McCarthy coined the term in 1955 however AI has its roots in the Turing Machine invented by Alan Turing during the second world war to break the Nazi Enigma code, which he did successfully. AI is also found as far back as Ancient Greece where an ancient creature made of bronze called Talos protected Crete from pirates and raiders, or so the myth goes. The section of AI we are focusing on is a joining of problem solving and game theory. AI is widely used in the gaming industry to simulate humans playing the game, also know as “Computer players”. They employ AI techniques so the computer player can deduce what to do and when, in a lot of cases computer players can out perform a human player. AI techniques are also used to solve general problems however you can view games just as complex problems to be solved. In this case Sudoku suits this perfectly.

Sudoku despite the name was not originally invented by the Japanese and in fact has its roots in the late 19th Century France. The modern day Sudoku however was first published in an American magazine in 1979 but was popularised in Japan in 1986 where it gained its name. It was introduced to the UK mainstream in 2004 when the The Times started to publish the games in their papers. The game revolves around a 9*9 square board with 9 segments. The puzzle is to complete the grid so that all rows, columns and segments have the numbers 1 to 9 in them. The user is presented with a partially completed puzzle, a more difficult puzzle will have more blank spaces, and a well-posed puzzle will only have one solution.

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

An unsolved Sudoku puzzle

In this report you will see a lot of references to the C++ programming language and the QT library. I will present you with a quick background for these.

C++ first appeared in 1983 and is a very popular object-orientated programming language. It is popular because it allows low level memory management in comparison to other popular programming languages such as Java or Python. This means it allows you to manipulate the stored data more easily which gives you more control and flexibility however it is a double-edged sword as there are more possible problems and is therefore a little more difficult to program in. This causes it it be a little off putting to new developers. C++ is the industry standard for the professional game designing industry because it allows the programmers more control over the Graphics Processing Unit (GPU). C++ is also completely run from compiled code where as other languages such as Python are interpreted and Java being a hybrid of the two. This leads to its performance being quicker than most of the popular alternatives.

Qt (pronounced “cute”) is a ”leading cross-platform application and UI development framework for all the major desktop, embedded and mobile operating systems” [1]. It is a library for the C++ operating system and cross-platform which makes it perfect for this project as I am developing on OSx. The obvious choice would have been Apple's native Cocoa however it only runs on Mac which may have caused complications when people wanted to use the program on other platforms. QT's framework is very

big and very powerful and can employ techniques such as GUI creation, database access, thread management and network support. However for this project it was only used it to create the GUI. QT is queer because it uses a system of connecting signals and slots together, meaning when an object (E.G. a button) is clicked it will emit the click signal. You then connect this signal to a slot (a user defined function) so that when the signal is emitted the slot is called. This differs from many GUI frameworks which more usually use things such as ActionListeners. QT allows styling through the use of CSS, a web standard for styling which is great for those coming from web background to a mobile or desktop background.

Random Search algorithm is an optimisation method more generally used with a fitness function however in this situation one has not been implemented due to the small size of the puzzle and time constraints. The algorithm will fill the blanks of a Sudoku puzzle with random numbers within the given range. It will keep doing this until a solution is found. On larger problems it would be nearly impossible to solve alone without a fitness function because the chance of a correct game state would be extremely low. However in a 9*9 Sudoku puzzle it is an acceptable level depending on the difficulty.

The A* algorithm is a search based algorithm primarily used for path finding however is often adapted to other problems such as game solving. I chose to use it because I had used it in the past to solve games and I believed it could be adapted to solve Sudoku puzzles which has proven to be correct. Each loop of the search will take the current game state and create a list of all the possible actions that could be taken. In this case an example of that would be putting an 8 in point(1,2). It then generates states based on these actions and adds it to the queue to be analysed. The queue is ordered on a heuristic and cost. The optimality of the solution is down to this heuristic. However in Sudoku the solution is always optimal as there is a definite number of moves that can be taken therefore the heuristics function here is purely to speed up the search.

The brute force algorithm systematically goes through the puzzle checking every possible solution until it finds a valid solution. Something which would take a human a large amount of time to do but would only take a computer a second to complete. This specific brute force algorithm employs backtracking meaning as soon as a clash

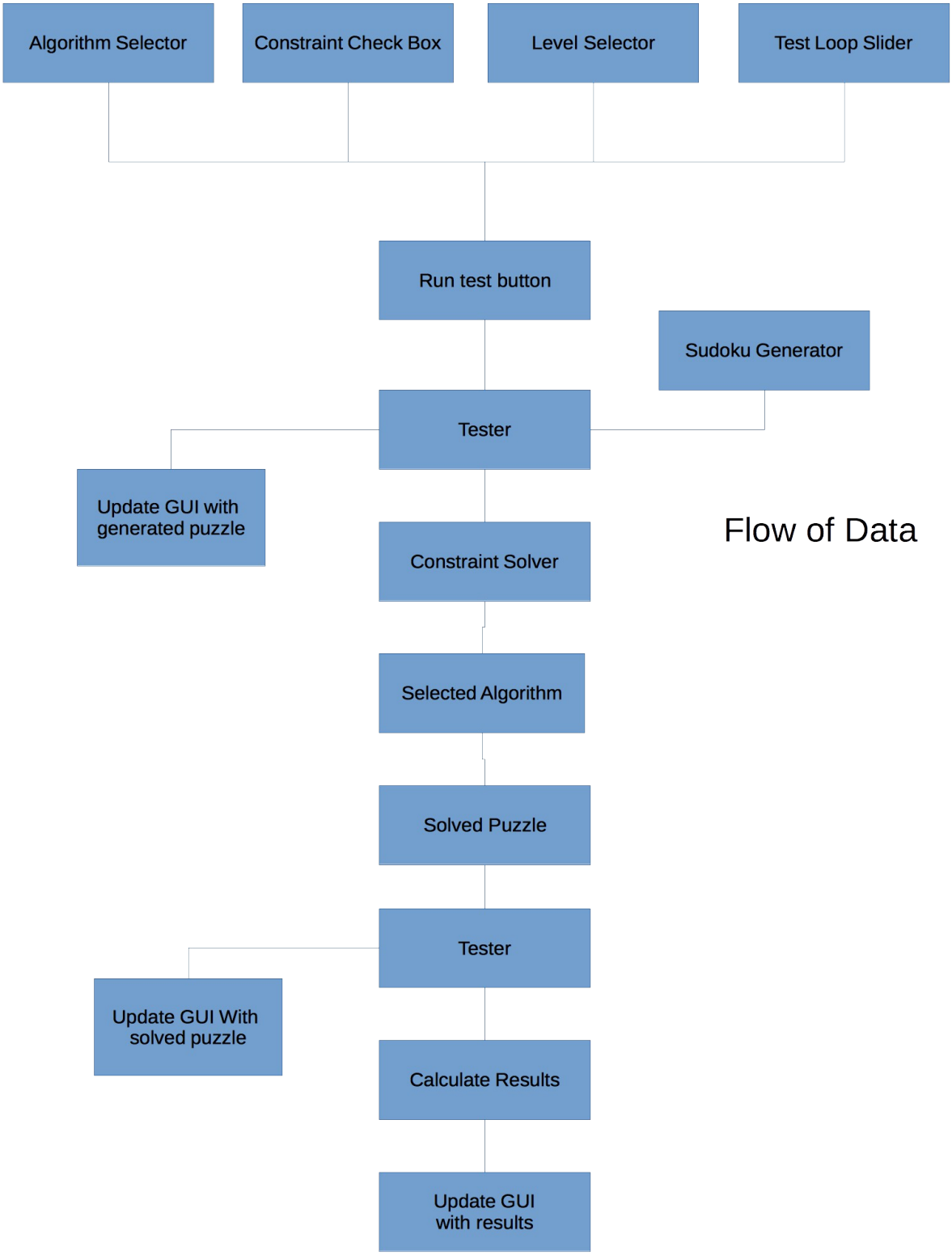
occurs in the puzzle (E.G. 1 in the same row twice) it will systematically go backwards through the points it has already checked incrementing their value until the clash is resolved before moving forward again.

The final algorithm I have used is one of my own making which I have dubbed as a Constraint Solver. It tries to solve the Sudoku puzzle logically as a human might by looking for definitives. I was looking here to see how a computer could process logic that humans can relatively quickly. I will release my findings later in the report. The constraint solver in the project was used as an aid to the other algorithms to reduce the number of blank spaces in the puzzle before they started however in some cases on the easier levels the constraint solver is able to solve it outright.

SPECIFICATION & DESIGN

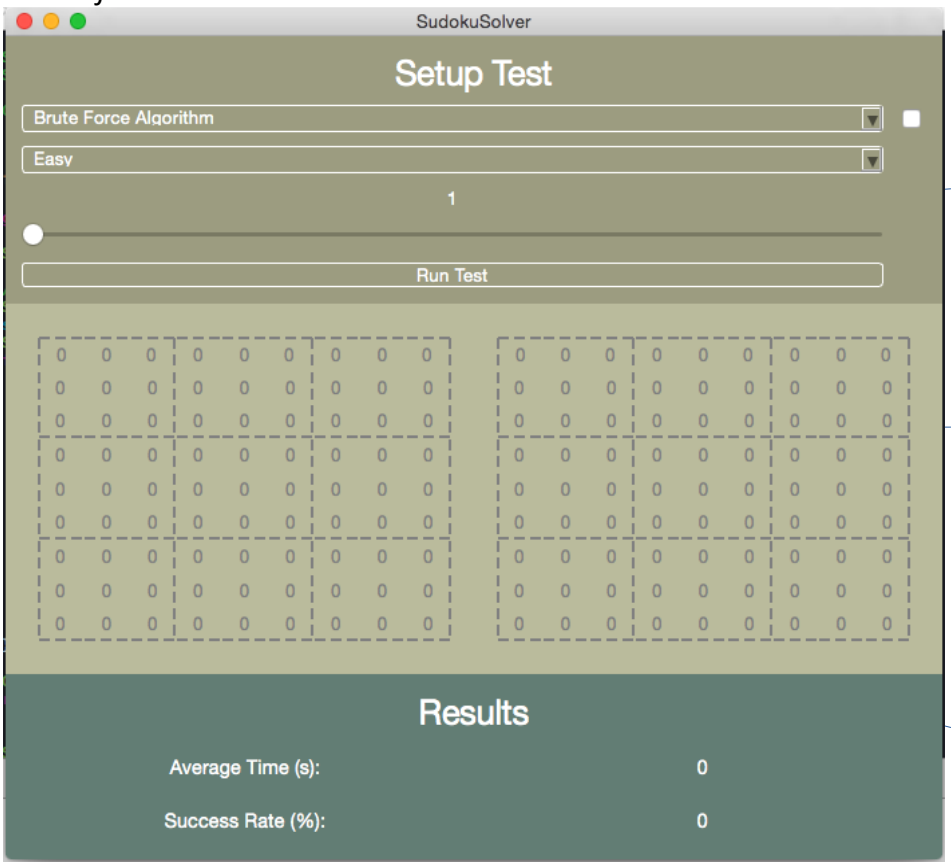
The software system allows users to pick an algorithm from a drop down menu and check whether or not they want to use the constraint solver. From there the user will select how many tests they want to perform (between 1 and 100) and select what difficulty they want to test the algorithm on from another drop down box. Once the user has selected all of their test options they then click the run test button. The system will then generate and solve sudoku puzzles to the specification defined by the user and display the puzzles on the GUI in the puzzle section. Once the test has finished running the average time and success rate of the algorithm will be displayed in the results section at the bottom of the GUI. From here the user may run another test if they wish or close the application.

On a code level, once the run button is clicked all of the options selected by the user are retrieved with get functions on the widgets. The tester class will then run one of three functions dependent on the algorithm selected. The system will then generate a puzzle and update the puzzle on the left of the GUI with the generated puzzle. If the constraint selector has been chosen this will run first to narrow down the search space as much as possible for the algorithm. The algorithm will then proceed to solve the puzzle if the constraint solver has not already. Once the puzzle is solved the tester will return it. If the puzzle is not solved the algorithm will time-out after 2 seconds to prevent the system getting stuck in an infinite loop, if this happens it is counted as a failure. This shows in the success rate result and a failed result is not taken into account when calculating the average time. In the case of a time out, the partly completed state is returned. Once the state is returned it is displayed in the GUI in the right puzzle. A loop sits outside all of this code and is set to the number of tests set by the user. On completion of all tests the average time and success rate are calculated and the GUI is updated in the results section with the respective results.

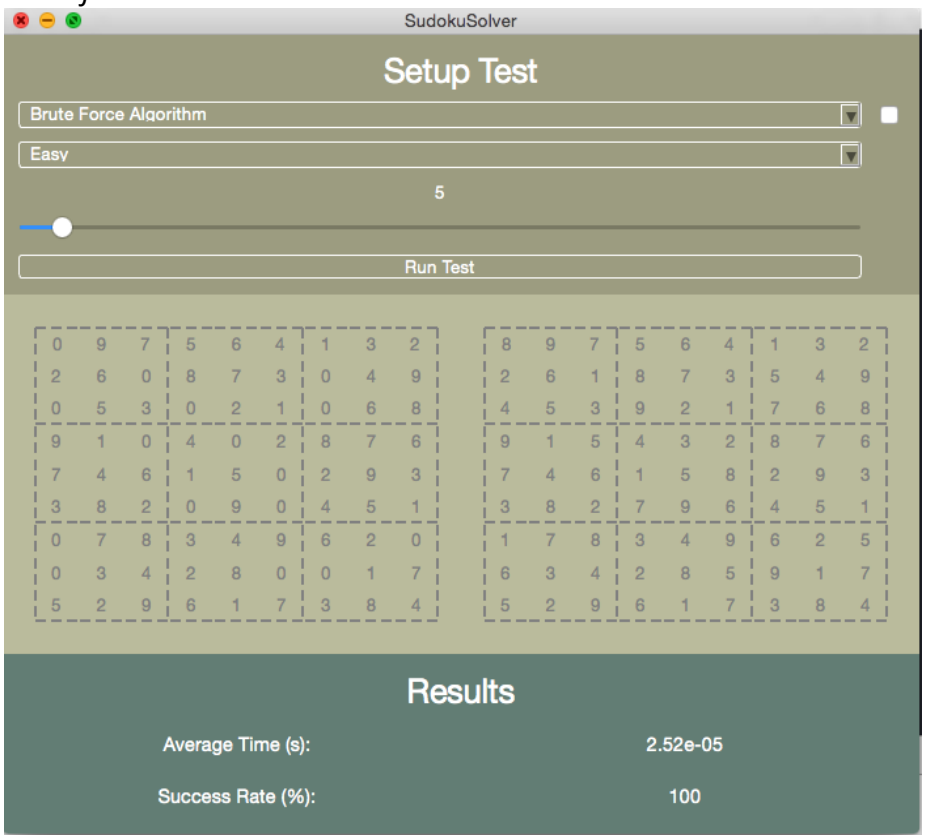

Algorithm Selector
Constraint Check Box
Level Selector
Test Loop Slider
Run test button
Sudoku Generator
Tester
Update GUI with generated puzzle
Constraint Solver
Selected Algorithm
Solved Puzzle
Tester
Update GUI With solved puzzle
Calculate Results
Update GUI with results

Flow of Data

GUI before test

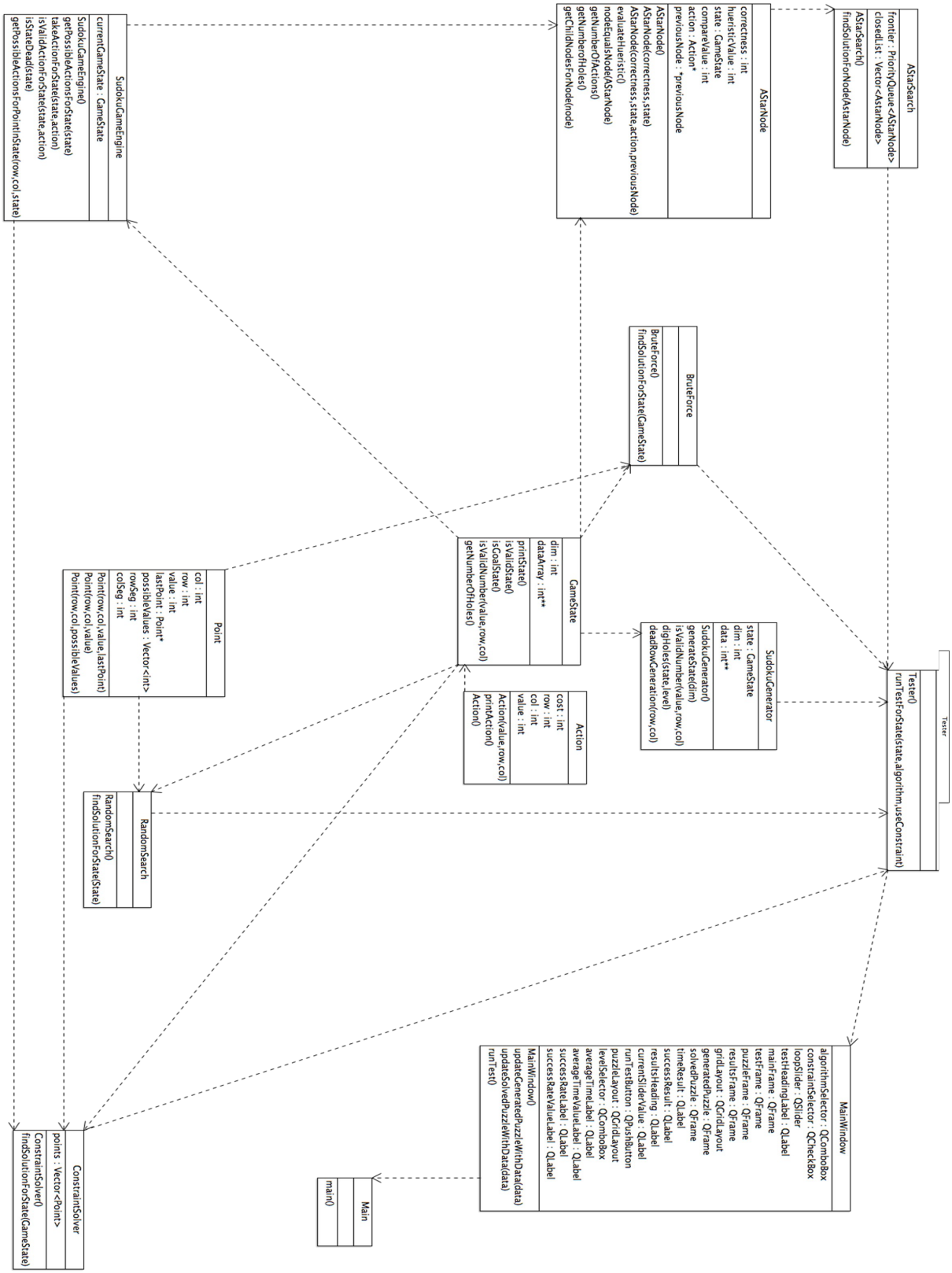


GUI after test



Simplicity was the big design feature that was implemented in the GUI's design. As you can see the window is split up into three sections to help break it up therefore making it easier for the user to find and use what they want. The layout used was a grid to keep everything clean and compact, there are no menus or other screens. Everything is kept on one screen to keep it simple. The GUI only has one function, to carry out tests so there is no need for any other screens. The GUI was originally designed so the user could play a Sudoku game as well as test the algorithms. However due to time constraints and its lack of relevance to the projects aim the design was scrapped. This also allowed the GUI to be kept more simple. The sections are also split by three colours to help define them. Colour is a good indicator in UI and as you can see I have used it to a good extent here. The colours were also specifically chosen to work well together and be aesthetically pleasing. The widgets for the test section (E.G. Button) were chosen based on there intuitiveness, so that users needed as little help as possible and there ability to fit well in the design. A good example of this is the slider for the number of tests to carry out, whilst intuitive you could argue using other kinds of widgets which require less room however I chose it because it fitted well in my design. The results section is very bare and only shows the two results we are interested in, this again is a purposeful choice to keep the GUI simple and clean.

Static architecture of code



There are 5 main algorithms that have been implemented in the system which I would like to outline the design for:

Sudoku Generator

```
def SudokuGenerator():
    For each square in puzzle:
        If deadRowGeneration():
            row--;
            col = 0;
        randNo = random % 9 + 1
        while (!isValidNumber(x))
            randNo = random % 9 +1
        square = randNo;
    digHoles(puzzle,level);
    return puzzle;
```

deadRowGeneration() checks if the row is valid for the grid.

IsValidNumber() checks if the number at that particular square is valid for the grid.

The algorithm goes through each square in the puzzle first checking if the row is currently valid (i.e. the values in it do not clash with the puzzle) and if it does revert back to the last row. If the row is fine it tries to fill the square with random numbers until one of them is valid for the puzzle. This takes longer than going through the numbers as you may try one number several times before you get another valid, however it is to make sure the puzzle is unique.

```
def digHoles(puzzle,level):
    for each square in puzzle:
        int chance = random;
        if chance < level:
            square = 0;
    return puzzle;
```

Once all squares are full the algorithm digs holes in the puzzle using chance, the chance changes with the level to get a scale.

Brute Force

```
def findSolutionForState(puzzle):
        list emptySquares;
        for each square in puzzle:
                if square == 0:
                        emptySquares.add(square)


        for square in emptySquares:
                for (n = square.getValue; n < 9)
                        if(isValidNumberForState(n))
                                square = n;
                                break;
                        else if(n==9)
                                square = 1;
                                go back a square;
```

The algorithm creates a list of all the empty squares and then loops through them. It takes the squares current value and loops through up to 9 until a valid number is found. When a valid number is found the square's current value is set to said number and the algorithm moves onto the next square. If no valid number is found the algorithm sets the current square to the previous square and continues. This is known as backtracking.

A Algorithm*

```
def findSolutionForState(state):
        Node  node = node(State)
        frontier.add(node)
        while(!frontier.empty()):
                node = frontier.top()
                if (node.isGoal())
                        return node;
                else:
                        closedList.add(node);
                if(node.isDeadState())
                        continue;
                if(closedList.contains(node))
                        continue;
                list children = node.getChildren();
                for child in children:
                        if(closedList.contains(child))
                                continue;
                        froniter.add(child);
```

isDeadState() checks to see if the current state is invalid and therefore taking any more actions is wasteful. It does this by making sure there is at least one action to be taken for every empty square.

IsGoalState() checks if the puzzle is completed.

This algorithm implements a priority queue as the frontier so nodes can be ordered depending on how they score against a heuristic.

The algorithm takes the state and turns it into a node and then adds said node to the frontier. It then begins a loop which won't exit unless the frontier is empty or a goal state is found. The algorithm first checks if the current node, taken from the front of the frontier, is the goal state. If it is, it returns the puzzle and if not it adds the current state to the closed list so it is not checked again. Next it checks if the current node is a dead state. If so, it starts the loop again with the next node in the frontier. Then the algorithm checks if the node is already in the closed list and if so it skips to the next node in the frontier. If the node passes all of these checks the algorithm gets all of it's

children. It's children being the states produced by all of the possible actions which could have been taken by the state. These children are added to the frontier.

The A* algorithm usually uses a cost function along with a heuristic so going further down the tree works against the algorithm however this is redundant in Sudoku because there is a definite number of moves to be taken, that being the number of holes that need filling. Therefore in the algorithm I replaced cost with another score that I called “probability correctness”. I figured out that when placing a number in a hole, holes which have less possible values in them are more likely to be correct when you place one of the possible values in it. I scored this in each node of the algorithm and therefore states which are more likely to be correct are prioritised and the goal is found quicker.

Random Search

```
findSolutionForState(puzzle):
    list emptySquares;
    for square in puzzle:
        if square.isEmpty()
            emptySquares.add(square)
    while(!puzzle.isGoalState())
        for square in emptySquares
            square = random(1 < x < 9)
    return puzzle
```

Like the brute force algorithm the random search algorithm first makes a list of all the empty squares in the puzzle. It then loops through the empty squares filling them with random numbers between 1 and 9. It keeps repeating this process until the puzzle is solved.

Constraint Solver

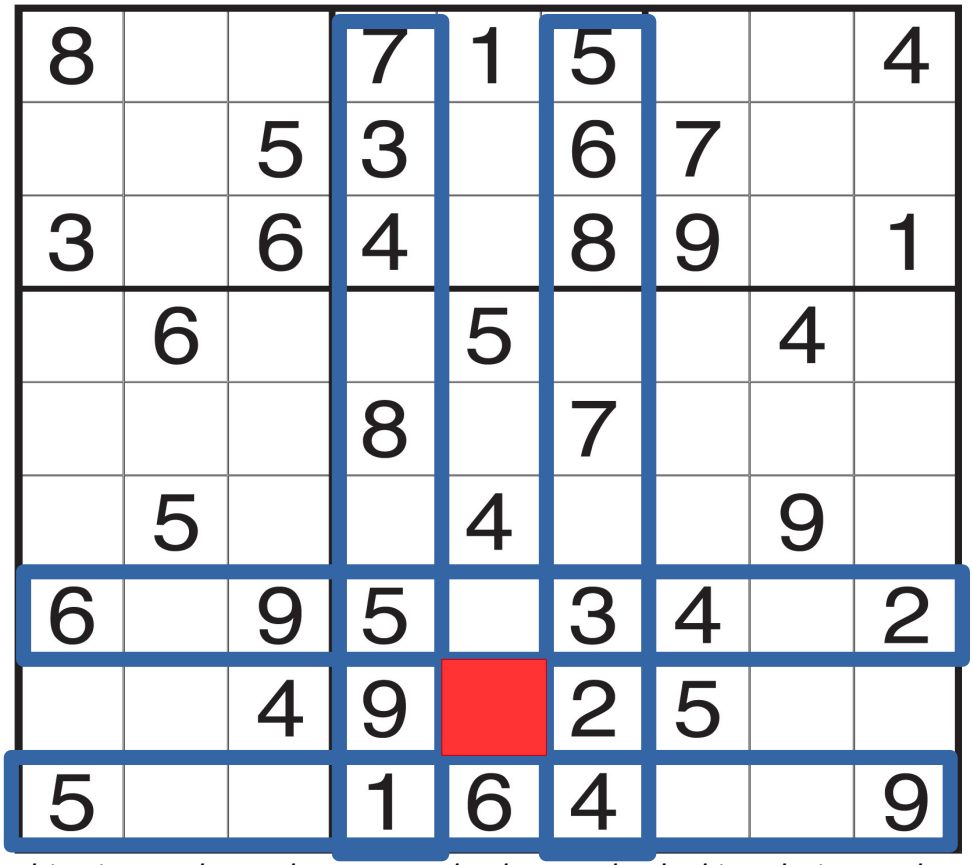
```
def findDefinitesForState(puzzle):
        for square in puzzle:
                if(!square.isEmpty()) continue;
                if(square.isLastHoleInSegment()):
                        for ( 0 < x < 10)
                                if(!segmentContaintsValue(x)):
                                        square = x;
                        continue
                if(square.isLastHoleInRow()):
                        for ( 0 < x < 10)
                                if(!rowContaintsValue(x)):
                                        square = x;
                        continue;
                if(square.isLastHoleInCol()):
                        for ( 0 < x < 10)
                                if(!colContaintsValue(x)):
                                        square = x;
                        continue;
                for ( 0 < x < 10)
                        if(isValueDefiniteForState(x))
                                square = x;
        return puzzle;
```

The algorithm loops through all of the empty squares. It then checks if the square is the last empty square in the segment and if so loops through and finds the missing value. It does the same for the row and column and once the value is found moves on to the next empty square. It then calls a function to check if the other values in the puzzle cause the value in the current square to be definitely decided.

```
def isValueDefiniteForState(puzzle,value,squarel):
        if segmentContainsValue(value):
                return false;
        for square get adjoining rows:
                if !rowContainsValue(x)
                        return false
        for square get adjoining cols:
                if !colContainsValue(x)
                        return false


        return true;
```

This function is designed to check the two other rows and columns according to the squares position in a segment for the value defined. If the value is in both rows and both columns and not in the segment the value is definitely in the current square. This is meant to replicate how a human often approaches a puzzle. This algorithm has a lot of room for expansion.



This picture shows the rows and columns checked in relation to the square.

IMPLEMENTATION

Below are pieces of code that are critical to the system or may be of particular interest. Each has been described in detail as well as remarking upon any issues that were faced.

This piece of code is an implementation in MainWindow.cpp and updates the sudoku board with new data.

```
void MainWindow::updateSolvedPuzzleWithData(int** data){
    QGridLayout *solvedBoardLayout = new QGridLayout();

    QWidget().setLayout(solvedPuzzle->layout());
    solvedPuzzle->setLayout(solvedBoardLayout);
    solvedBoardLayout->setSpacing(0);
    for (int i = 0; i < 9; i++){
        for (int j = 0; j < 9; j++){
            QLabel *number = new QLabel();
            number->setNum(data[i][j]);
            solvedBoardLayout->addWidget(number,i,j);
            number->setAlignment(Qt::AlignCenter);
            styleSudokuPuzzleForPoint(i, j, number);
        }
    }
}
```

The trouble I found was replacing the labels with numbers as I had not stored the labels as class variables. I didn't do that because I believed it was a bit redundant to save 81 QLabels within the class as you would have to loop through them anyway so I thought it easier to just update the board with new Qlabels.

StyleSudokuPuzzleForPoint() is a function which styles the Sudoku board in the main window. I found it easier to style the points individually as they were created rather than to pass the whole board into a function after. For me this was cleaner as it meant the whole board was created in the scope of the for loops.

```
void MainWindow::styleSudokuPuzzleForPoint(int i, int j, QLabel *number)
{

    QString style = "color: grey;";
    if(!i) style += "border-top: 2px dashed grey;";
    if(!j) style += "border-left: 2px dashed grey;";
    if (j == 8) style+= "border-right: 2px dashed grey;";
    if (i == 8) style += "border-bottom: 2px dashed grey;";
    if(j == 2 || j == 5) style += "border-right: 2px dashed grey;";
    if (i == 2 || i == 5) style += "border-bottom: 2px dashed grey;";

    number->setStyleSheet(style);
```

```
}
```
The Sudoku Generator which I described in pseudocode in the design section of this report was implemented as specified below.

```
GameState SudokuGenerator::GenerateState(int dim){

    //Generate random full board
    SudokuGenerator::dim = dim;
    srand (time(NULL));
    SudokuGenerator::data = new int *[dim];
    for (int i = 0; i < dim; i++) {
        data[i] = new int[dim];
        for (int j = 0; j < dim; j++) {
            if(deadRowGeneration(i, j)){
                i--;
                j= 0;
            }
            int randNo = rand() % 9 +1;
            int x = 0;
            while(!isValidNumber(randNo, i, j) && x < 20){
                randNo = rand() % 9 +1;
                x++;
            }
            data[i][j] = randNo;

        }
    }

    GameState *generatedState = new GameState(data, dim);
```

The differences made between design and implementation being that one the algorithm here returns a GameState rather than the data itself. Also it returns a completed puzzle. The holes are dug outside of the function. This was done purely for flexibility during implementation and testing of the code, having a completed version of the puzzle for comparison was handy. Also in the implementation if a random number is not found to be valid after 20 tries the algorithm gives up and therefore a deadRowGeneration was called. This was to stop the algorithm getting caught in an infinite loop. This was the first of the main algorithms to be implemented and by the end of the project I had a few ideas of how to improve it however I had no time. I believed using back tracking, which is used in the BruteForce algorithm, would be more efficient than checking to see if a whole row is dead and starting it again as it is quicker and simpler as you are not losing a load of filled holes in the process. Despite this the algorithm implemented works great for the system and it is the solving algorithms that are being timed and not this so any speed up is for convenience only.

When implementing the A* Algorithm it is important to know when a state is dead and further expanding it will bring no results. Therefore in SudokuGameEngine.h I had created a function to check for such dead states.

```
bool SudokuGameEngine::isStateDead(GameState state) {
    for (int i = 0; i < state.getDim(); i++) {
        for (int j = 0; j < state.getDim(); j++) {
            if (!state.getData()[i][j]) {
                vector<Action> possibleActions =
getPossibleActionsForPointinState(i, j, state);
                if (possibleActions.size() == 0)
                    return true;
            }
        }
    }

    return false;
}
```

It counts the number of actions available for a certain point and if there are no actions available then that state must be dead because you could never fill the hole.

The brute force algorithm was implemented as follows.

```
GameState BruteForce::findSolutionForState(GameState state) {

    std::clock_t start = clock();
    double duration;

    SudokuGameEngine engine = SudokuGameEngine();
    vector<Point> points;
    for (int i = 0; i < state.getDim(); i++)
        for(int j = 0; j < state.getDim(); j++)
            if(!state.getData()[i][j])
                points.push_back(Point(i,j,1));

    for (int i = 0; i < points.size();i++) {
        duration = ( std::clock() - start ) / (double) CLOCKS_PER_SEC;
        if (duration > 2) return state;
        Point *currentPoint = &points.at(i);
        for(int n = currentPoint->getValue(); n <= 9; n++) {

            //Create Action
            Action action = Action(n,currentPoint-
>getRow(),currentPoint->getCol());

            //If valid number edit state accordingly and move on to next
point
            if(engine.isValidActionForState(action, state)) {
                state.getData()[currentPoint->getRow()][currentPoint-
>getCol()] = n;
                currentPoint->setValue(n);
                break;
            }
            //If no valid number move back to the last point
            else if(n == 9) {
                i -= 2;
                currentPoint->setValue(1);
                state.getData()[currentPoint->getRow()][currentPoint-
>getCol()] = 0;
            }
        }

    }

    return state;
}
```

The brute-force algorithm follows the specification exactly in terms of layout, only data structures and syntax change. I did not implement a square class instead I implemented a Point class, which is essentially the same thing. One difference between design and implementation is when n== 9 I decrement I by 2. This is essentially going back two squares where the algorithm only said one. This is because after, it increments by 1 when the loop finishes anyway so it is 2 to compensate for this change. The addition of the clock is to handle time outs. This was added in the testing stage and was not a part of the design. The algorithm works fantastically.

```
AStarNode AStarSearch::findSolutionForNode(AStarNode node) {

    std::clock_t start = clock();
    double duration;


    SudokuGameEngine engine = SudokuGameEngine();

    frontier.push(node);

    while (!frontier.empty()) {


        AStarNode node = frontier.top();
        frontier.pop();
        duration = ( std::clock() - start ) / (double) CLOCKS_PER_SEC;
        if (duration > 2) return node;


        if(node.getState().isGoalState())
            return node;
        else
            closedList.push_back(node);

        //Dead State
        if(engine.isStateDead(node.getState())) {
            closedList.push_back(node);
            continue;
        }


        // Implement multiple path pruning
        for (AStarNode closedNode : closedList)
            if (node.nodeEqualsNode(closedNode))
                continue;

//        cout << frontier.size();
//        cout << "\n";

        vector<AStarNode> children = node.getChildNodesForNode(node);

        for (AStarNode node : children) {
            bool inClosedList = false;
            for (AStarNode closedNode : closedList)
                if (node.nodeEqualsNode(closedNode))
                    inClosedList = true;
                if (!inClosedList)
                    frontier.push(node);
            }


    }

    return node;
}
```

The algorithm printed of the previous page is the A* Search algorithm. Excluding the clock and syntax changes due to the language vs the Pseudocode the algorithm is exactly the same as specified in the design. The dead state algorithm I specified earlier is implemented here as well as Multiple Path Pruning (MPP). MPP involves keeping a closed list of the states already expanded and therefore prevents them from being checked again. It is more efficient than its alternative cyclic checking which prevents the algorithm from getting into a cycle however it does require more memory, the trade off here was worth it as it speeds up the algorithm dramatically. I had major problems trying to compare the current node to those in the closed list as the vector class does not support this by default this unlike other languages like Java. So instead I had to write my own function which loops through the data of each node and compares them for differences.

```
GameState RandomSearch::findSolutionForState(GameState state) {
    srand (time(NULL));
    vector<Point> points;
    std::clock_t start = clock();
    double duration;
    for (int i = 0; i < state.getDim(); i++)
        for(int j = 0; j < state.getDim(); j++)
            if(!state.getData()[i][j])
                points.push_back(Point(i,j,0));
    while(!state.isGoalState()) {
        //Generate a random completed state from the param state
        duration = ( std::clock() - start ) / (double) CLOCKS_PER_SEC;
        if (duration > 2) return state;

        for(Point point : points)
                if(!state.isValidNumber(state.getData()[point.getRow()]
[point.getCol()], point.getRow(), point.getCol())) {
                    int randNo = rand() % 9 +1;

                    state.getData()[point.getRow()][point.getCol()] =
randNo;
                }
    }


    return state;
}
```

Like all of the other algorithms above I have mentioned, Random Search is also implemented to the specified design above with differences being between syntax and data structures. The only main difference being that on every loop the number is only changed if it is invalid. This does not necessarily guarantee it is correct as a number placed later in the puzzle can make it incorrect however it means it has a slightly better chance of being correct and therefore speeds up the algorithm.

```
bool ConstraintSolver::valueIsDefiniteForPointInState(int value, Point
point, GameState state){
    SudokuGameEngine engine = SudokuGameEngine();
    Action action = Action(value, point.getRow(), point.getCol());
    if(!engine.isValidActionForState(action, state))
        return false;


    if (point.getRow() == 0 ||point.getRow() == 3 ||point.getRow() == 6)
{


        if(!rowContainsValueForState(point.getRow()+1, value, state,
true) ||
           !rowContainsValueForState(point.getRow()+2, value, state,
true))
            return false;
        else if (!state.getData()[point.getRow()+1][point.getCol()] &&
                 !state.getData()[point.getRow()+2][point.getCol()])
            return true;


    }
    else if(point.getRow() == 1 ||point.getRow() == 4 ||point.getRow()
== 7 ) {
        if(!rowContainsValueForState(point.getRow()-1, value, state,
true) ||
           !rowContainsValueForState(point.getRow()+1, value, state,
true))
            return false;
        else if (!state.getData()[point.getRow()+1][point.getCol()] &&
                !state.getData()[point.getRow()-1][point.getCol()])
                return true;
    }
    else {
        if(!rowContainsValueForState(point.getRow()-1, value, state,
true) ||
            !rowContainsValueForState(point.getRow()-2, value, state,
true))
            return false;
        else if (!state.getData()[point.getRow()-1][point.getCol()] &&
                 !state.getData()[point.getRow()-2][point.getCol()])
                    return true;
```

```
    }


    if (point.getCol() == 0 ||point.getCol() == 3 ||point.getCol() == 6)
{
        if(!colContainsValueForState(point.getCol()+1, value, state,
true) ||
           !colContainsValueForState(point.getCol()+2, value, state,
true))
            return false;
        else if (!state.getData()[point.getRow()][point.getCol()+1] &&
                 !state.getData()[point.getRow()][point.getCol()+2])
                   return true;


    }
    else if(point.getCol() == 1 ||point.getCol() == 4 ||point.getCol()
== 7 ) {
        if(!colContainsValueForState(point.getCol()-1, value, state,
true) ||
           !colContainsValueForState(point.getCol()+1, value, state,
true))
            return false;
        else if (!state.getData()[point.getRow()][point.getCol()-1] &&
                !state.getData()[point.getRow()][point.getCol()+1])
                    return true;
    }
    else {
        if(!colContainsValueForState(point.getCol()-1, value, state,
true) ||
           !colContainsValueForState(point.getCol()-2, value, state,
true))
            return false;
        else if (!state.getData()[point.getRow()][point.getCol()-1] &&
                !state.getData()[point.getRow()][point.getCol()-2])
                    return true;
    }
    return true;
}
```

The algorithm described on the previous two pages is in the main implementation of the constraint solver. I have not included every function necessary as it would take up ten pages. This brings me to an interesting point about this algorithm its size and complexity for something that humans can compute so simply. In comparison to the brute force algorithm which is small and simple but very repetitive and slow for humans to do, this algorithm is large and complex but easier for humans to carry out. This to me draws a very big comparison between humans and computers and shows a giant barrier in term of artificial intelligence. This kind of comparison is not unique to this problem but is seen everywhere across the artificial community. To completely replicate human logic and thinking patterns in a computer is extremely difficult. Especially when you take into account that this is only a simple problem, if you scale that up for example to robotics it because even harder. However you could implement brute force and gain the same outcome but the question is, if a human and computer reach the same outcome but by different means is it really the same?

This algorithm checks the two corresponding rows and columns in a segment as described in design for a value. If all 4 contain it but the segment does not then that hole must contain that value. The algorithm is a huge if statement because it requires two variables the row and column position which makes the use of a cleaner if statement impossible as they only take one parameter. I could have nested switch statements but that would have been even messier. This was tough to implement because testing how it was working was particularly tough as there was no easy way to track the algorithm to easily debug it.

There were a few problems that I came across whilst implementing the system. The first was arrays in C++. Since the data for the puzzle is kept in a 2d-array I had to implement a double pointer int. Having not used C++ to a great extent in over a year and not having a great deal of experience with it in the first place I found it difficult to begin with to use these. However this is why I chose to use C++, to get this kind of experience.

Another problem I ran into was the GUI. I have experience in Cocoa but could not use it because it was not cross-platform so I decided to pick up QT. This was a library I had known of but had no experience with. Also the majority of tutorials involved using its IDE where the GUI's were made with a drag and drop interface. However I wanted to do it programmatically as I believed this is better practice for learning, I also didn't want to have to create a new project and move all my files over at the risk of creating errors which would take ages to fix. The pay off was not that great though as it took a while to learn the basics of QT especially as it has it's own compiler which was very

difficult to integrate into xCode. After I got through all of this I found out that I had an old copy of QT and had to re-download the whole library which is massive and I had poor internet so this took the best part of the day and in the meantime there wasn't much I could do. In the end I got it working and I am better off from having persevered through it.

I had two unnecessary and overambitious project aims I did not implement in the end for those reasons:

1. To implement the ability for a human to play a sudoku puzzle. Whilst a great learning experience and nice feature it had very little to do with comparing algorithms, so I chose not to implement it.
2. The other was the ability to show the puzzle being solved step by step. Whilst a nice feature which has relevance for testing and presentation within the project, it was over all unnecessary. It would have also required slowing the algorithms right down to human speed or saving the steps and going through them. In the end there was just not enough time to implement this.

I would have liked to improve a few of the algorithms, especially the constraint solver as it has a lot of room to grow, and possibly implement another algorithm however there was not enough time for this, so I focused on what I had which I think is more than enough on its own regardless.

RESULTS AND EVALUATION

I am proud to say that the experiment was a success and that all algorithms functioned. Below are the results I have gotten from my experiments. The algorithms were set to solve 100 puzzles with a timeout time of 2 seconds. This was to give enough time for the algorithm to feasibly solve the puzzle. If it could not solve the puzzle in the given time it had deemed to fail and the algorithm would quit to avoid the system hanging (getting stuck). The only algorithm this was unfair on was random search because theoretically given infinite amount of time the random search will eventually find the solution by chance however we were not interested in if the algorithm could solve it in two days we want to compare performances. Therefore any test taking over 2 seconds was deemed too long and therefore counted as a fail.

Each algorithm was set to run 100 tests. I chose 100 because it was enough to create reliable results but not so many as to risk the computer hanging and not responding which would alter the results. If I had a more powerful computer at hand to test it on, I may have risked more tests for more reliable results. I also had time restraints and could not risk waiting hours for results. 100 is also a nice number to draw statistics from.

The algorithms were tested across 5 levels to get a nice wide range of results for comparison. How the generation algorithm works may mean an impossible puzzle on the easy side may be very similar to an insane puzzle on the hard side, this was done on purpose to produce a nice range like I said.

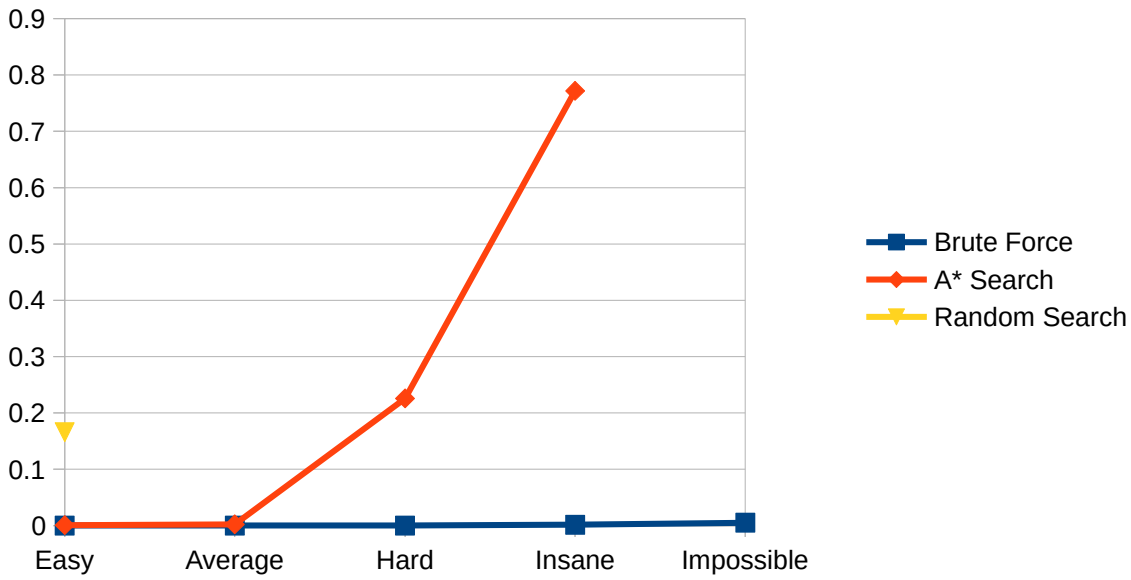
Table showing average time

Algorithm		Average Time to solve puzzle level (seconds)				
		Easy	Average	Hard	Insane	Impossible
Brute Force	Using Constraint Solver	5.988e-05	0.0001327 8	0.0001787 4	0.0020390 2	0.0084185 2
	Not using Constraint Solver	3.01e-05s	4.456e-05	6.547e-05	0.0015561 1	0.0051022
A* Search	Using Constraint Solver	5.516e-05	0.0212649	0.0834358	Failed	Failed
	Not Using Constraint Solver	0.0008666 4	0.0023630 7	0.225843	0.771866	Failed
Random Search	Using Constraint Solver	0.0007370 5	0.0823296	Failed	Failed	Failed
	Not Using Constraint Solver	0.165981	Failed	Failed	Failed	Failed

Table showing success rate

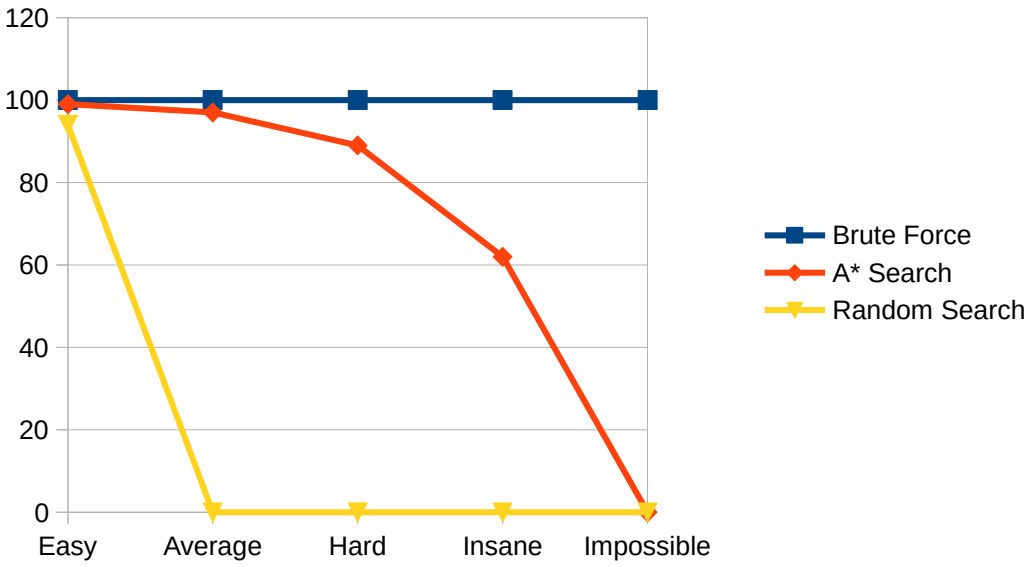
Algorithm		Success rate (%) over 100 puzzles				
		Easy	Average	Hard	Insane	Impossible
Brute Force	Using Constraint Solver	100	91	52	87	100
	Not using Constraint Solver	100	100	100	100	100
A* Search	Using Constraint Solver	100	92	75	0	0
	Not Using Constraint Solver	99	97	89	62	0
Random Search	Using Constraint Solver	100	97	0	0	0
	Not Using Constraint Solver	94	0	0	0	0

A graph detailing the algorithms time across varying difficulty of puzzles (Not using the constraint solver)



This graph details the results to our main objective *"to implement and compare various algorithm's performance at solving sudoku puzzles".* As we can see from the graph the brute force algorithm is superior by a huge margin to the A* algorithm and random search. As the difficulty of the puzzles increase, all of the algorithms see an increase in the time it takes to solve said puzzles. There is an exception with the random search algorithm which fails to solve any more than the easiest puzzles which suggests it follows the same trend to an extreme.

A graph detailing the success rate of algorithms across varying difficultly (Not using the constraint solver)



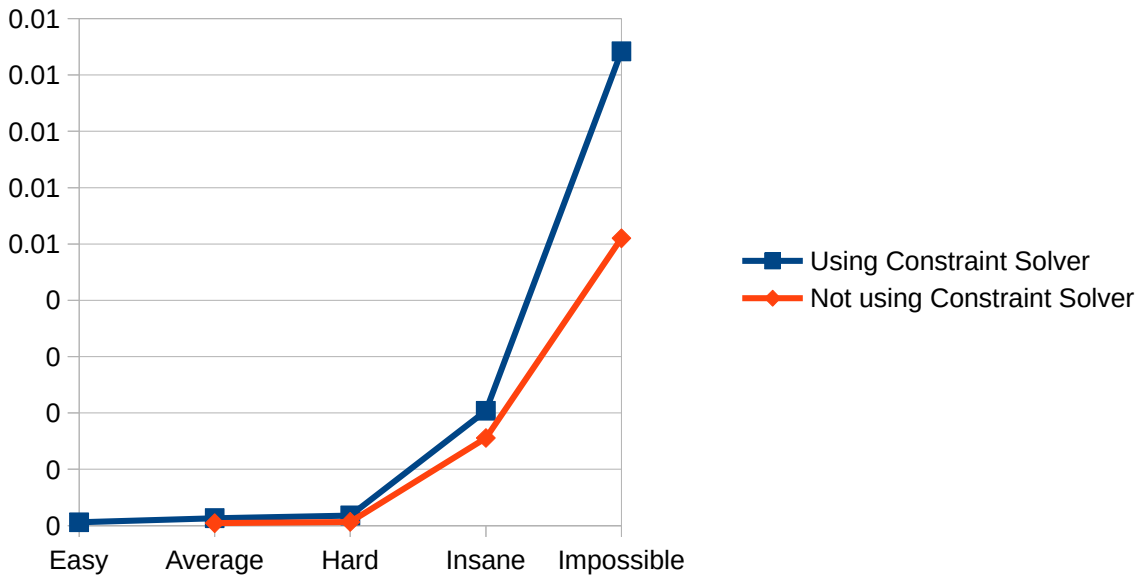
As you can see from the graph the brute force algorithm is 100% successful at all difficulties, with the A* search algorithm being less successful and failing to solve the impossible puzzles. The random search puzzle is even less successful, only being able to solve the easiest of puzzles. This means that the brute force algorithm is overall the most successful algorithm.

From the two graphs above you can see that in a comparison of the 3 algorithms above that they rank in the following order:

1. Brute force
2. A* Search
3. Random Search

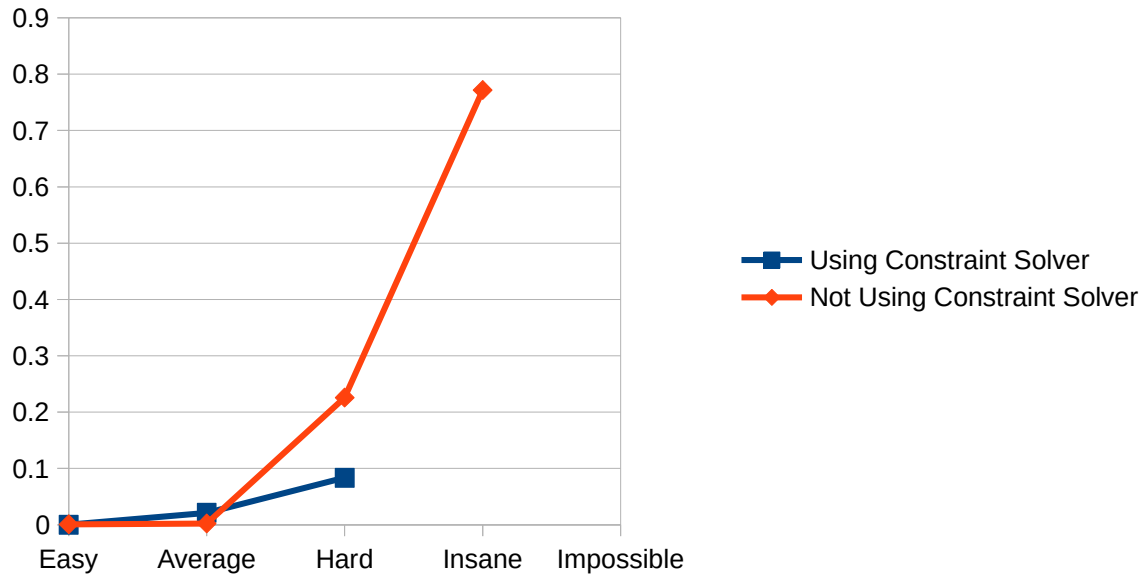
This makes the brute force algorithm the best, out of the ones we have tested, for solving 9*9 sudoku puzzles.

A graph detailing the effect of the constraint solver on the average speed of the brute force algorithm



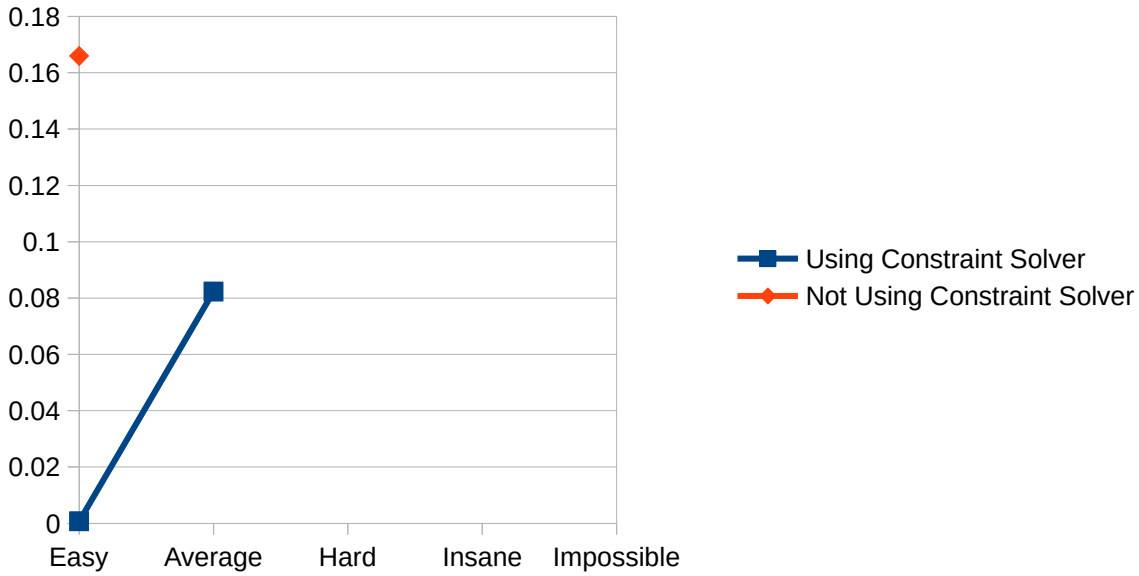
As you can see from this graph using the constraint solver hinders the brute force algorithm making it slower.

A graph detailing the effect of the constraint solver on the average speed of the A search algorithm*



As you can see from the graph using the constraint solver seems to lesson the impact of increasing the puzzle difficulty on the algorithm. However due to it taking longer on average puzzles and not being able to complete the insane puzzles compared to not using it, it is very hard to draw any meaningful conclusions from this data.

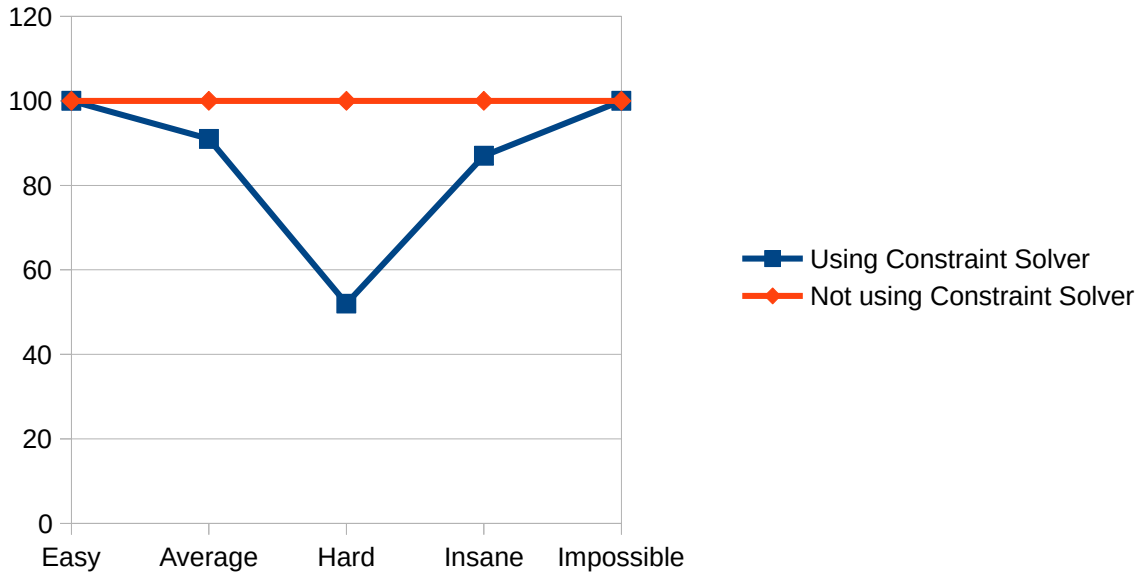
A graph detailing the effect of the constraint solver on the average speed of the random search algorithm



As you can see from the graph the constraint solver has a positive effect on the use of the random search algorithm. It performs quicker and is even able to solve average puzzles where as it could not without it.

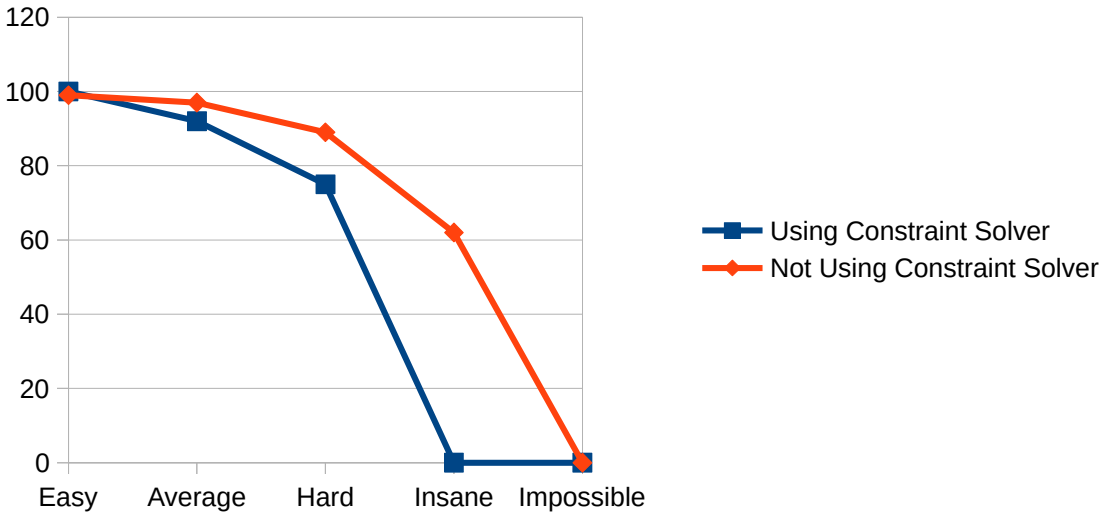
Having completed these tests, there are a few things I would do like to remark on or do differently if the test was to be repeated. The first being to test the constraint solver as a stand alone algorithm rather than one assisting the others. The reason for this, Is because it can actually solve the easiest puzzles on its own and I believe if the algorithm were to be expanded and improved it could solve harder puzzles and would make for a good comparison along with the others. I would also increase the number of tests to 1000 to create more reliable results. Whilst I believe 100 is reliable enough as it is, given the time and a stronger computer I would of liked to make the results more reliable.

A graph detailing the effect of the constraint solver on the success rate of the brute force algorithm



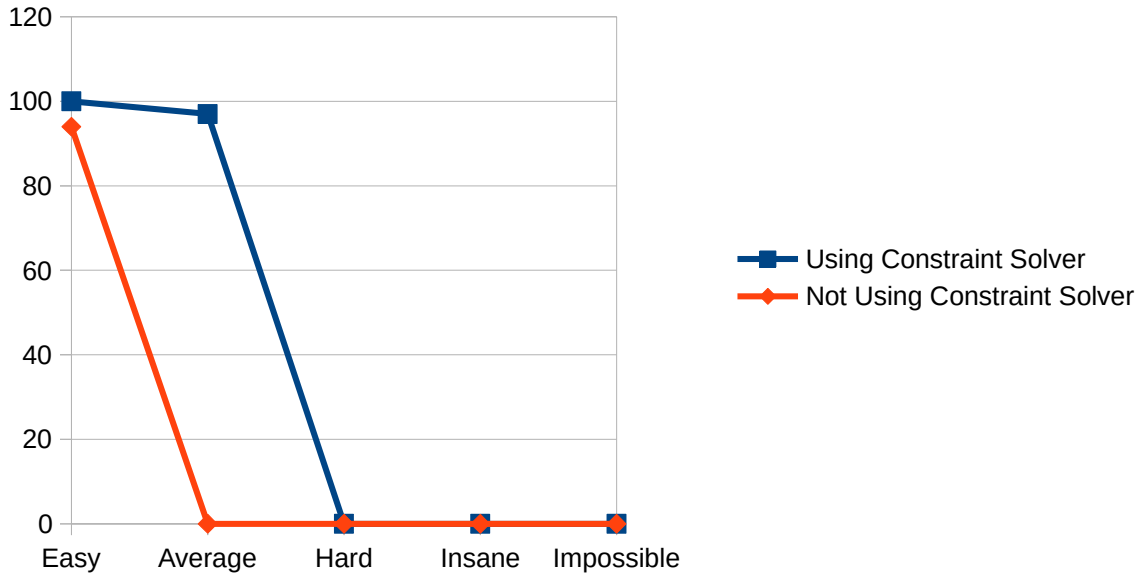
The graph clearly shows that on all of the easy and impossible levels the constraint solver had no effect on the success rate of the algorithm but on all other levels it reduces the success rate. Due to the pattern of these results, I can not come up with any reasons as to why they are this way. More tests could produce some more results which would shed some more light on the mater. Over all the constraint solver generally decreases the success rate of the brute force algorithm.

A graph detailing the effect of the constraint solver on the success rate of the A search algorithm*



This graph shows a definite trend in the data. Using and not using the constraint solver has no difference on the trend of the success rate. The difference being is that the success rate is lower when using the constraint solver. The constraint solver is therefore hindering the performance of the A* search algorithm.

A graph detailing the effect of the constraint solver on the success rate of the random search algorithm



Now this graph gives contradicting data to what we have seen so far in that using the constraint solver actually increases the performance of the algorithm. It even allows it solve the hard level puzzles. This is probably due to the algorithm reducing the number of points to be solved to a level the random search can handle.

Overall I think of the project as a success. It compares the algorithms successfully, draws comparisons between the ability of humans and computers and includes a GUI for other people except myself to use. The project goes above and beyond what was required with it and gives a rich set of results and thoughts on the forefront of AI. Having said that, there are always things that could be done to improve it. The biggest improvement I would like to have done to the project was to introduce a fitness function into the random search algorithm which would hopefully increase its performance creating a better comparison between itself and the other algorithms.

Another would be to improve upon the current constraint solver algorithm and see just how far you could take it. This would allow for more comparisons to be made between human logic and computers ability to replicate it.

FUTURE WORK

During this project, I did not achieve implementing the ability for a player to play Sudoku using the GUI and neither did I implement the ability for the user to watch the puzzle be solved in slow motion. However in my opinion these are not very important to the overall project aim and therefore they are only suggested extras for future work.

One feature I am very disappointed I did not implement was threads in the GUI. As it stands you do not see all the generated and solved puzzles as the GUI does not update until all of the tests are complete. The code is not ordered this way but threading would be required to update the GUI as the function is run. I did make a quick attempt to thread the GUI however one method required me to update my compiler and reconfigure xcode to work with it so I could include I certain header. Not explicitly hard but it was late into the project and I was not going to risk losing anymore time unnecessarily. The other was to use the QThreads, which are part of the QT library, I did attempt this quickly however how I had laid out my code made it difficult to work with QThreads so I decided not to waste any more time on them however in the future this is something I would love to do. This was a disadvantage to me using C++ as I was no stranger to using threads with the swing GUI library in Java.

A certain constraint in my experiment is that I only experimenting with a 9*9 sudoku board (the classic size). This is to keep the timing of the experiments low and true to the game itself however we may find algorithms may perform worse or better across larger boards. For example I would predict A* search and Random search to perform much worse across a larger board as the search space would be much bigger and it would therefore be harder to find the solution. I would not believe that brute force or the constraint solver would be extremely effected by a bigger board. However this is only speculation and can be left to a future experiment.

Throughout the project it my mind slowly changed to believe that implementing a fitness function into the random search algorithm would be more beneficial. This was mostly due to seeing its weak results in comparison to the others and felt that without the fitness function I was not doing the algorithm justice. However due to time

restraints this was not implemented. In the future it would be great to implement a fitness function and see how it's performance is affected. In fact an interesting experiment might be comparing different fitness functions.

Another area of my project which could be vastly expanded as I have stated several times in this project is the constraint solver. What I have implemented is only a small fraction of how far it could go. There are countless more cognitive processes that humans go through to solve a Sudoku puzzle and I believe implementing such would give a great comparison between humans and computers. Finding innovative ways to implement such algorithms could make big steps in AI for replication human thought patterns. This sort of thing is not only restricted to solving Sudoku, seeing how replicating human methods for solving problems across multiple puzzles and comparing them would be a great project. That way you could see if a certain puzzle allows such things to become easier. I believe this project would be great for the AI community.

For anyone wishing to continue my work directly, I would suggest the following:

- Implement the fitness function for random search
- Treat the constraint solver as a separate algorithm
- Include more algorithms for comparison
- Test the algorithms 1000 times each time for more reliable results
- Compare the algorithms across different sized Sudoku boards

CONCLUSION

The aim of this project was *“to implement and compare various algorithm's performance at solving sudoku puzzles”.* I believe this was successfully done to a great standard and I have some rich results to tell it. The project has produced a great functioning UI which allows the user to carry out tests to their specifications. There are improvements that could made yes, but I would mark the project overall a success.

To conclude from my results, of the algorithms tested brute force was definitely the strongest and the one I would recommend for use in solving Sudoku puzzles as it is both strong and quick. A* star search and random search are both slower and weaker and therefore less suited to this problem. The constraint solver is slow but with more work could be just as reliable as brute force. I believe this is because the search algorithms are reliant upon the search space which grows to big for them to function as the levels increase. Where as brute force is always guaranteed to find a result and across 9*9 puzzles the algorithm does not have a lot of data to brute force. In my results the success rate was averagely lower when using the constraint solver, I believe this may have been due to an unfound error in the algorithm or it taking so long it times out. The constraint solver works well with the search algorithms as it is designed to reduce the search space.

REFLECTION ON LEARNING

In this project I have expanded my options for approaching a problem and seen that sometimes the simplest way is the best. I have also delved deeper into AI, an area I am very interested in. Technically I have gained a solid base in the C++ language and even creating full functioning graphical systems with it with the thanks of QT. Industry standard skills which could easily be transferred to a job or another project. I have learnt a lot as well in how humans and computers work differently and how to adjust my way of thinking so that I can approach a problem for an optimal code based solution and not a human logical solution. I have also increased my ability to take code from its design to it's completion as well as the ability to adjust to new challenges with new approaches to solving bugs. I have learnt even more about the importance of designing a system rather than jumping headlong into code as soon as possible. Finally I have also gotten a better feel for time constraints regarding implementation, this is very important as it will allow me in future to plan more accordingly and prioritise important features as well as allowing me to actually get more coding done.

Of the three assumptions I made all three were correct to some degree. I was definitely right when I made the assumption that no special hardware or software requirements would be needed. However if I were to expand the puzzles or the number of tests this may become incorrect very quickly, this may also depend on the ability to thread the application.

The first assumption was that I could use similar approaches that I have for other problems in AI. I was not wrong, the A* search algorithm which I have used before was able to solve the Sudoku puzzles. However it was not entirely suited as it was unable to solve the more difficult puzzles despite the implementation of multiple path pruning and dead states.

The second assumption was that human techniques could be turned into an algorithm to solve the puzzle and this was not wrong they could and they worked however I did not realise at the time how hard and time consuming it would be to do so.

REFERENCES

[1] QT. http://www.qt.io

APPENDIX

[1] c1218350_solving_sudoku_code.zip

This contains all of the code for my project.