University of Leeds

Mathematics

Project in Mathematics

Automatic Puzzle Solving

Author:
Thomas Morgans

Supervisor:
Dr. Daniel Kirk

October 30, 2020

Acknowledgements

I would like to acknowledge my fellow peers Ellie McKillop, Hugh Parry, Jake Priestman, Katy Rigby and Ryan Pettit for collaborating with me to produce the initial backtracking code in section 4. I would also like to thank my project supervisor Dr. Daniel Kirk in offering advice and support when needed.

Contents

1 Introduction

A puzzle is simply a problem that requires some ingenuity or knowledge to understand and solve. People solve puzzles everyday, whether they are crosswords in newspapers, questions in tests, or more mundane things such as what is the best way to travel to work. Puzzles intrigue us as they provide mental challenge followed by a sense of achievment once the problem has been solved. This project will look into a more recent type of puzzle; the *sudoku*. Sudokus were invented in the late 20th century, but in their short history they have become one of the most popular puzzles in newspapers and magazines around the world.

The aims of this project are to produce algorithms to solve less studied variations of the traditional sudoku, the X sudoku and the 4×4 variant. The project will explore the nature of these puzzles, explaining what they are, how they can be generated and ultimately how they can be solved. To achieve this, algorithms for generating and solving regular sudokus will be produced, before being adapted to fit the properties of the desired puzzles.

The first few sections will look at what type of problem sudokus are and what are the best ways to approach them, before using these methods to devise an algorithm that a computer could follow to solve any sudoku puzzle.

To be able to test this algorithm rigorously will require many different puzzles of varying difficulties. The next section will explore how sudoku puzzles are generated as well as how their respective difficulties are graded. Researching into the *solution space* will help better understand the constraints when generating sudokus, in addition to providing insight into why computers are necessary tools for sudoku production. This information will help build an algorithm for generating sudoku with a specified difficulty, which can then be solved by the aforementioned solving algorithm. By analysing the speed at which the sudoku puzzles are completed we can see how the solving algorithm handles problems of increasing complexity.

The final sections will compare and contrast the nature and mathematical structure of the traditional sudoku with other variations, before determining whether or not they can be solved with a similar algorithm. Furthermore, they will discuss how effective the algorithm was at scaling up to more complex puzzles and what improvements could be made.

2 Preliminaries

2.1 Graphs

Graphs shall be used briefly to help explain some concepts, but a complete understanding is not required.

Definition 2.11 (Graph) A *graph* G is a triple

$$G = (V, E, f)$$

where V is a set of vertices, E a set of edges and f a function assigning each edge $e \in E$ a pair of vertices in V called *endpoints*. We say that vertices $u, v \in V$ are adjacent if they are endpoints to the same edge.

Definition 2.12 (Path) A *path* is a sequence of edges of a graph such that you can travel along from a start vertex to an end vertex without visiting any edge or vertex twice (other than the start vertex). A *circuit* is a path where the start and end vertex are the same. A connected graph is a graph where there is a path from each vertex to any other.

Definition 2.13 (Tree) A *tree* is a connected graph with no circuits.

2.2 Groups and latin squares

Definition 2.21 (Group) A *group* is a non-empty set G on which is defined a binary operation $\circ$ such that:

- G is closed under operation hence if $x, y \in G$ then $x \circ y \in G$
- There exists an identity element $e \in G$ such that $x \circ e = e \circ x = x$ for all $x \in G$
- Every element $x \in G$ has an inverse $x' \in G$ such that $x \circ x' = x' \circ x = e$
- $\circ$ is associative therefore for all $x, y, z \in G$ $(x \circ y) \circ z = x \circ (y \circ z)$

Definition 2.22 (Equivalence relation) An *equivalence relation* is a binary relation $\sim$ that is:

- Reflexive: $x \sim x$
- Symmetric: if $x \sim y$ then $y \sim x$
- Transitive: if $x \sim y$ and $y \sim z$ then $x \sim z$

An equivalence relation partitions a set into *equivalence classes*. Two elements are equivalent if and only if they lie in the same equivalence class.

Definition 2.23 (Latin square) A *Latin square* of order n is an $n \times n$ array with each entry being an element of the set $\{1, ..., n\}$ with the property that each number $1, ..., n$ occurs exactly once in each column and row. The rows, columns and symbols in a latin square can be permutated to obtain new latin squares, said to be *isotopic* to each other. Isotopism is an equivalence relation, and so the set of all latin squares can be partitioned into isotopic equivalence classes.

Theorem 2.24 For any positive integer $n \in \mathbb{N}$, there exists a latin square of order n.

Proof Suppose you have a top row with filled with numbers $1, 2, ..., n$ from left to right. On the next row permute the each element of the last one place to the left, so that the second row reads $2, 3, ..., n, 1$. Repeat this process until the final rows reads $n, 1, ..., n-1$. The resulting grid has n rows and columns, with each row and column containing each number $1, 2, ..., n$ precisely once. This is by definition a latin square.

3 Constraint Satisfaction Problems

The puzzles studied in this project are examples of *constraint satisfaction problems*. These are problems where a solution can be found using the particular constraints applied to the problem.

Definition 3.01 (Constraint satisfaction problem) A constraint satisfaction problem (CSP) is a mathematical problem with the following properties:
a) A finite variable set $X = \{x_1, x_2, ..., x_n\}$
b) A domain set $D = \{d_{x1}, d_{x2}, ..., d_{xn}\}$ where each non-empty d_{xi} corresponds to the possible values that x_i can take
c) A constraint set $C = \{c_1, c_2, ..., c_m\}$ where each c_i limits the possible values of some $X' \subset X$
A *solution* to a CSP is an assignment to each variable such that all the constraints are adhered to.

One way to solve a CSP is by brute force, where an assignment of variables is applied and then checked to see if the assignment is a solution. If not then another assignment is attempted, and so on until a solution is found. The number of different possible assignments is the cartesian product of all variable domains which is more often than not a very large number, and so

this method would be very inefficient. To solve CSPs more effectively we require some faster algorithms.

3.1 Backtracking

Backtracking is an extremely useful iterative algorithm to help solve CSPs. A backtracking algorithm will check if a solution for a problem works, and if it doesn't it will backtrack to the previous point and try another solution. This can be represented graphically as a tree, where each possible initial step of a problem is a branch from the root vertex, and each branch of every other node is possible step after that. At least one of these paths is a correct solution, so if one path fails you can backtrack to the previous node and try the next logical path. Repeating this process recursively is guaranteed to eventually obtain the correct solution to a given problem. The diagram below (Figure 1) illustrates this, with the start vertex being the initial problem and each circle being a possible candidate or step. The red circles are the failed steps, with the green ones being the solutions. The green arrows represent the solution path the backtracking algorithm has found, and the double line from the start vertex to the failed candidate represents where the algorithm has had to backtrack. Note that the algorithm does not find all of the solutions, as indicated by the unchecked green circle in the bottom right.

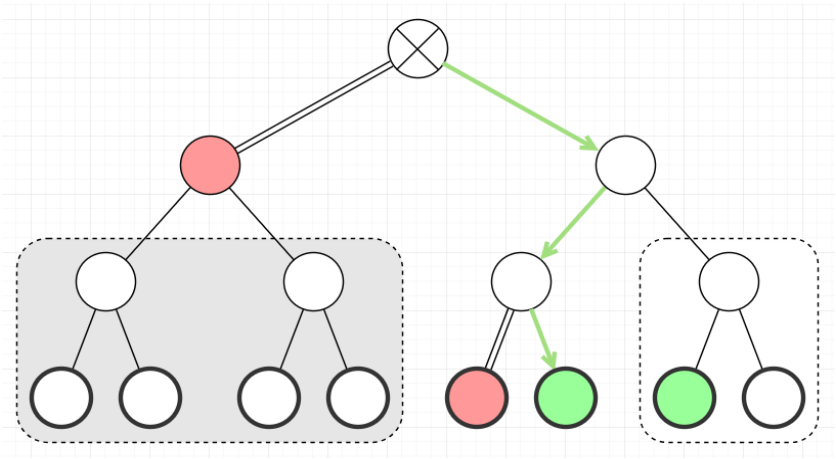


Figure 1: Backtracking. (Zibbu, 2018)

One problem with backtracking algorithms, as explained by Miguel and Shen (2001), is *thrashing*. Thrashing is when a variable x_i will fail for all d_{xi} because of some constraint, but a regular backtracking algorithm will not notice this and instead try all d_{xi}. This is obviously a very inefficient way of approaching a problem. As discussed by Jussien and Lhomme (2002), the

problem of thrashing can be alleviated by using *look-back enhancements*, which "exploit information about the search which has already been performed", and *look-ahead enhancements*, which "exploit information about the remaining search space". An example of a look-back enhancement is backjumping, where the algorithm can go back several phases when in a dead-end situation. Filtering techniques are look-ahead enhancements which remove subsets of the search space that would necessarily fail (Dechter, 1990).

3.2 Constraint propagation and local search

Constraint propagation algorithms work in one of two ways: to reduce the constraints of a CSP whilst maintaining equivalence of the problem, or reducing the domains whilst maintaining equivalence in the same way (Apt, 1999). The former method is called *constraint reduction*, and works by replacing existing constraints with smaller ones. Constraint reduction will only work if the new constraints do not change the logic of the CSP.

Local search algorithms "perform incomplete exploration of the search space by repairing infeasible complete assignments" (Jussien and Lhomme, 2002). This makes them much faster than other algorithms such as regular backtracking, but they are not always guaranteed to find a solution. Local search algorithms are often used along with other algorithms, forming so called hybrids.

4 Solving Sudoku

The sudoku puzzle is a CSP which was invented as we know it today by Howard Garns in 1979 when it was published in Dell Pencil Puzzles as well as Word Games magazine. It wasn't until 1984 that it was named sudoku by Maki Kaji in japan and published by puzzle company Nikoli.

Sudokus can come in many different forms, but the most widely used is the 9×9 grid of squares or cells, separated into nine 3×3 boxes of nine cells each. The numbers 1 through to 9 must be within a cell in every row, column and box, with no number appearing twice. By the defintion of a CSP, the cells make up the variable set where each cell has the domain set {1,2,...,9}. Unsolved sudokus appear with just a few of the cells filled in with numbers, called clues. At any stage of the sudoku, the viable entries in a given empty cell are called the candidates. Puzzle solvers must use the laws of sudoku and the given clues to eliminate candidates, until they can deduce

what numbers the remaining empty cells must contain. A well-formed sudoku puzzle must have a unique solution, with only one way to fill up the grid. A solved sudoku is a special example of a 9×9 latin square.

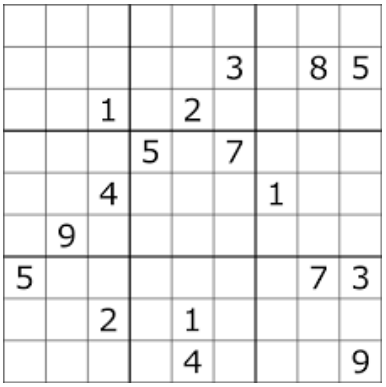


Figure 2: Unsolved sudoku with clues. (Seif, 2017)

4.1 Human methods

When we look to solve sudoku puzzles we often start by noticing that some squares have to hold certain numbers.

Definition 4.11 (Naked single) A *naked single* is a cell with only one candidate.

Definition 4.12 (Hidden single) A *hidden single* is a cell with multiple candidates but only one of these is viable due to the row, column or box constraints.

By looking for naked and hidden singles a solution to easier puzzles can be found. Indeed even more challenging sudokus have such trivial empty cells early on, but after the first few entries it often takes some more complex methods to complete. Preemptive sets can help eliminate possibilities and be used to deduce which numbers go where. The following definitions and theorems are taken from the *Notices of the AMS* (Crooks 2009).

Definition 4.13 (Preemptive sets) A *preemptive set* is a set composed of numbers $\{1, 2, \ldots, 9\}$ of size m, with $2 \leq m \leq 9$, whose numbers are potential occupants of m cells exclusively, where exclusively means that no other numbers in the set $\{1, 2, \ldots, 9\}$ other than the members of the preemptive set are potential occupants of those m cells. The range of the

preemptive set is the row, box or column in which all of the cells in the preemptive set are located.

The markup of a cell is the possible numbers that could currently occupy it. By searching for preemptive sets we can reduce the numbers in the markup of a cell, with the help of the occupancy theorem.

Theorem 4.14 (Occupancy theorem) Let X be a preemptive set in a sudoku puzzle markup. Then every number in X that appears in the markup of cells not in X over the range of X cannot be a part of the puzzle solution.

Proof If any number in X is chosen as the entry for a cell not in X, then the number of numbers to be distributed over the m cells in X will be reduced to m−1, which means that one of the m cells in X will be unoccupied, which violates the sudoku solution definition. Hence, to continue a partial solution, all numbers in X must be eliminated wherever they occur in cells not in X over the range of X.

The occupancy theorem tells us that any number in a preemptive set lying in a specific row, column or box cannot be placed outside of the preemptive set. This is clearly very useful, however not every cell is initially within an obvious preemptive set. Often hidden preemptive sets can be found using existing ones, using the following theorem.

Theorem 4.15 (Preemptive sets) There is always a preemptive set that can be invoked to unhide a hidden set, which then changes a hidden set into a preemptive set except in the case of a singleton (cell with only one possible value).

In Figure 3, there is a preemptive set {1,2,6} in cells (1,7), (1,9) and (2,9). 1,2 and 6 can now be removed from any other cell markup. It follows that 8 is now a naked single, and the only possible solution for cell (1,8). Removing 8 from any cell markup reveals {3,5} to be a new preemptive set. Using these two theorems we can reduce the size of cell markups, making the puzzle much simpler to solve.

Often these are the only tools necessary to solve sudoku puzzles, but for harder ones it is usually not enough. One such situation is when there are two possible solutions to at least one cell, but there are no more premptive sets to be uncovered. When this happens the solver could use backtracking, and choose one solution and see if the puzzle can be solved thereafter. If not, they must backtrack to the cell of their earlier choice and choose a different solution.

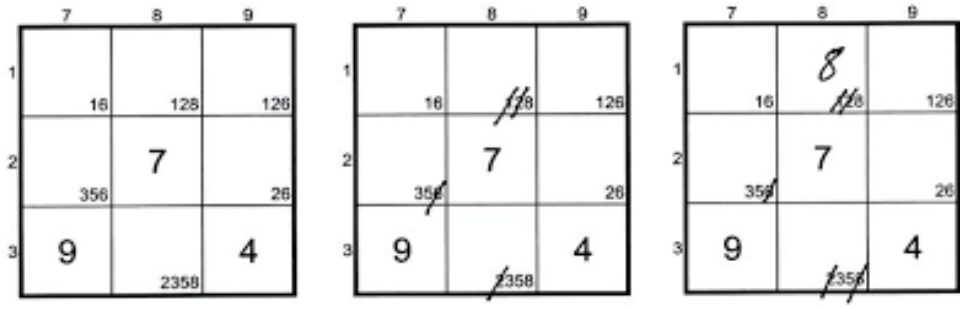


Figure 3: Uncovering hidden pair. (Crook, 2009)

4.2 Backtracking algorithm in Python

Although every well-formed sudoku puzzle should be solvable by human methods, these can be very cumbersome when compared to computing methods; problems that take even the best sudoku solvers minutes can be computationally solved in seconds. Computers can carry out complex algorithms much faster than we can, and with the aid of programming languages such as Python, sudoku solving can become much easier.

Backtracking is a good place to start when writing algorithms for CSPs, as they are often easily followed and will always produce a solution. A backtracking algorithm for a sudoku could be the following:

---

while There are empty cells **do**
  Find an empty cell
  while Cell is empty **do**
    for i in$\{1, 2, ..., 9\}$ **do**
      if i is a valid entry **then**
        Enter i into the cell
      end if
    end for
    Backtrack to last valid entry
    Let this entry be x
    for j in$\{x + 1, ..., 9\}$ **do**
      if j is a valid entry **then**
        Enter j into the cell
      end if
    end for
  end while
end while

The algorithm here traverses the empty cells and checks which of the num-

bers 1 through to 9 are valid entries. When there are no valid entries it backtracks to the previous empty cell and tries a different value. If there are no longer any valid entries in this cell then the algorithm backtracks again to the empty cell before it, and so forth until a complete solution is found.

It is possible to use the programming language Python to code a recursive algorithm for solving sudoku, using the pseudocode above as a skeleton. First of all we require a function that, given a well formed sudoku puzzle, could find the empty cells and return their position in the puzzle (See appendix A). Next we need a function that can check if a number in a specific cell adheres to the laws of sudoku; that there is no other copy of the number contained in the same column, row or box.

```
def valid(sudoku, num, position):
    for i in range(len(sudoku[0])):
        if sudoku[position[0]][i] == num and position[1] != i:
            return False
    for j in range(len(sudoku)):
        if sudoku[j][position[1]] == num and position[0] != j:
            return False
    box1 = position[1]//3
    box2 = position[0]//3
    for i in range(box2*3, box2*3+3):
        for j in range(box1*3, box1*3+3):
            if sudoku[i][j] == num and (i,j) != position:
                return False
    return True
```

The actual backtracking method is used in the recursive function below. It uses the functions above to enter solutions into empty cells and check if they are valid, and when it needs to backtrack it simply calls itself. Once there are no more empty cells the function will return True. This solves a sudoku in array form and will return false if it is unsolvable. The variable sudoku is now in fact the solution to the puzzle.

```
def backtrack(sudoku):
    position = emptySquare(sudoku)
    if not position:
        return True
    else:
        r,c = position
    for i in range(1,10):
        valid(sudoku, i, position)
        r,c = position
```

```
        if valid(sudoku, i, position):
            sudoku[r][c] = i
            if backtrack(sudoku):
                return True
            sudoku[r][c] = 0
    return False
```

This code should solve all sudoku puzzles, however it should be noted that this code only finds one solution, hence it will not find all the solutions if the sudoku is not well-formed. To test the code rigorously requires a wide range of sudoku puzzles of various difficulties. The following section will look to investigate how we can generate these.

5 Generating Sudoku Puzzles

Sudoku puzzles are completed by thousands of people daily, whether in newspapers or on online websites, and are becoming increasingly popular. As a result, new sudoku problems must be created by puzzle makers continuously to supply the ever increasing demand. Furthermore, often each one will produce different levels of sudoku, be it easy, medium or hard. From this arises many questions: how many clues do we require for a sudoku to be unique? How many ways are there to fill a sudoku grid and what determines the difficulty of each puzzle? This section will look to answer these questions, in addition to proposing an algorithm for generating sudokus.

5.1 How many sudoku solutions are there?

Filled sudoku grids are special cases of 9×9 latin squares. It has been calculated that there are $5524751496156892842531225600 \approx 5.525 \times 10^{27}$ such latin squares (Bammel and Rothstein, 1975). The number of filled sudoku grids is smaller than this number, as a completed sudoku puzzle is a latin square with the added restriction that each 3×3 box must only hold one of each number up to nine.

Definition 5.11 (Bands and stacks) A *band* in a sudoku is a group of three horizontal boxes, with cell dimensions of 3×9. A *stack* is a group of three vertical boxes, with cell dimensions 9×3.

Felgenhauer and Jarvis (2006) devised a way of calculating how many of these special latin squares there are. They first noticed that there are 9! ways of rearranging the numbers one to nine in any given box. Using what they called the "standard form" box, with subsets of the set of possible en-

tries {1,2,3} in the first row, {4,5,6} in the second and {7,8,9} in the third, they realised that there are twenty ways of filling out the top band of the sudoku with the initial box in standard form. Either the top rows of the second and third boxes in the top band are the subsets {4,5,6} and {7,8,9}, in which case there are two options, or they are a mixture of these two, giving another eighteen options. In the former choice you have to fill each remaining box rows with another subset. There are six ways of rearranging rows and six ways of rearranging each subset. Using one of the latter eighteen choices for the top row gives three times the number of arrangements as this. Adding this all together gives $2 \times 6^6 + 18 \times 3 \times 6^6 = 2612736$ ways of filling the top band with the top lefthand box in standard form. Multiplying this number by 9! gives the number of ways of filling out any band, precisely $9.481096397 \times 10^{11}$.

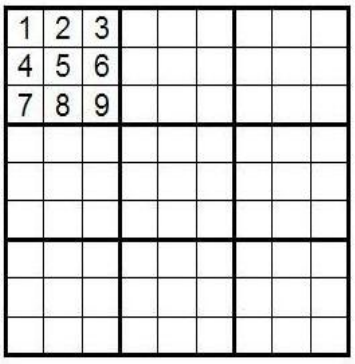


Figure 4: Standard form. (Mahmood, 2009)

To find the total number of sudoku grids we need to find how many different ways there are of filling the rest of the grid and multiply this by the number above. This is a very large number, so trying to count all of these would be incredibly inefficient. Felgenhauer and Jarvis realised that you could partition the set of all possible top bands into equivalent classes. Bands in the same equivalency class are isotopic or logically equivalent, thus have the same number of possible ways to fill the remaining bands. This method of counting is called *set enumeration*. By establishing what the equivalence classes were they would only require to count the possible last two bands for each class. By permuting columns in each box, swapping certain cells and relabelling numbers they could maintain logical equivalence. Once this had been accomplished a brute force algorithm was used to compute the possibilities for each class. Summing all of this they calculated that there are approximately 6.671×10^{21} ways to fill up a sudoku grid.

However, many of these sudoku grids are isotopic to each other, and logically

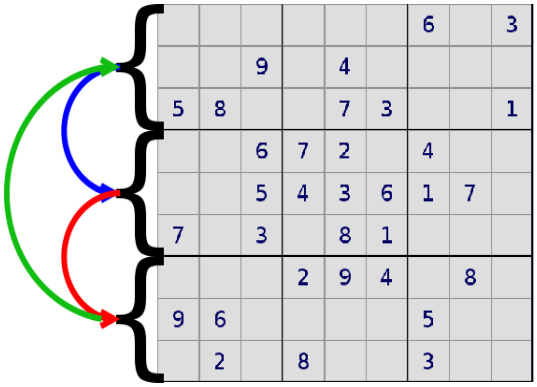


Figure 5: Permuting bands. (Sudoku Garden, no date)

speaking identical to all other grids in their equivalency class. Russel and Jarvis (2006) used logical equivalence to find out how many filled sudoku grids are essentially different. They noted that the following operations preserved the isotopism of the puzzle:

- Rotating the grid by 90°, 180° and 270°
- Reflecting the grid in the vertical, horizontal and both diagonal axes
- Permuting the three stacks
- Permuting the three columns within a stack
- Permuting the three bands
- Permuting the three rows within a band
- Relabelling numbers

The first two of these operations are precisely the group symmetries of a square. The order of the stacks and bands do not affect their contents and so can be permuted. Similarly, the columns and rows within the stacks and bands do not affect other columns or rows respectively when permuted. These permutations are said to be *orthogonal* to each other, as they don't interact. The final operation arises from the fact that the numbers in each cell are purely symbolic and can be swapped around or permuted. For example, if you were to swap all ones with twos, all twos with threes, all threes with fours and so on the puzzle itself maintains its structure, but with different symbols.

Defining a group G consisting of the symmetries built up from these operations, the number of elements of G is equal to the number of grids isotopic to each other. There are 6^6 ways of reordering rows and columns, thirty six

ways of permuting stacks and bands and 9! ways of relabelling the numbers. Although there are eight symmetries to a square, six of these are covered by row permutations, so we only obtain two new symmetries. From this we get that each equivalence class has $6^6 \times 36 \times 2 \times 9! = 1.218998108 \times 10^{12}$ elements. Dividing the number of possible sudoku grids by this gives 5472730538 logically different filled sudokus.

This massive solution space is why we require computers capable of doing lots of calculations quickly in order to produce sudoku puzzles.

5.2 How many clues does a well-formed puzzle require?

It has long been surmised that that fewest amount of clues that a well-formed sudoku puzzle requires to be solved is seventeen, but this has only been proven in the last decade (McGuire et al., 2013). Gary McGuire and his team confirmed that there are no well-formed sudoku puzzles with sixteen clues or less using a brute force algorithm and a computer program called Checker. There are roughly 10^{21} such puzzles, but they used set enumeration along with logical equivalence to significantly cut down the number of grids they had to check.

Finally setting a lower bound on the necessary number of clues is useful in generating puzzles with unique solutions, but it also leads to a good question: do sudokus become more difficult with fewer clues?

5.3 Determining sudoku difficulty

Most newspapers or puzzle websites and magazines will have sudoku grids ranging from beginner level difficulty to harder ones for experts. The tougher sudokus will often have less clues, leading to the assumption that fewer clues means a more complex puzzle. Whilst this is the case more often than not, and indeed the hardest sudoku will have the fewest clues, it is not true that the number of clues defines the difficulty. It is possible for a sudoku with thirty clues to be solved by filling in naked and hidden singles, which would be considerably easier than one with forty clues that requires the use of preemptive sets.

Important things to factor in, as well as how many initial clues there are, are how many naked or hidden singles are available from the start, what techniques are required to complete the puzzle, and how many different techniques must be used (Hunt et al., 2007). The number of initial clues and, more importantly, where and what those clues are, helps determine

these factors that in turn define the difficulty. For example, suppose you have two sudoku puzzles with the same number of clues, but one of the puzzles has no 8s or3s, whereas the other puzzle has all the numbers 1 to 9 somewhere in the grid. The latter is clearly more likely to be considered the easier of the two.

Before an algorithm can be written to generate unsolved sudoku grids, a metric should be used to define the puzzle difficulty. A good metric needs to incorporate more than just one of the ideas listed above. A possible metric could be the following:

- **Easy**: Can be solved using naked singles or uncovering hidden singles
- **Medium**: Can be solved without the need to find preemptive sets with more than three elements
- **Hard**: Cannot be solved with just methods above

5.4 Generating sudoku in Python

This is not a particularly useful metric when programming in Python, as the code does not use the human methods in the same way the metric requires. The simplest way to define the difficulty would be to restrict the number of initial clues, which as we discussed will have its own limitations. From this we can build an algorithm:

Start with a full grid
Assign a value c the desired number of clues
for j in $\{1, 2, ..., c\}$ **do**
  Remove entry from a random cell
  Check solution is unique
  while Solution is not unique **do**
    Replace the removed entry and remove a different one
  end while
end for

The code below uses the concepts of the above algorithm, but the backtracking code in the earlier section fails to verify that the solution is unique and so the sudoku generated is not necessarily well-formed.

```
def sudokuGenerator(grid, clues):
    for i in range(10):
        x = rand.randint(0,8)
        y = rand.randint(0,8)
```

```
        if grid[x][y] != 0:
            while grid[x][y] != 0:
                x = rand.randint(0,8)
                y = rand.randint(0,8)
        while grid[x][y] == 0:
            r = rand.randint(1,9)
            if bs.valid(grid, r, (x,y)):
                grid[x][y] = r
    bs.backtrack(grid)
    for i in range(81-clues):
        x = rand.randint(0,8)
        y = rand.randint(0,8)
        if grid[x][y] == 0:
            while grid[x][y] == 0:
                x = rand.randint(0,8)
                y = rand.randint(0,8)
        grid[x][y] = 0
    return grid
```

This code takes a blank sudoku grid and fills ten random cells with randomly generated entries, using the random module, checking that each entry does not violate the constraints of the puzzle. The backtracking code is called and then solves the puzzle to produce a full grid. The first ten entries are purely to ensure that each sudoku produced does not have an identical solution, as they all would if the backtracking code worked on an empty grid. The for loop iterates just ten times as too many filled cells might cause the puzzle to be unsolvable. Note that it doesn't matter that there are many solutions to a ten clue sudoku, as this is just a means to produce a full grid. After obtaining a full sudoku grid, the code removes entries until the desired number of clues is achieved.

6 Testing

The backtracking algorithm can be tested against sudokus of differing complexity using the generating code to produce puzzles with a desired number of clues. The hypothesis is that as the difficulty of the puzzles increase, so should the time taken for the algorithm to solve them. Scatter graphs were used to represent the data collected as they effectively show the individual datapoints, but line graphs that emphasise the relationship between the time and the number of clues can be found in appendix B.

6.1 Results

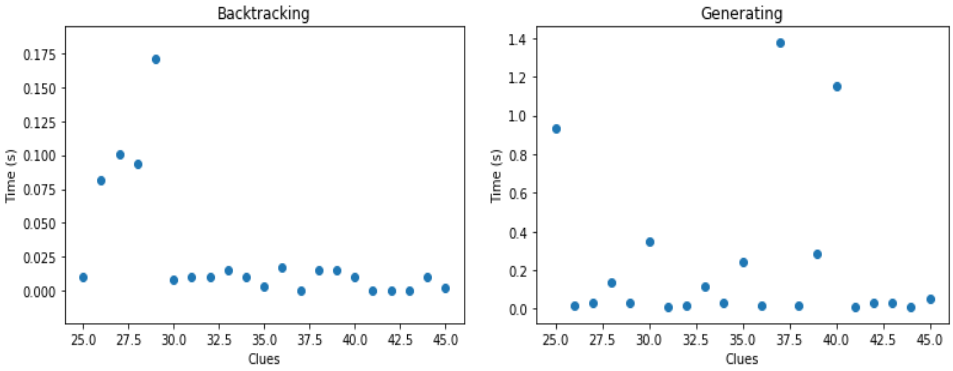


Figure 6: Time taken for backtracking code (left) and generating code (right)

The code timed how long it took to generate and solve puzzles separately, with an increasing number of initial clues from twenty-five up to forty-five. The code generated and solved puzzles considerably faster than a human could, with the puzzle the backtracking algorithm found most difficult taking roughly 0.175 seconds. The graphs at the top of the page show that interestingly there is seemingly no correlation between the time taken to generate a puzzle and how difficult the puzzle is to solve. Although it is clearly true that the grids with the most initial clues all took comparatively little time to solve, and a line of best fit would support the hypothesis above, the sudoku do not consistently become easier to solve with each increment in the number of clues. This either implies that the code doesn't take longer to solve "harder" puzzles, or that the puzzle does not necessarily increase with difficulty if there are more clues.

6.2 Limitations

The hypothesis was only partially supported, but this is likely down to a number of limitations of the code. As stated previously, the number of initial clues of a sudoku affects the difficulty, but it does not define it. As a result the metric used is not an accurate representation of actual difficulty. Some of the sudoku with more clues may have taken longer to solve than ones with fewer because they were in fact more difficult. Furthermore, as the sudoku solution was defined only by how the ten clue sudoku was solved, many of the sudokus tested had similar solutions. This is perhaps why there was very little difference in the time taken to complete most of the puzzles.

Although the backtracking code clearly worked and solved all the puzzles

within a few seconds, it did not verify that the solution found was unique. This meant that the generating code was flawed in that it didn't guarantee that the sudokus being produced were well-formed. Hence it could be argued that the validity of this data is affected, although some of the puzzles likely were well-formed.

7 Sudoku Variants

7.1 X sudoku

The X sudoku, or the sudoku X, is a modern variation on the regular sudoku puzzle. The X sudoku retains the same dimensions as a regular sudoku, but with the added constraint that each of the numbers $1, 2, ..., 9$ must occur in the both diagonals exactly once. Because of this, the solution space is smaller than that of a regular sudokus.

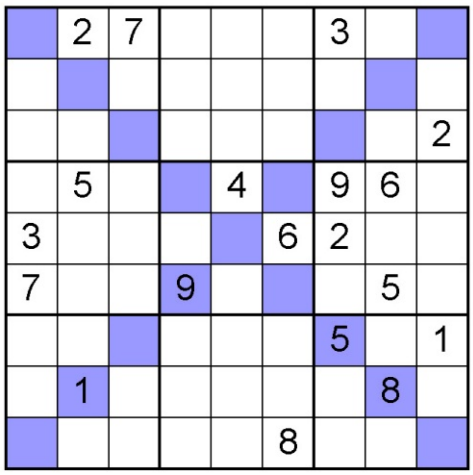


Figure 7: X sudoku. (Try Sudoku, no date)

To solve this puzzle the backtracking algorithm needs to check whether or not this constraint is violated, on top of the constraints of the regular sudoku. This means that the only part of the existing code that needs modifying is the function that checks the validity of an entry.

```
if position[0] == position[1]:
    for i in range(len(sudoku)):
        if sudoku[i][i] == num and position[0] != i:
            return False
if position[0] + position[1] == 8:
```

```
    for i in range(len(sudoku)):
        if sudoku[i][8-i] == num and position[0] != i:
            return False
return True
```

This is part of the new validating function first checks the diagonal from the top left to the bottom right of the grid, before checking the diagonal from the top right to the bottom left.

The backtracking code is the same process as before, as is the generating code (see appendix A) excluding how many squares are randomly entered into the initial blank grid. The code for the X sudoku enters in seven numbers instead of ten, as the additional constraint may cause a sudoku with ten random clues to have no solution. As a result, there will be fewer differences with each solution.

7.2 Higher dimensions

As sudokus are a special type of latin square, and there exists a latin square of order n for any $n \in \mathbb{N}$, then there are $n \times n$ sudokus for all such n. These sudokus adhere to the same rules as regular sudoku, just on a larger scale, and so the current backtracking algorithm should be able to solve any $n \times n$ sudoku if it is modified to traverse the correct grid dimensions, albeit increasingly slowly as $n \to \infty$.

This paper will look at the 4×4 variant, as higher dimensions could take the code an exceedingly long time to produce results. For this reason the number of cells filled in to define the puzzle solution was only increased from ten to twelve, and so there was even less differentiation between the solutions given the size of the grid.

8 Results

As in the testing section, scatter graphs are used with the line graphs in the appendix B.

8.1 X sudoku

The algorithm remained efficient for X sudokus in that it solved the puzzles in similar times to regular sudokus, with the exception of the twenty-nine clue X sudoku that took over half a second. It is hard to determine whether there is any correlation between the puzzle difficulties and the time taken to

solve them, and so a larger range of puzzles should be tested. This wasn't achieved as the generating function took considerably longer to generate X sudoku puzzles than it did with regular ones (as shown by the graph in figure 8). Once again the time taken to generate a puzzle seems to be independent with the time taken for that puzzle to be solved.

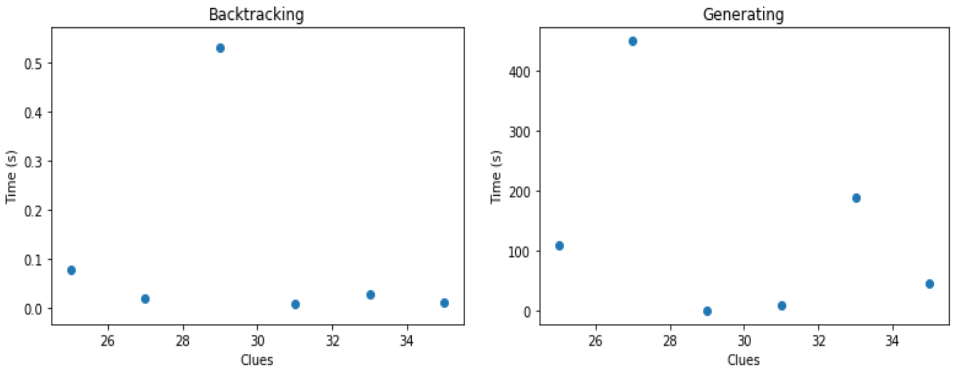


Figure 8: X sudoku

8.2 4×4 variants

The backtracking algorithm solved the 4×4 grids even faster than the regular sudokus. This implies that the number of clues and difficulties was not scaled up effectively from the regular grid to the larger one. The range of puzzles tested was based off how long the generating algorithm was taking to produce grids, and to compensate for the increased time it took to do this a large range was used with only four grids actually being tested. This larger range with fewer datapoints did however show evidence for a downwards curve as the number of clues increased. As with the other results there was no relationship evident between the times taken for a puzzle to be generated and then solved.

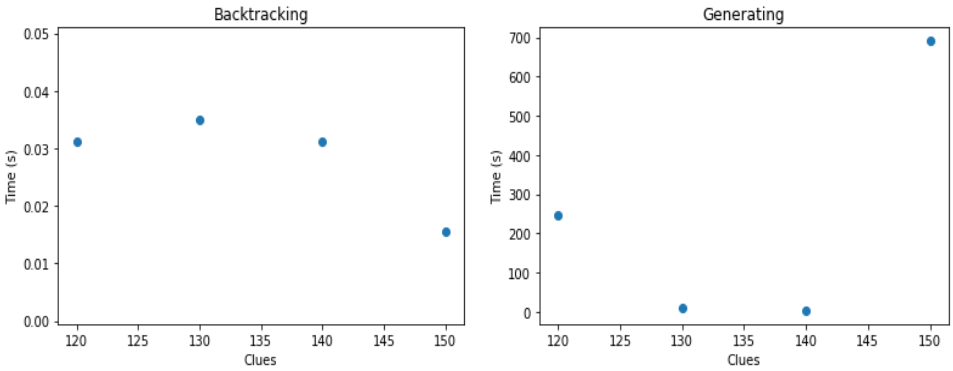


Figure 9: 4×4 sudoku

8.3 Conclusions from the results

From these results we can infer that although the metric for difficulty was not very thorough, the data does suggest that over a large enough range of sudoku grids more clues do simplify the complexity of the puzzles, even if this is not obvious from the results of the X sudoku. The backtracking algorithm also managed to solve all the grids it was used on efficiently, with most of the differences between solving times for different sudoku fractional. It is unclear what determines the time taken for the generating code to produce sudokus, as there is no evident trend. This seems to be because of the way empty grid is filled up in the algorithm, with the assortment of clues randomising the grid's difficulty and thus how long the backtracking algorithm takes to solve it.

9 Conclusion

The aims of the project were to produce algorithms for solving less studied variations on the traditional sudoku puzzle, in addition to exploring the nature of sudoku solutions. Indeed, using a backtracking algorithm that worked on regular sudokus, an adapted version was able to solve both the X sudoku and the 4×4 variant, with the potential for similar algorithms to be used on sudoku grids of any dimension. The results section showed that the backtracking algorithm was very efficient at solving these sudokus with all grids being solved in under a second, including the ones with the fewest clues. Therefore it can be concluded that the algorithm was also effective at dealing with puzzles of higher complexities.

However, the conclusions drawn from these results must acknowledge the limitations of the methodology. The metric used to define the puzzle diffi-

culty was the number of initial clues it held, which is only one factor of many contributing to the puzzle complexity. This explains why the backtracking algorithm did not necessarily solve the "easier" puzzles in less time. Furthermore, the code that generated sudoku grids to be solved did not necessarily produce grids that had unique solutions. Moreover, because the backtracking algorithm was used in the generating code to fill the grid, and the grid only held a few clues, many of the solutions to the puzzles produced were very similar. This is likely why many of the sudokus have taken virtually the same time to be solved. The use of the backtracking algorithm to solve a mostly empty grid was also what slowed the generation of sudoku puzzles down, especially in the case of the 4×4 sudoku. This meant that the paper was unable to test a wide range of sudokus for the 4×4 and X sudoku variants, therefore could not provide strong evidence for a relationship between the difficulty of the puzzle and the time taken for it to be solved.

The reason why the generating code could not guarantee the solution of the puzzle to be unique was that the backtracking algorithm had no means to check this. A way to ensure that the code produced well-formed puzzles would be to devise an algorithm that could check that the solution it has found is indeed the only solution. This could then be implemented into the generating code by checking the solution after every cell is removed from the completed grid. The code could also have an improved way of initially filling the grid to produce a wider range of solutions, or even better, the algorithm could start from an empty grid, entering in valid candidates whilst checking there is a solution after each entry.

Further research could look into how the backtracking algorithm produced in this project could be enhanced by local search, or even bettered by a constraint propagating approach. It would also be interesting to explore whether there are better ways to determine difficulty of generated puzzles in Python, incorporating all factors contributing to the complexity. Another way of taking this project further would be to investigate what the minimum number of clues is to maintain a unique solution for the X sudokus, or even sudokus of different shapes and dimensions. By looking at the minimum clues required by the 2×2 and the 3×3 sudokus perhaps it is possible to deduce the necessary clues needed for the 4×4 and the 3×4 sudoku puzzles.

References

[1] Anon. [No date]. Similar Sudokus and Isomorphism. *Sudoku Garden*. [Online]. [Accessed 2 February 2020]. Available from: https://sudokugarden.de/en/info/similar

[2] Apt, K.R. 1999. The essence of constraint Propagation *Theoretical computer science*. 221(1-2), pp.179-210.

[3] Bammel, S.E. and Rothstein, J. 1975. The number of 9×9 Latin squares. *Discrete Mathematics*. 11(1), pp.93-95

[4] Crook, J.F. 2009. A pencil-and-paper algorithm for solving Sudoku puzzles. *Notices of the AMS*. 56(4), pp.460-468

[5] Crook, J.F. 2009. A pencil-and-paper algorithm for solving Sudoku puzzles. *Notices of the AMS*. [Online]. [Accessed 12 March 2020]. Available from: https://www.ams.org/notices/200904/rtx090400460p.pdf

[6] Dechter, R. 1990. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial intelligence*. 41(3), pp.273-312

[7] Felgenhauer, B. and Jarvis, F. 2006. Mathematics of Sudoku 1. *Mathematical spectrum*. 39(1), pp.15-22

[8] Hunt, M. Pong, C. and Tucker, G.2007. Difficulty-driven sudoku puzzle generation. *The UMAP Journal*. 29(3), pp.343-362.

[9] Jones, S.K. Roach, P.A. and Perkins, S. 2011. Sudoku Puzzle Complexity. *Proceedings of the 6th Research Student Workshop*. pp.19-24

[10] Jussien, N. and Lhomme, O. 2002. Local search with constraint propagation and conflict-based heuristics. *Artificial intelligence*. 139(1), pp.21-45

[11] Laywine, C.F. and Mullen, G.L. 1998. *Discrete mathematics using Latin squares*. (Vol. 49). John Wiley and Sons.

[12] Mahmood. 2009. *The Math Behind Sudoku: Counting Solutions*. [Online]. [Accessed 4 April 2020]. Available from: http://pi.math.cornell.edu/ mec/Summer2009/Mahmood/Count.html

[13] McGuire, G. Tugemann, B. and Civario, G. 2014. There is no 16-clue Sudoku: Solving the Sudoku minimum number of clues problem via hitting set enumeration. *arXiv preprint arXiv:1201.0749.*

[14] Miguel, I. and Shen, Q. 2001. Solution techniques for constraint satisafction problems. *Artificial intelligence review*. 15(4), pp.243-267

[15] Pang, S. Li, E. Song, T. and Zhang, P. 2010. Rating and generating sudoku puzzles. In *2010 Second International Workshop on Education Technology and Computer Science*. pp. 457-460. IEEE.

[16] Russel, E. and Jarvis, F. 2006. Mathematics of Sudoku 2. *Mathematical spectrum*. 39(2), pp.54-58

[17] Seif, G. 2017. *Solving a sudoku using a simple search algorithm*. [Online]. [Accessed 12 March 2020]. Available from: https://medium.com/@george.seif94/solving-sudoku-using-a-simple-search-algorithm-3ac44857fee8

[18] Try Sudoku. [No date]. *Free X sudoku*. [Online]. [Accessed 6 April 2020]. Available from: http://www.trysudoku.com/sudoku-x/4

[19] Wilson, R. 2006. The sudoku epidemic. *Focus*. 26(1), pp.5-7

[20] Zibbu, S. 2018. *Sudoku and Backtracking*. [Online]. [Accessed 15 March 2020]. Available from: https://hackernoon.com/sudoku-and-backtracking-6613d33229af

Appendices

A Code

A.1 Backtracking algorithm

```
def emptySquare(sudoku):
    for i in range(len(sudoku)):
        for j in range(len(sudoku[0])):
            if sudoku[i][j] == 0:
                position = (i,j)
                return position
    return None
```

```
def valid(sudoku, num, position):
    for i in range(len(sudoku[0])):
        if sudoku[position[0]][i] == num and position[1] != i:
            return False
    for j in range(len(sudoku)):
        if sudoku[j][position[1]] == num and position[0] != j:
            return False
    box1 = position[1]//3
    box2 = position[0]//3
    for i in range(box2*3, box2*3+3):
        for j in range(box1*3, box1*3+3):
            if sudoku[i][j] == num and (i,j) != position:
                return False
    return True
```

```
def backtrack(sudoku):
    position = emptySquare(sudoku)
    if not position:
        return True
    else:
        r,c = position
    for i in range(1,10):
        valid(sudoku, i, position)
        r,c = position
        if valid(sudoku, i, position):
            sudoku[r][c] = i
            if backtrack(sudoku):
                return True
            sudoku[r][c] = 0
    return False
```

A.2 Generating sudoku

```
def sudokuGenerator(grid, clues):
    for i in range(10):
        x = rand.randint(0,8)
        y = rand.randint(0,8)
        if grid[x][y] != 0:
            while grid[x][y] != 0:
                x = rand.randint(0,8)
                y = rand.randint(0,8)
        while grid[x][y] == 0:
            r = rand.randint(1,9)
            if bs.valid(grid, r, (x,y)):
                grid[x][y] = r
    bs.backtrack(grid)
    for i in range(81-clues):
        x = rand.randint(0,8)
        y = rand.randint(0,8)
        if grid[x][y] == 0:
            while grid[x][y] == 0:
                x = rand.randint(0,8)
                y = rand.randint(0,8)
        grid[x][y] = 0
    return grid
```

A.3 X sudoku

```
def validX(sudoku, num, position):
    for i in range(len(sudoku[0])):
        if sudoku[position[0]][i] == num and position[1] != i:
            return False
    for j in range(len(sudoku)):
        if sudoku[j][position[1]] == num and position[0] != j:
            return False
    box1 = position[1]//3
    box2 = position[0]//3
    for i in range(box2*3, box2*3+3):
        for j in range(box1*3, box1*3+3):
            if sudoku[i][j] == num and (i,j) != position:
                return False
    if position[0] == position[1]:
        for i in range(len(sudoku)):
            if sudoku[i][i] == num and position[0] != i:
                return False
    if position[0] + position[1] == 8:
```

```
        for i in range(len(sudoku)):
            if sudoku[i][8-i] == num and position[0] != i:
                return False
    return True
```

```
def backtrackX(sudoku):
    position = bs.emptySquare(sudoku)
    if not position:
        return True
    else:
        r,c = position
    for i in range(1,10):
        validX(sudoku, i, position)
        r,c = position
        if validX(sudoku, i, position):
            sudoku[r][c] = i
            if backtrackX(sudoku):
                return True
            sudoku[r][c] = 0
    return False
```

```
def sudokuGeneratorX(grid, clues):
    for i in range(10):
        x = rand.randint(0,8)
        y = rand.randint(0,8)
        if grid[x][y] != 0:
            while grid[x][y] != 0:
                x = rand.randint(0,8)
                y = rand.randint(0,8)
        while grid[x][y] == 0:
            r = rand.randint(1,9)
            if bx.validX(grid, r, (x,y)):
                grid[x][y] = r
    bx.backtrackX(grid)
    for i in range(81-clues):
        x = rand.randint(0,8)
        y = rand.randint(0,8)
        if grid[x][y] == 0:
            while grid[x][y] == 0:
                x = rand.randint(0,8)
                y = rand.randint(0,8)
        grid[x][y] = 0
    return grid
```

A.4 4×4 sudoku

```
def valid4x4(sudoku, num, position):
    for i in range(len(sudoku[0])):
        if sudoku[position[0]][i] == num and position[1] != i:
            return False
    for j in range(len(sudoku)):
        if sudoku[j][position[1]] == num and position[0] != j:
            return False
    box1 = position[1]//4
    box2 = position[0]//4
    for i in range(box2*4, box2*4+4):
        for j in range(box1*4, box1*4+4):
            if sudoku[i][j] == num and (i,j) != position:
                return False
    return True
```

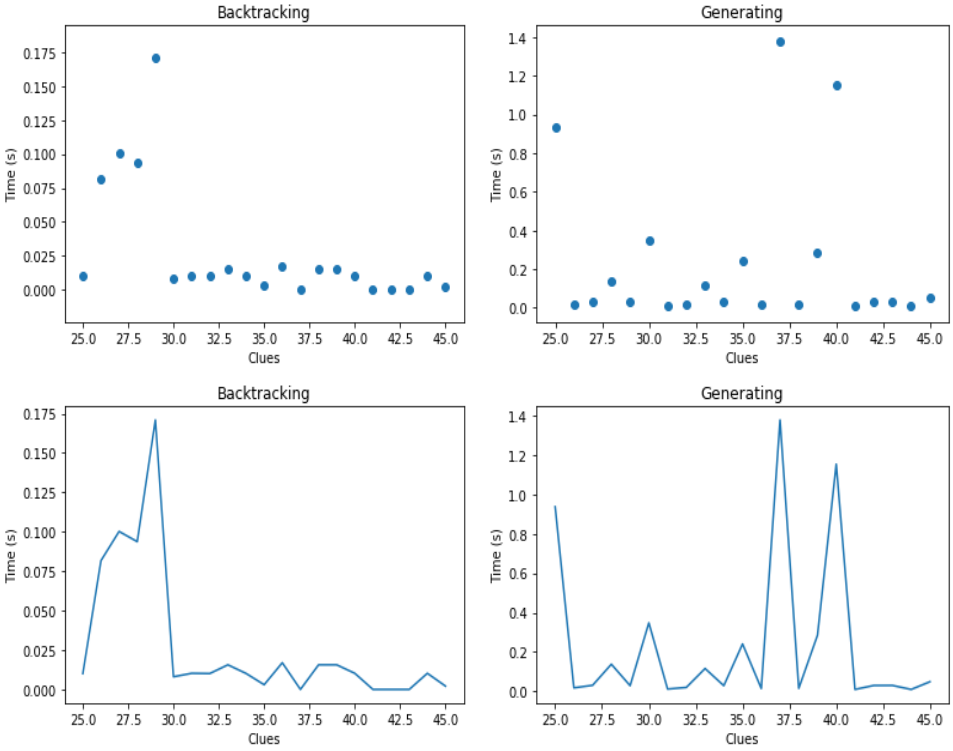
```
def backtrack4x4(sudoku):
    position = bs.emptySquare(sudoku)
    if not position:
        return True
    else:
        r,c = position
    for i in range(1,len(sudoku)+1):
        valid4x4(sudoku, i, position)
        r,c = position
        if valid4x4(sudoku, i, position):
            sudoku[r][c] = i
            if backtrack4x4(sudoku):
                return True
            sudoku[r][c] = 0
    return False
```

```
def sudokuGenerator4x4(grid, clues):
    for i in range(12):
        x = rand.randint(0,15)
        y = rand.randint(0,15)
        if grid[x][y] != 0:
            while grid[x][y] != 0:
                x = rand.randint(0,15)
                y = rand.randint(0,15)
        while grid[x][y] == 0:
            r = rand.randint(1,16)
            if b4.valid4x4(grid, r, (x,y)):
```

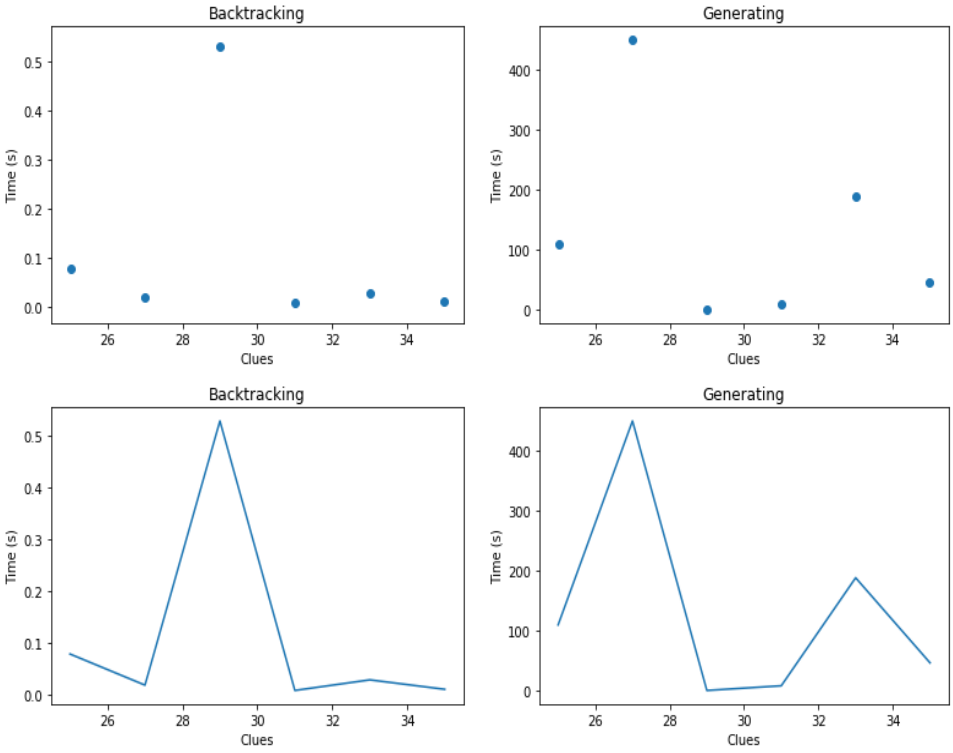
```
                grid[x][y] = r
    b4.backtrack4x4(grid)
    for i in range(256-clues):
        x = rand.randint(0,15)
        y = rand.randint(0,15)
        if grid[x][y] == 0:
            while grid[x][y] == 0:
                x = rand.randint(0,15)
                y = rand.randint(0,15)
        grid[x][y] = 0
    return grid
```

B Graphs

B.1 Regular sudoku



B.2 X sudoku


Backtracking
Time (s)
Clues
Generating
Time (s)
Clues
Backtracking
Time (s)
Clues
Generating
Time (s)
Clues

B.3 4×4 sudoku