Backtracking for Sudoku

Van Thuan Romoli

Semester 2, 2023/2024

1 Introduction

A Sudoku puzzle is a logic-based, combinatorial, number-placement puzzle. To solve the puzzle, the objective is to fill an $n^2 \times n^2$ grid with digits 1 to n^2 such that each column, row, and each of the $n \times n$ sub-grids (called boxes) that compose the grid contain no repeated digit.

The origins of the Sudoku track back to the late 19th century when a Paris daily published a puzzle consisting of magic squares with missing numbers. The aim was to fill in the numbers so that each row, column, and diagonal added up to the same number. A Sudoku is a semi-magic square, a magic square whose diagonal requirement is dropped.

The modern Sudoku was first published in 1979 by Dell Magazines. It then spread to Japan (wherefrom the current name was given) and two additional constraint were added: the number of givens was restricted to 32 and the givens were distributed in rotational symmetry. Hong Kong Judge Wayne Gould developed a computer program to generate unique Sudoku and the puzzle took off rapidly outside Japan across various newspapers and publications.

The modern Sudoku lends itself quite naturally to the study of mathematics and mathematical computation due to its connections to abstract algebra, combinatorics, and graph theory (in pure maths), backtracking algorithms (in computer science), and stochastic based algorithms (in statistics).

The aim of this project is to explore how we may represent Sudoku as mathematical objects and how we can leverage mathematical tools to solve Sudoku puzzles.

This project is divided in 8 Sections. In Section 2, we define Sudoku formally and draw some parallels with abstract algebra, allowing us to study how many Sudoku exist and how they permute. In Section 3, we show how Sudoku can be viewed as graphs, allowing us to apply graph algorithms to solve Sudoku puzzles. In section 4, we give the implementation details of how we can work with Sudoku in Python. In Section 5, we implement a backtracking algorithms to solve Sudoku puzzles. In Section 6, we explore how we can improve the Naive Sudoku solver using prior knowledge. In Section 7, we measure the time our functions take to solve a Sudoku. In Section 8, we summarise the work done in this project writing a conclusion.

2 Mathematics of Sudoku

In this section, we define *Sudoku* formally and draw some parallels with abstract algebra, which allows us to study how many Sudoku exist and how they permute.

An n-Sudoku is a $n^2 \times n^2$ grid, divided into n^2 disjoint squares (called "boxes") of side-length n, and filled with the numbers 1 to n^2 (in "cells") in such a way that no box, column, or row contains a repeated digit. The standard Sudoku is the 3-Sudoku. An n-Sudoku *puzzle* is a partially-filled-in n-Sudoku; the goal is to fill the rest of the grid in to obtain an n-Sudoku.

Latin squares are a $n \times n$ array filled with n different symbols, each occurring exactly once in each row and exactly once in each column. Sudoku puzzles are a special case of Latin squares, in fact any solution to a puzzle is a Latin square (however, the converse is not true).

We may treat Sudoku and Latin squares as mathematical objects. In fact, Latin squares and quasigroups are equivalent mathematical objects. This can be used, for example, to generate Sudoku. Consider the direct product $\mathbb{Z}_n \times \mathbb{Z}_n = \{(x, y) : x, y \in \mathbb{Z}_n\}$ of the finite abelian additive group $\mathbb{Z}_n$, under addition modulo n. The direct product $\mathbb{Z}_n \times \mathbf{0} = \{(x, 0) : x \in \mathbb{Z}\}$ is a normal subgroup of $\mathbb{Z}_n \times \mathbb{Z}_n$ and is isomorphic to $\mathbb{Z}_n$. Define a map $\pi : \mathbb{Z}_n \times \mathbb{Z}_n \to \mathbb{Z}_n$ by $\pi(x, y) = x$. This map is a homomorphism and by using the first isomorphism theorem, it can be shown that $\dfrac{\mathbb{Z}_n \times \mathbb{Z}_n}{\mathbb{Z}_n \times \mathbf{0}} = \dfrac{\mathbb{Z}_n \times \mathbb{Z}_n}{\ker \pi} \cong \operatorname{im} \pi = \mathbb{Z}_n$. To summarise, we found that $\mathbb{Z}_n$ is isomorphic to a quotient group of $\mathbb{Z}_n \times \mathbb{Z}_n$ and isomorphic to a subgroup of $\mathbb{Z}_n \times \mathbb{Z}_n$.

Under this view, Figure 1a shows an example for $n = 3$. Each 3×3 box in the 3-Sudoku looks the same on the second component (namely like the subgroup $\mathbb{Z}_3$). On the other hand, the first components are equal within each box, and if imagine each box as one cell of a 3×3 grid, then these first components show the same pattern (namely like the quotient group $\mathbb{Z}_3$). This is a Latin square of order 9.

Now, to yield a Sudoku, we permute the rows (or equivalently the columns) in such a way that each row of a box is redistributed exactly once into each box. This preserves the Latin square property. Within each box, the rows have distinct first components (by construction), and each row has distinct second components (since the box second component originally formed a Latin square of order 3 from the subgroup $\mathbb{Z}_3$). Renaming the pairs to numbers $1, \ldots, 9$ yields a Sudoku; Figure 1b illustrates this process.

(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)	(2,0)	(2,1)	(2,2)
(0,1)	(0,2)	(0,0)	(1,1)	(1,2)	(1,0)	(2,1)	(2,2)	(2,0)
(0,2)	(0,0)	(0,1)	(1,2)	(1,0)	(1,1)	(2,2)	(2,0)	(2,1)
(1,0)	(1,1)	(1,2)	(2,0)	(2,1)	(2,2)	(0,0)	(0,1)	(0,2)
(1,1)	(1,2)	(1,0)	(2,1)	(2,2)	(2,0)	(0,1)	(0,2)	(0,0)
(1,2)	(1,0)	(1,1)	(2,2)	(2,0)	(2,1)	(0,2)	(0,0)	(0,1)
(2,0)	(2,1)	(2,2)	(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)
(2,1)	(2,2)	(2,0)	(0,1)	(0,2)	(0,0)	(1,1)	(1,2)	(1,0)
(2,2)	(2,0)	(2,1)	(0,2)	(0,0)	(0,1)	(1,2)	(1,0)	(1,1)

(a) Cayley table of $\mathbb{Z}_n \times \mathbb{Z}_n$.

(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)	(2,0)	(2,1)	(2,2)
(1,0)	(1,1)	(1,2)	(2,0)	(2,1)	(2,2)	(0,0)	(0,1)	(0,2)
(2,0)	(2,1)	(2,2)	(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)
(0,1)	(0,2)	(0,0)	(1,1)	(1,2)	(1,0)	(2,1)	(2,2)	(2,0)
(1,1)	(1,2)	(1,0)	(2,1)	(2,2)	(2,0)	(0,1)	(0,2)	(0,0)
(2,1)	(2,2)	(2,0)	(0,1)	(0,2)	(0,0)	(1,1)	(1,2)	(1,0)
(0,2)	(0,0)	(0,1)	(1,2)	(1,0)	(1,1)	(2,2)	(2,0)	(2,1)
(1,2)	(1,0)	(1,1)	(2,2)	(2,0)	(2,1)	(0,2)	(0,0)	(0,1)
(2,2)	(2,0)	(2,1)	(0,2)	(0,0)	(0,1)	(1,2)	(1,0)	(1,1)

(b) Permuted Cayley table of $\mathbb{Z}_n \times \mathbb{Z}_n$.

Figure 1: Generating a Sudoku by permutation of rows and columns.

For this method to work, one generally does not need the direct product of two equally-sized groups. A so-called *short exact sequence of finite groups of appropriate size* ensures this method works for groups that are not direct products. For example, the group $\mathbb{Z}_9$ with quotient and subgroup $\mathbb{Z}_3$ could also generate a Sudoku.

Although we can construct a Sudoku by considering groups, not all Sudoku can be generated this way and in general, Sudoku are not groups. One natural question to ask is "How many 3-Sudoku are there?". The answer depends on the definition of when similar solutions are considered different.

The first known solution to complete enumeration was given by Guenter Stertenbrink in 2003, obtaining 6,670,903,752,021,072,936,960 (6.67×10^{21}) distinct solutions. This number may be reduced by considering essentially different solutions.

A first step would be considering the Sudoku symmetry group. It can be expressed using the wreath product as $(S_3 \wr S_3 \wr C_2) \times S_9$ and has order 1,218,998,108,160. Ed Russell and Frazer Jarvis in 2005 computed the number of essentially different sudoku solutions as 5,472,730,538. To obtain this number,

they used Burnside's lemma to count the number of Burnside fixed points, grids that either do not change under the rearrangement operation or only differ by relabelling.

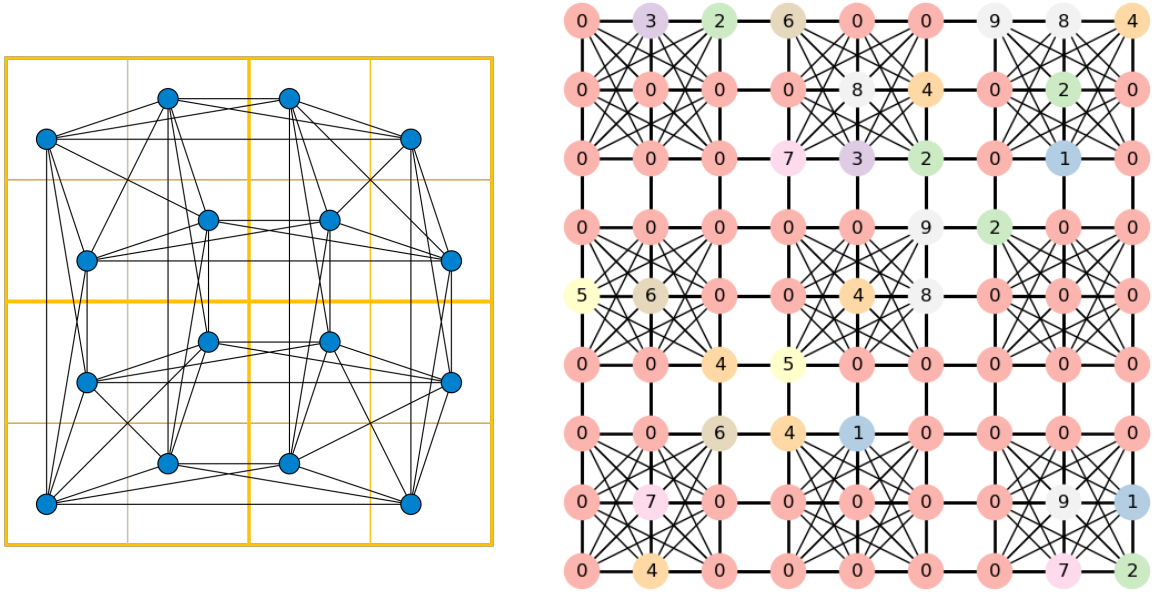
Solving a Sudoku puzzle is obviously a monumental task due to the vast amount of possible solutions. Therefore, computational techniques can be implemented to aid us.

3 Computing of Sudoku – graphs, colouring, and backtracking algorithms

In this section, we show how Sudoku can be viewed as graphs, allowing us to apply graph algorithms to solve Sudoku puzzles.

A Sudoku graph is an undirected graph whose vertices represent the cells of a (blank) Sudoku and whose edges represent pairs of cells that belong to the same row, column, or block of the Sudoku. Figure 2a shows an example of how a 2-Sudoku can be viewed as a graph.

Given a Sudoku puzzle, we may map the numbers in each cell to a colour, for example $1 \rightarrow$ blue, $2 \rightarrow$ green, etc. If a cell contains no number, we simply fill it in with a placeholder number and map it to a colour, for example fill an empty cell with 0 and map it to red. Figure 2b shows an example of how we may represent a 3-Sudoku puzzle as a coloured graph.



(a) Graph of an empty 2-Sudoku.

(b) Coloured graph of a 3-Sudoku puzzle.

Figure 2: Graph of an empty 2-Sudoku and coloured graph of a partially-filled-in 3-Sudoku.

Under this view, we can re-frame a Sudoku puzzle to a graph colouring problem. We leverage a recursive depth-first search algorithm called backtrack search (Algorithm 1). Starting from a given vertex, we may incrementally colour the Sudoku by examining the colour of each of the vertex's neighbour. The equivalence between colouring a graph and solving a Sudoku is quite natural when we consider that filling in the values in a Sudoku is wholly reliant on the value of related cells.

Algorithm 1 Backtrack search.

```
Initialise any data that needs keeping track of, such as a partial solution.
Define an inner function called DIVE, which does the following:
function DIVE( )
    if Partial solution is a potential full solution then
        Check that it is genuinely a solution.
        if Partial solution is genuinely a solution then
            return Solution.
        else
            return None. (This represents a failure).
        end if
    end if
    for Each possible way of extending the partial solution do
        Extend the partial solution in that way.
        if This partial solution is consistent then
            Call DIVE() to recursively search for a solution.
            if DIVE() returned a solution then
                return That solution immediately (to "pass it up").
            end if
        end if
    end for
    return None. (If this point is reached, then no solution was found extending this partial solution).
end function
return DIVE().
```

4 Using Python to represent Sudoku

In this section, we give the implementation details of how we can work with Sudoku in Python.

Before we can start implementing Algorithm 1 to solve a Sudoku puzzle in Python, we need to establish some IO conventions and define a few functions that will be useful throughout this project. The cells of an n-Sudoku are labelled by $0, 1, \ldots, n^4-1$ where $0, 1, \ldots, n^2-1$ is the first row, $n^2, n^2+1, \ldots, 2n^2-1$ is the second row, and so on. The first row of the Sudoku is row 0, and first column is column 0. A partial solution to an n-Sudoku is a list `X` of length n^4, where `X[i]` is the value of the cell i, and unassigned values have value 0.

We provide a few Sudoku puzzles that will be used as examples throughout this projects. They are stored in three files named `unique-3x3.tar.gz`, `non-unique-3x3.tar.gz`, `4x4-puzzle.tar.gz` each containing a list of Sudoku puzzles represented as discussed above.

```
import networkx as nx
import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np
import math
import gzip
import pickle

with gzip.open("unique-3x3.tar.gz", "rb") as f:
    unique_3x3 = pickle.load(f)

with gzip.open("non-unique-3x3.tar.gz", "rb") as f:
    non_unique_3x3 = pickle.load(f)

with gzip.open("4x4-puzzle.tar.gz", "rb") as f:
    puzzle_4x4 = pickle.load(f)
```

```
print(unique_3x3[0])
#    [5, 3, 0, 2, 0, 0, 0, 0, 9,
#     0, 0, 0, 0, 0, 9, 0, 0, 0,
#     2, 0, 0, 0, 4, 3, 0, 8, 0,
#     4, 0, 0, 0, 0, 0, 8, 5, 0,
#     0, 0, 6, 4, 0, 0, 7, 0, 0,
#     7, 9, 0, 0, 0, 0, 0, 0, 1,
#     0, 0, 2, 0, 3, 1, 4, 7, 0,
#     1, 0, 0, 0, 5, 0, 0, 0, 8,
#     0, 0, 8, 0, 7, 2, 0, 0, 0]
```

The function `draw_coloured_sudoku` takes a Sudoku `X` and draws it as a graph. For example, Figure 2b is obtained using this function.

```
def draw_coloured_sudoku(X):

    # calculates the size of the Sudoku and creates a graph of such size
    n = int(len(X)**(1/4))
    G = nx.sudoku_graph(n)

    mapping = dict(zip(G.nodes(), X))
    pos = dict(zip(list(G.nodes()), nx.grid_2d_graph(n * n, n * n)))

    # we map the nodes 1-9 to a colormap
    low, *_, high = sorted(mapping.values())
    norm = mpl.colors.Normalize(vmin=low, vmax=high, clip=True)
    mapper = mpl.cm.ScalarMappable(norm=norm, cmap=mpl.cm.Pastel1)

    # draw the graph
    plt.figure(figsize=(6, 6))
    nx.draw(
            G,
            labels=mapping,
            pos=pos,
            with_labels=True,
            node_color=[mapper.to_rgba(i) for i in mapping.values()],
            width=1,
            node_size=500,
    )
```

It can be useful to change the representation of a n-Sudoku from a single list of n^4 items to a list of n^2 sub-lists of size n^2, where each list represents a row. `list_converter` takes a Sudoku in the list form and returns a Sudoku in the list of lists form.

```
def list_converter(X):

    n = len(X)
    a = int(math.sqrt(n))

    sudoku_sol_lol = []

    # for each row

    for i in range(0, a):
        rows = []
        for m in range(a * i, a * i + a):
            rows.append(X[m])
        sudoku_sol_lol.append(rows)

    return sudoku_sol_lol

print(list_converter(unique_3x3[0]))
# [[5, 3, 0, 2, 0, 0, 0, 0, 9],
#  [0, 0, 0, 0, 0, 9, 0, 0, 0],
#  ...,
#  [0, 0, 8, 0, 7, 2, 0, 0, 0]]
```

The function `access(sudoku_list,row,column)` has inputs a Sudoku `X`, and the position of an element denoted by its `row` and `column`, for example (`row,column`) $= (0, 0)$ would indicate the top left element.

`access` returns a list of three lists, each containing the elements that are in the same row, column, or box as the given element.

```
def access(X, row, column):

    n = len(X)
    a = int(math.sqrt(n))
    b = int(math.sqrt(a))

    # convert the list into a lists of lists where each lists represents a row
    sudoku_ll = list_converter(X)

    # obtain row of box
    if row <= a and column <= a:
        # gets the row which is required (1 through n where 1 is the top row and n is
    the bottom row)
        row_contained = sudoku_ll[row]

        # obtain the column
        column_contained = []

        for i in range (0,a):
            # gets the element which is the column value along each list and appends to
    make a new list the column
            column_contained.append(sudoku_ll[i][column])

        # obtain the box which the given element is in
        box_contained = []
        # find the inital row and column
        row_start = int(row / b) * b
        col_start = int(column / b) * b

        for i in range(row_start, row_start + b):
            for j in range(col_start, col_start + b):
                box_contained.append(X[i * a + j])

    return [row_contained, column_contained, box_contained]

print(access(unique_3x3[0],0,0))
# [[5, 3, 0, 2, 0, 0, 0, 0, 9], [5, 0, 2, 4, 0, 7, 0, 1, 0], [5, 3, 0, 0, 0, 0, 2, 0,
    0]]
```

We define `checker`, which takes a Sudoku `X` and returns `True` if it is solved and `False` otherwise.

```
def checker(X):
    n = int((len(X)**0.5))
    flat_sudoku = np.array(X)
    sudoku = np.array([flat_sudoku.reshape(n,n),flat_sudoku.reshape(n,n).transpose()])

    for a in sudoku:
        for array in a:
        # check sum,col row
            if not int(np.sum(array)) == 0.5 * n * (n+1):
                return False
        # check uniq row,col
            if not len(set(array)) == n:
                return False

    b = int(math.sqrt(n))

    box_list = []

    for r in range(0,b):
        for s in range(0,b):
            l = access(X,r*b, s*b)
            box_list.append(l[2])


    for box in box_list:
        if not int(sum(box)) == 0.5 * n * (n+1):
```

```
            return False
        if not len(set(box)) == n:
            return False

    return True

print(checker(unique_3x3[0]))
# False
```

The function `check_solver(solver)` takes as a parameter a Sudoku `solver` function and outputs whether it correctly solves the examples provided.

```
def check_solver(solver):
    for i in range(len(unique_3x3)):
        if checker(solver(unique_3x3[i])):
            print(f'unique_3x3 {i} has correct solution')
        else:
            print(f'unique_3x3 {i} has wrong solution')

    for i in range(len(non_unique_3x3)):
        if checker(solver(non_unique_3x3[i])):
            print(f'non_unique_3x3 {i} has correct solution')
        else:
            print(f'non_unique_3x3 {i} has wrong solution')
```

With all of these functions defined, we are now ready to write code to solve Sudoku puzzles!

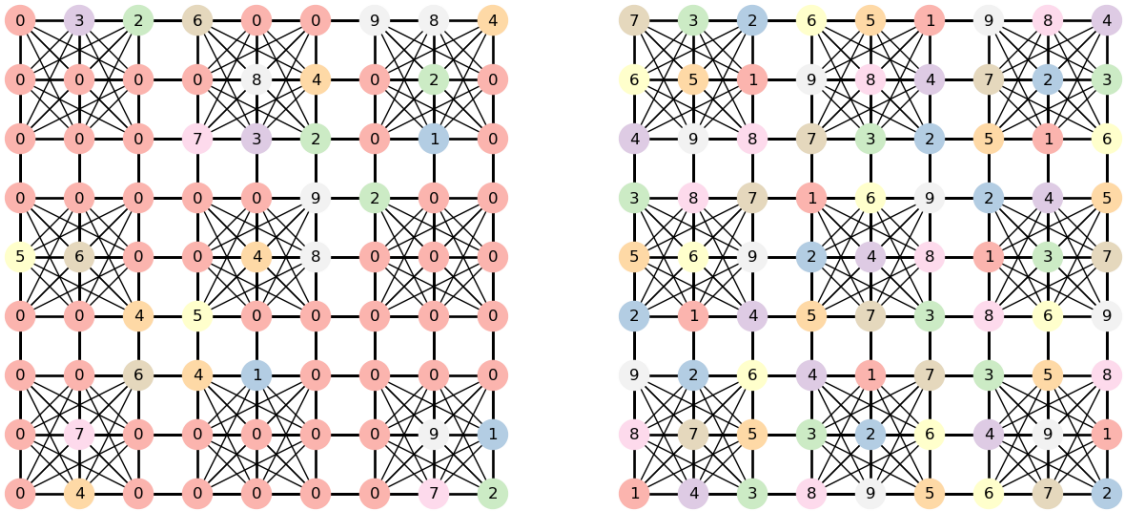
5 Naive Sudoku solver

In this section, we explore how backtracking algorithms can be used to solve Sudoku puzzles.

We adapt Algorithm 1 to define a function `solver_naive` which solves Sudoku puzzles using backtracking and has the option of checking whether the solution is unique. The function expects the parameter `X`, which represents the initial colouring of a Sudoku puzzle, and returns the completed colouring if a solution exists, otherwise it returns `None`. If uniqueness is checked, the function returns a tuple containing a possible solution and whether it is unique. Figure 3 shows a partially coloured Sudoku graph and its solution.

```
def solver_naive(X, check_uniqueness=False):

    n = len(X)
    G = nx.sudoku_graph(int(n ** 0.25))
    # array for solutions
    solutions = []
    k = int(n ** 0.5)

    # dive function
    def dive(v):
        # checks if all vertecies are coloured
        if v == n:
            solutions.append(X.copy())
            # continue search if less than 2 sol found
            return len(solutions) < 2

        # skip already coloured vertices
        if X[v] != 0:
            return dive(v + 1)

        # colour assigning loop
        for col in range(1, k + 1):
            X[v] = col
            # verifies correct solution
            if is_consistent_colouring(G, X, v):
                # If a second solution is found, stop searching
                if not dive(v + 1):
                    return False
            X[v] = 0
```

```
        return True

    dive(0)

    # conditionals for unique solutions
    if check_uniqueness and len(solutions)>1:
        print("Multiple Solutions. One of solutions:")
        # return first solution
        return (solutions[0], False)
    elif check_uniqueness:
        # return first solution or None
        return (solutions[0], True) if solutions else None
    else:
        return solutions[0]

# consistency check
def is_consistent_colouring(G, X, v):
    for neighbour in G[v]:
        if X[neighbour] == X[v]:
            return False
    return True

draw_coloured_sudoku((unique_3x3[1]))
draw_coloured_sudoku(solver_naive(unique_3x3[1]))
```



(a) Partially coloured 3-Sudoku puzzle.

(b) Solution to the 3-Sudoku puzzle.

Figure 3: Visualisation of 3-Sudoku graph colouring before and after Algorithm 1 is applied.

To assign colours, the function utilises a modified version of the `DIVE()` function in Algorithm 1. The embedded function goes through the vertices one by one starting from vertex 0. If a given vertex has a colour already assigned from the initial colouring, then it's skipped and the function moves onto the next vertex; otherwise the function colours the vertex with the first colour and using the helper function `is_consistent_colouring` it checks that the proposed colour is consistent. If it is, then it moves onto the next vertex, otherwise if all colours are inconsistent, then the function backtracks by resetting current and previous colours to 0. The function then tries to colour the given vertex with the second colour, and the above process repeats until either a complete colouring is achieved or no solution exists.

```
check_solver(solver_naive)
# Returns all correct solutions
```

The complexity of a backtracking k-colouring algorithm is $O(a^k)$ where a is the number of possible colours and k is the number of nodes to be filled. A n-Sudoku graph has $O(n^2)$ colours and $O(n^4)$ nodes, hence the time complexity for this algorithm is $O(n^{2n^4})$. This makes it impractical for calculating n-Sudoku with $n > 3$. This inefficiency is further amplified if we desire to check whether the solution is unique. An analysis of the runtime for this algorithm is provided in Section 7.

The design of the function is inefficient as it does not use any prior knowledge during the backtrack search or any specific Sudokus techniques. It instead relies solely on brute force.

6 Improved Sudoku solver

In this section, we explore how we can improve the Naive Sudoku solver using prior knowledge.

An improvement consists in utilising constraints to reduce the space of potential colours for a given vertex. We define a function `solver_improved`, similar to `solver_naive`, which takes a Sudoku puzzle `X` and returns a solution, if one exists. The constraint in the new function is achieved by using `possible_colours(node)`, which takes a vertex and returns a list of its possible colourings. It works by creating a list of the colours of the vertex's neighbours and then the restriction is simply the colours not in that list.

To allow us to investigate if the order in which we choose to colour the vertices makes a difference, we implement a function `select_next_node()`. The main function takes three possible arguments: `default`, `fewest`, and `most`. If the argument is `default`, the next node will be chosen by finding the first uncoloured node. For the remaining options, we first create a dictionary comprising of each non-coloured node and how many possible colourings they have. Then by choosing `fewest` or `most`, we return the node that has the fewest or most number of candidate colours, respectively.

The difference between the new function and the one in Section 5 lies in the `dive()` function, where instead of iterating through all possible colours, we only consider the possible colours using the function `possible_colours(node)`. Furthermore, we choose the next node to consider by calling the function `select_next_node()`.

```
def solver_improved(X, check_uniqueness=False, colouring_order='default'):
    n = int(len(X)**0.25)
    G = nx.sudoku_graph(n)
    # array for solutions
    solutions = []
    # Use the initial coloring
    colours = X.copy()

    def possible_colours(node):
        used = {colours[neighbor] for neighbor in G.neighbors(node) if colours[neighbor]
     != 0}
        return [color for color in range(1, n*n + 1) if color not in used]

    def select_next_node():
        # Selects the next node based on the colouring_order
        if colouring_order == 'default':
            for node in G.nodes():
                if colours[node] == 0:
                    return node
            return None  # No uncolored nodes left

        # For both fewest and most candidate colours, calculate remaining values
        remaining_values = {node: len(possible_colours(node)) for node in G.nodes() if
    colours[node] == 0}

        if colouring_order == 'fewest':
            return min(remaining_values, key=remaining_values.get, default=None)
        elif colouring_order == 'most':
            remaining_values_inverted = {key: n*n - remaining_values[key] for key in
    remaining_values}
            return min(remaining_values_inverted, key=remaining_values.get, default=None
    )
```

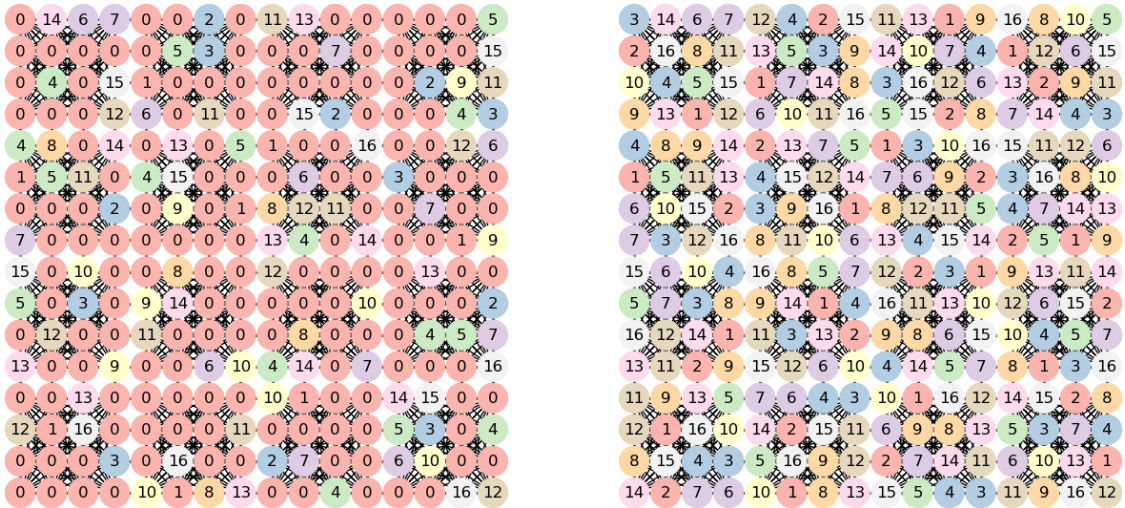
```
    def dive():
        if all(colours[node] != 0 for node in G.nodes()):
            solutions.append(colours.copy())
            return len(solutions) < 2

        node = select_next_node()
        if node is None:
            return True

        for color in possible_colours(node):
            colours[node] = color
            if not dive():
                # Stop search if a second solution is found
                return False
            # Reset on backtrack
            colours[node] = 0
        return True

    dive()

    if check_uniqueness and len(solutions) > 1:
        print("Multiple Solutions. One of the solutions:")
        return (solutions[0], False)
    elif check_uniqueness:
        return (solutions[0], True) if solutions else None
    else:
        return solutions[0]
```

The function correctly solves all Sudoku examples.

```
check_solver(solver_improved)
# Returns all correct solutions
```

Furthermore, we are finally able to solve a 4-Sudoku, albeit the function is slow and does take about 10 seconds to run. Figure 4 shows a picture of it.

```
draw_coloured_sudoku(puzzle_4x4)
draw_coloured_sudoku(solver_improved(puzzle_4x4, colouring_order='fewest'))
```



(a) Partially coloured 4-Sudoku puzzle.

(b) Solution to the 4-Sudoku puzzle.

Figure 4: Visualisation of 4-Sudoku graph colouring before and after `solver_improved` is used.

An analysis of the runtime for this algorithm and how it compares to the Naive Sudoku solver is provided in Section 7.

7 Runtimes

In this section, we measure the time our functions take to solve a Sudoku.

The time measured depends on many factors such as the specific machine used, the Python version, etc. The aim of this analysis is to compare the two functions (they are run in the same environment) and is not an absolute measure of how fast they are.

We measure the time for the functions `solver_naive` and `solver_improved` to solve all the Sudoku puzzles in our example lists `unique_3x3` and `non_unique_3x3`. For `solver_improved`, we measure the time using all three arguments `fewest`, `default`, and `most` for the order of colouring. Then, we calculate the average time to solve one Sudoku puzzle and plot a bar chart with our results, as shown in Figure 5.

```
from time import process_time
import matplotlib.pyplot as plt
from statistics import mean
%matplotlib inline

test = unique_3x3 + non_unique_3x3
G = nx.sudoku_graph(3)
time_taken_1 = []
time_taken_2a = []
time_taken_2b = []
time_taken_2c = []

for sudoku in test:

    #1.
    start_time_1 = process_time()
    solver_naive(sudoku)
    end_time_1 = process_time()
    time_taken_1.append(end_time_1 - start_time_1)

    #2a.
    start_time_2a = process_time()
    solver_improved(sudoku, colouring_order='default')
    end_time_2a = process_time()
    time_taken_2a.append(end_time_2a - start_time_2a)

    #2b.
    start_time_2b = process_time()
    solver_improved(sudoku, colouring_order='fewest')
    end_time_2b = process_time()
    time_taken_2b.append(end_time_2b - start_time_2b)

    #2c.
    start_time_2c = process_time()
    solver_improved(sudoku, colouring_order='most')
    end_time_2c = process_time()
    time_taken_2c.append(end_time_2c - start_time_2c)

t_1 = mean(time_taken_1)
t_2a = mean(time_taken_2a)
t_2b = mean(time_taken_2b)
t_2c = mean(time_taken_2c)

times = {'Q1': t_1, 'Q2a': t_2a,'Q2b': t_2b,'Q2c': t_2c}
Q = list(times.keys())
time_values = list((times.values()))
colours = {"#006400", "#FF0000", "#FF1493", "#000080"}
fig = plt.figure(figsize = (7,6))

plt.bar(Q, time_values, color = colours, width = 0.5)
plt.xlabel("Function to solve sudoku puzzle from each question")
```

```
plt.ylabel("Mean runtime (s)")
plt.show()
```

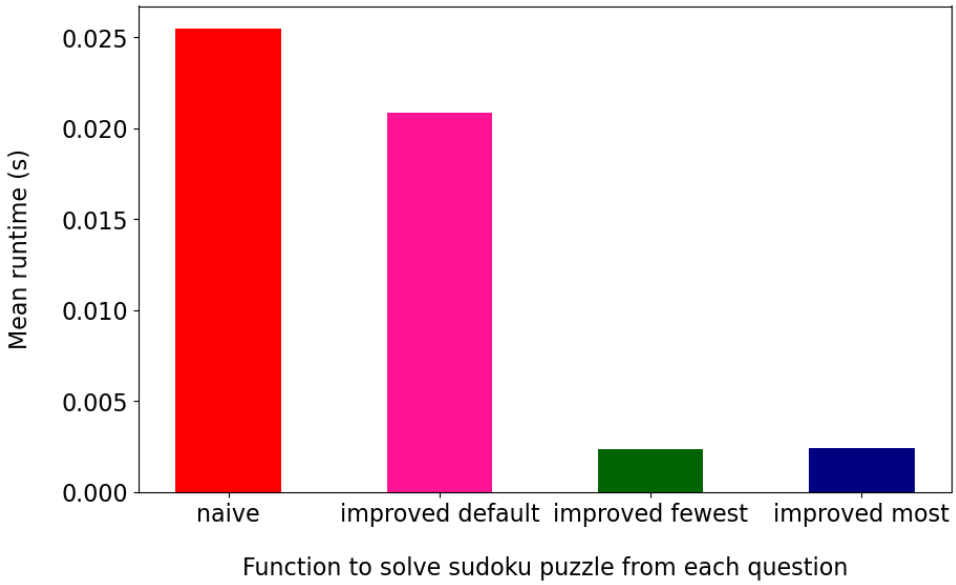


Figure 5: Bar chart of average runtimes for each function to solve a Sudoku puzzle.

As expected, the naive solver takes the most amount of time. The function `solver_improved` with any parameter returns a solution quicker than the function `solver_naive`. Using the parameters `fewest` and `most` shows a substantial improvement to using the `default` option. It is to be expected as we add a further constraint to the function. The difference between `fewest` and `most` is small, and could be due to minor fluctuations and inaccuracies in the way we obtain the runtimes.

8 Conclusion

In this project, we view Sudoku as mathematical objects and leverage mathematical tools to solve a Sudoku puzzles. In Section 2, we start by exploring how the Sudoku symmetric group allows us to study how many Sudoku exist and how they permute. In Section 3, we show that Sudoku can be viewed as graphs and that a Sudoku puzzle is equivalent to a graph colouring problem, introducing the backtrack search algorithm to solve it. In Section 4, we give the implementation details of how we can work with Sudoku in Python. In Section 5, we implement a backtracking algorithm to solve Sudoku puzzles. In Section 6, we write an improved function which utilises constraints to improve its efficiency. In Section 7, we measure the time our functions take to solve a Sudoku.

Future work includes: a discussion of p and np problems and how they relate to Sudoku solvers; a computational exploration of the Sudoku symmetric group and its permutations; and further improvements to the Sudoku solver, such as applying techniques which are used to solve Sudoku by hand.

A Computational exploration of permutations of Sudoku

This section expands on ideas established in Section 2 on 'essentially different' Sudoku. Whilst the the total number of Sudoku is extremely large, this can be massively reduced by considering two Sudoku 'morphs', whose definition is declared below.

There are a number of operations that can be performed on an n-Sudoku X which leave it as a valid Sudoku. Here are three of them:

1. We can permute the values: if f is a permutation of $\{1, 2, ..., n^2\}$, then applying f to each value in a cell in X gives another valid Sudoku.
2. We can permute rows and/or columns within a "band" or "stack", where a band is a "row of boxes", and a stack is a "column of boxes". For example, in a 3-Sudoku we can swap rows 0 and 1, or 0 and 2, but not necessarily rows 0 and 3; similarly, we could "rotate" the three middle columns $3 \to 4 \to 5 \to 3$ (see Section 2).
3. We can transpose the Sudoku.

We define a morph of X to be an n-Sudoku obtained by applying some combination of these operations.

For example, a morph of X could be obtained by swapping rows $1 \leftrightarrow 2$ and rows $6 \leftrightarrow 8$, swapping columns $3 \leftrightarrow 4$, transposing the result, and then permuting the values $4 \to 2 \to 5 \to 4$.

The huge amount of possible permutations makes the study of morphs difficult, we leverage the power of computers once again. The aim of this section is to write a function to determine whether two n-Sudoku are morphs.

We will assume that any morph is obtained by a row permutation, followed by a column permutation, followed by a transposition, followed by a permutation of values. And we note that in this section we deal with completed n-Sudokus, rather than puzzles.

We start by defining functions that apply the operations described above.

The function `to_matrix` takes a Sudoku `X` and returns its `numpy` matrix representation. Then the functions `row_perm` and `column_perm` take a Sudoku in the matrix form `M` and the parameter `order` and return a Sudoku in the matrix form with the new `order`. The parameter `order` is expected to be a list representing the r-cycle permutation of the row or columns. For example, if we want to swap the middle columns $3 \to 4 \to 5 \to 3$ of a 3-Sudoku, then we set `order = [0,1,2, 5,3,4, ,6,7,8]`. The function `transpose` takes a Sudoku in the matrix form `M` and the boolean parameter `marker` and return the transpose of `M` if `marker = True`, otherwise returns `M`. Finally, the function `element_perm` takes a Sudoku `X` and the parameter `order` as described above to permute each of the elements in the Sudoku.

```
import itertools as it

def to_matrix(X):
    n = int(np.sqrt(len(X)))
    # convert the list into array and reshape the array into a 'matrix' form
    return np.array(X).reshape(n, n)

def row_perm(M, order):
    # change the order of the rows
    return M[order]

def column_perm(M, order):
    # change the order of the columns
    return M[:,order]

def transpose(M, marker):
    if marker:
        # transpose the Sudoku
        M = np.transpose(M)
    return M

def element_perm(X, order):
    # elements are 1 to 9 while indices are 0 to 8 so reduce by 1 to match indices
    X = [x-1 for x in X]
    # identity r-cycle
    original =  np.array([x for x in range(len(order))])
    # reduce items by 1 to match indices
    order = np.array([x-1 for x in order])

    # calculates by how much we need to increase or decrease
    swapby = order - original

    # inc / dec all elements by the necessary amount
```

```
    for i in range(len(X)):
        X[i] += swapby[X[i]]
        i += 1

    return([x+1 for x in X])


def to_list(M):
    # converts back the array to list
    return M.ravel().tolist()
```

We may generate a morph using a combination of these operations. For example, consider a 3-Sudoku

```
X =  [5, 4, 9, 1, 3, 6, 8, 2, 7,
      2, 6, 7, 5, 8, 9, 3, 4, 1,
      3, 8, 1, 4, 2, 7, 5, 9, 6,
      7, 1, 4, 8, 9, 2, 6, 5, 3,
      9, 3, 5, 6, 4, 1, 7, 8, 2,
      8, 2, 6, 3, 7, 5, 9, 1, 4,
      6, 9, 3, 2, 5, 4, 1, 7, 8,
      1, 5, 2, 7, 6, 8, 4, 3, 9,
      4, 7, 8, 9, 1, 3, 2, 6, 5]
```

and apply the operations as described in our earlier example

```
M = to_matrix(X)
# Swap rows 1,2 and 6,8
M = row_perm(M, [0,2,1, 3,4,5, 8,7,6])
# Swap columns 3,4
M = column_perm(M, [0,1,2, 4,3,5, 6,7,8])
# Transpose the result
M = transpose(M, True)
X = to_list(M)
# Permuting the values 4,2,5,4
X = element_perm(X,[1,5,3,2,4,6,7,8,9])

print(X)
# [4, 3, 5, 7, 9, 8, 2, 1, 6,
#  2, 8, 6, 1, 3, 5, 7, 4, 9,
#  9, 1, 7, 2, 4, 6, 8, 5, 3,
#  3, 5, 8, 9, 2, 7, 1, 6, 4,
#  1, 2, 4, 8, 6, 3, 9, 7, 5,
#  6, 7, 9, 5, 1, 4, 3, 8, 2,
#  8, 4, 3, 6, 7, 9, 5, 2, 1,
#  5, 9, 2, 4, 8, 1, 6, 3, 7,
#  7, 6, 1, 3, 5, 2, 4, 9, 8]
```

Generation of morphs is one face of the coin. We now create a function that detects whether two Sudoku are morphs.

Consider how many permutations of a 3-Sudoku there are when we use all the allowed operations (transpose, permutation of rows/columns within band/stack, and permuting value labels):

- Transpose: 2.
- Row permutation within stack: $3 * 3! = 216$.
- Column permutation within band: $3 * 3! = 216$.
- Value Permutation: $9! = 362880$.

When multiplying these together, we get a pretty big number: 33861058560.

So far, we allowed our functions to accept any parameter `order` and we made sure the operation was valid by manually checking. When detecting, we need to make sure the operations are valid. We define a function `possible_perms(n)` that creates all the "legal" `order` for row and column permutations of a `n`-Sudoku.

```
def possible_perms(n):
    # generate a list of n-lists where each inner list contains n indices
    indices = [list(range(i*n, (i+1)*n)) for i in range(n)]
    # permute inside the inner lists since these are the allowed permutations
    perms = [list(it.permutations(nums)) for nums in indices]
    # calculate the cartesian product for the permutations
    L = np.array(list(it.product(*perms)))
    # flatten and return a list of the orders we can use in the permutation functions
    return L.reshape(-1, n*len(indices[0])).tolist()
```

The function `all_perms(M)` takes a Sudoku in matrix form `M` and uses the operations we defined (apart from the permutation of elements) to return all the possible morphs.

```
def all_perms(M):
    num = possible_perms(int(np.sqrt(len(M))))
    # produce all the possibilities for a single Sudoku
    # first transposing, then row perms, then column perms
    for marker in [True,False]:
        res0 = transpose(M,marker)
        for orders in num:
            res1 = row_perm(res0,orders)
            for orders in num:
                res2 = column_perm(res1,orders)
                yield res2
```

We could now detect if two Sudokus are morphs, provided there is no value permutation involved. We still need to include the possibilities from permuting values. To avoid the factorial cases, we define a function `position_matrices(M)` to produce n^2 so called "position matrices".

Consider a n-Sudoku represented by $\texttt{M} = [m_{ij}]_{n^2 \times n^2}$. Then a position matrix for the index $k = 1, \ldots, n^2$ is a matrix $[c_{ij}]_{n^2 \times n^2}$ such that $c_{ij} = \texttt{True}$ if $m_{ij} = k$ and `False` otherwise. The motivation is that the ability of permuting values doesn't depend on the values themselves but depends on their relative positions.

```
def position_matrices(M):

    positions = []
    # produce the 'truth-table' matrices for every number in the Sudoku
    for i in range(1, len(M)+1):
        pos = (M == i)
        positions.append(pos)

    return positions

position_matrices(np.array([[1,2],
                            [1,2]]))
#[array([[ True, False],
#        [ True, False]]),
# array([[False,  True],
#        [False,  True]])]
```

All the possibilities of the permutations can be obtained by applying `all_perms(M)` on `position_matrices(M)`.

Finally, `comparison(U,V)` takes two n-Sudoku and returns whether they are morphs. The function first generates all the possible permutations of the position matrices for `U` and then it compares them to the position matrices for `V`. If there is an overlap, then the functions returns `True`, otherwise it returns `False`.

```
def comparison(U,V):
    # convert to matrices
    X = to_matrix(U)
    Y = to_matrix(V)

    # all permutations for all position matrices for the Sudoku X (= all possibilities)
    x_possibilities = [list(all_perms(x)) for x in position_matrices(X)]

    # position matrices for the Sudoku Y
    y_pos = position_matrices(Y)

```

```
    # initialize the dive-indices as ALL of possibilities
    dive = range(len(x_possibilities[0]))

    # check the x_possibilities, for the dive indices, if any member of Y's position
    matrices matches one of them
    for xpos in x_possibilities:
        indices = [i for i in dive
                        if any(np.array_equal(xpos[i], member_y) for member_y in y_pos)]
        if not indices:
            return False
        # upate the dive indices and go back to the loop with restricted solutions
        dive = indices.copy()

    # return True if there is a possibility that is identical where all of Y's position
    matrices match
    return True
```

As an example, if we have two 3-Sudoku U and V in the form of a list of length 81, calling the `comparison(U,V)` function would first convert the Sudokus to two 9×9 `numpy.array`, generate all the possible permutations of U's position matrices (a list of length 9, each member containing 93312 permutations). Then we generate V's position matrices, and

1. Check whether one of V's position matrices exactly matches one of U' possibilities (in the inner list).
2. If it matches, update the dive indices to restrict the choices.
3. Otherwise keep going to the next inner list and check the corresponding dive indices hoping we can find an index j that all 9 j-th matrices from each list are equivalent to V's position matrices.
4. If there is indeed a match, return `True`, if we go through all the possibilities and still do not find any, return `False`.

```
print(comparison(X,X_morph))
# True

print(comparison(X,unique_3x3[1]))
# False
```

The function as expected returns `True` when two Sudoku are morphs and `False` when two Sudoku are not morphs.

`comparison(U,V)` is quite slow since the number of allowed permutations grows factorially.

We end this section by noting that a n-Sudoku puzzle cannot have a unique solution if it contains fewer than $n^2 - 1$ entries.

An n-Sudoku takes n^2 different values for the completion of the Sudoku. If the puzzle contains fewer than $n^2 - 1$ entries in the initial hints, the 'unused' entries can be interchanged with each other in order to produce a different Sudoku solution. For example if an initial puzzle for a 3-Sudoku uses values 1-7, the positions of all the 8s and all the 9s could be swapped in order to produce an equally valid Sudoku, hence the set of hints cannot produce a unique solution.