Sudoku Solving In Python

by

Adam Aldridge

Abstract

The aim of this project was to write a program that is able to solve Sudoku puzzles with pen and paper style methods. In this report, a variety of Sudoku solving methods will be explored in detail. Then it will be explained how these methods were programmed in Python and combined to solve Sudokus. The program is capable of solving square Sudokus of any size, although for some that are bigger than the traditional 9 by 9 this can take a significant amount of time. It is able to find the single solution of any uniquely solvable Sudoku puzzle. The program can also find every solution to Sudokus which have more than one.

Contents

1 Introduction

Sudoku is a type of logic based number placement puzzle. They are in the form of square grids of cells, usually 9 rows, 9 columns (although other sized square puzzles will be discussed). These grids are also split into 9 smaller square boxes, each containing 9 cells. No row, column or box can contain more than one of a number. For a puzzle to be solved each row, column, and box must have every number from 1 to 9 contained in exactly one of their cells. When attempting to solve a Sudoku, some cells will already contain numbers. Looking at these with the rules in mind can give insight into possible values to fill empty cells.

Difficulty

Depending on how many and which cells already contain numbers, different Sudoku puzzles can have varying difficulty. Generally Sudokus with less clues are harder. An easy puzzle with lots of clues may be solved by simply considering the rule that a row, column or, box can only contain one of each number. More challenging Sudokus with less clues will likely require more involved concepts and methods to solve. The difficulty of a Sudoku can be classified by the complexity of solving methods that are required to solve it.

Uniqueness

A given Sudoku may have a single solution (this is the case for most given puzzles), however it is possible for Sudokus to have multiple solutions. If this is the case, the number of possible solutions will depend on the cells that are initially filled. Generally the more initially given values, the less solutions there will be. A problem could have two solutions or, in the extreme case of no initial clues, a 9×9 Sudoku will have 6,670,903,752,021,072,936,960 solutions [2], an empty 4×4 puzzle will have 288. This includes solutions which are symmetrically equivalent to others. The minimum number of clues required to create a 9×9 Sudoku puzzle with a unique solution is 17 [3].

1.1 Nomenclature Used

House
A set of all cells contained in any single row, column or, box. In a 9 by 9 Sudoku each house contains 9 cells.

Candidate
A candidate for a cell is a number that is not known to be a false placement in that cell.

Candidate list
For a specific cell, this is a list containing all of the numbers which are candidates for that cell.

n
n will refer to the side length of a box, For a 9 by 9 Sudoku, $n = 3$.

Index's
In the program, it is important to be able to refer to specific cells and houses easily. To enable this, each cell, row, column, and box in a Sudoku is given its own identity. Rows and columns are numbered $1, \ldots, n^2$. Each cell is labeled based on which row and column it lies in, that label takes the form, (i_{row}, j_{col}) for $i, j \in \{1 \leq x \leq n^2 \mid x \in \mathbb{Z}\}$. Box indices are similar to those of cells however, instead of being identified by a row and column, a box is labeled with its resident box-row and box-column.

For a 9 by 9 grid, cells and boxes are labeled as follows:

(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)	(1,9)
(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)	(2,9)
(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	(3,8)	(3,9)
(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	(4,8)	(4,9)
(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	(5,8)	(5,9)
(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)	(6,8)	(6,9)
(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)	(7,8)	(7,9)
(8,1)	(8,2)	(8,3)	(8,4)	(8,5)	(8,6)	(8,7)	(8,8)	(8,9)
(9,1)	(9,2)	(9,3)	(9,4)	(9,5)	(9,6)	(9,7)	(9,8)	(9,9)

Cell Indices

(1,1)	(1,2)	(1,3)
(2,1)	(2,2)	(2,3)
(3,1)	(3,2)	(3,3)

Box Indices

2 Solving Theory

2.1 General Solving Ideas

There are a large number of methods available to solve Sudokus, many of these use an approach of logical deduction. Solving a Sudoku only with logical methods can be satisfactory, but unfortunately, not every Sudoku can be solved with logic alone. For those than cannot, a process of trial and error must be adopted.

Logical Deduction

Generally, logical methods work by seeking the satisfaction of special conditions in the candidate lists of particular cells. Some of these methods can be used to allocate a value in a particular cell. Other logical methods are just able to reduce the candidate lists of cells, these methods cannot solve a Sudoku on their own, however, reducing candidate lists aids the methods which fill cells.

Trial And Error

Methods involving trial and error work by repeatedly trying different candidates for empty cells until a solution is found. There are several ways of doing this, some more efficient than others. Trial and error alone can solve any Sudoku but in many cases will take an unreasonable amount of time. To amend this, it is possible to speed up the process dramatically by combining trial and error with logical methods.

The Best Order To Apply Solving Methods

There are multiple solving methods that can be used, but what is the best order to apply them? A goal in writing this program was to make it solve Sudokus as fast as possible. To this aim, the best order will be the one that is most time efficient. Some solving methods are less complex that others which will make them work faster. Generally, logical methods will act faster than trial and error methods. With this in mind, the program will first apply logical solving methods to progress a given Sudoku as far as possible. If it gets stuck, trial and error methods will be used to fill in the gaps.

2.2 Logical Methods To Allocate Numbers

Naked Single

[4] If a cell has only one candidate remaining in its candidate list, that candidate is the only possible number which can fill the cell. When this case is present, the number in question is called a Naked Single and it is placed in the relevant cell.

Hidden Single

[4] Considering the rule that each house must contain exactly one of each number in the set: $\{1 \leq x \leq n^2 \mid x \in \mathbb{Z}\}$. If a candidate for an empty cell in a house is not present in any other candidate list within that house then, that number must belong in the empty cell.

2.3 Logical Methods To Eliminate Candidates

Markup

[2] Marking up is a simple but important method of reducing the candidate lists of empty cells. It uses the rule that each house cannot contain more than one of any number. If a house has a value V filling one of its cells, no other cell in that house can contain V. With this in mind, every empty cell in that house can have V removed from its candidate list.

Each cell in a Sudoku occupies three houses; a row, column and a box. And so, every filled value in an unfinished Sudoku can be taken out of the candidate lists of all the empty cells which share a row, column or, box with the cell holding that value. This process is referred to as marking up.

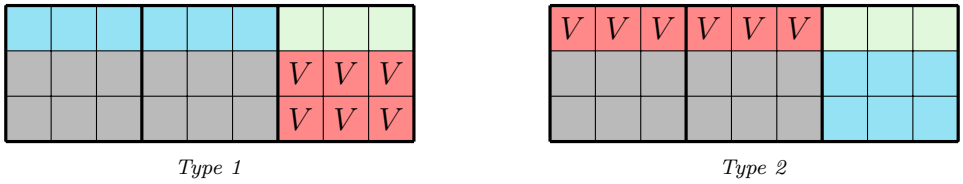
Box - Line Intersections

[4] Also known as Locked Candidates this method uses intersections between boxes and rows or columns. If a value V lies only in the candidate lists of a subset of the cells in a house ($house_1$), one of those cells in that subset must contain V. If that subset of cells in $house_1$ lie only in the intersection between $house_1$ and another house ($house_2$), one of those empty cells in the intersection must contain V. Because it is now known where V will lie in $house_2$, it is deducible that any empty cell in $house_2$ which is not in the intersection cannot contain V. And so, each of those empty cells can have V removed from their respective candidate lists.

In Sudoku grids, house intersections only occur between lines and boxes or, rows and columns (lines being either rows or, columns). However, this method is irrelevant between rows and columns because if the appropriate scenario arises in a row-column intersection, the value in question would be a hidden single for the lone cell in the intersection.

When considering box-line intersections, this method comes in two flavors. There has to be one house where, a specific digit is only a candidate for empty cells within the intersection. One variety of the method *(Type 1)* is where this house is the line. The other *(Type 2)* is where this house is the box. If both the line and the box satisfy this property, the method cannot deduce anything.

Examples Of Type 1 And Type 2 Box-Line Intersections.

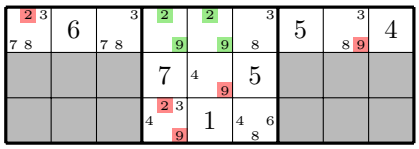


Naked Subsets

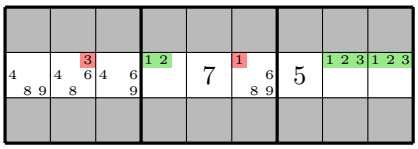
[4] Say in a house there is a group of X empty cells (lets call it: $group_X$). Now if the candidate lists of those X cells are spanned by a subset of exactly X digits. That is to say, each of the cells in $group_X$ only has candidates which are elements of this subset, we call this subset a Naked Subset. Since the number of candidates in the subset is equal to the number of cells which only have candidates appearing in that subset, it won't be controversial to assume, that each of the cells in $group_X$ must

be filled by one of the digits in the Naked Subset. Furthermore, since each digit can only fill one cell in any given house, any cell that is not in $group_X$ cannot contain any digit in the Naked Subset. This result is useful as it allows us to remove those candidates from the cells which are in the house but outside $group_X$.

Examples Of Naked Subsets.



Naked Pair



Naked Triple

Candidates in Naked Subset.

Candidates that can be eliminated.

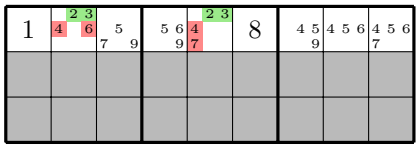
In the literature, different sized Naked Subsets usually have different names. A subset of length two for example is usually called a Naked Pair, length three is a Hidden Triple and so on. My source for this method only mentioned subsets up to a length of 4 digits (Naked Quad). Despite this, larger subsets are still valid.

To build on the researched material this shall be explored. By the above definition, The length (X) of a Naked Subset in a house will be in the range: $1 \leq X \leq E$ (where E is the number of empty cells in that house). However, through inspection it can be found that not all of these cases are helpful to us. If for example $X = E$, nothing can be determined as there are no empty cells that are simultaneously in the house and outside the group spanned by the Naked subset. Consider also the case of $X = 1$, here the single digit in the Naked Subset would automatically satisfy the condition to be a Naked Single, so regarding this method $X = 1$ is also a useless case. With these eliminations, the length (X) of a useful Naked Subset will be in the range: $2 \leq X \leq (E - 1)$ (for a house in a standard Sudoku with all its cells empty, this range is: $2 \leq X \leq 8$).

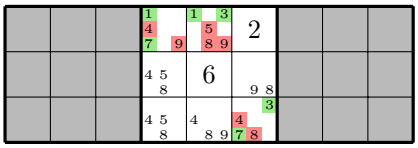
Hidden Subsets

For a given house, if there is a subset of say X digits, and each of those digits is only a candidate for cells which are in a group of X empty cells in the house. This subset of digits is called a Hidden Subset. It can be reasoned that all of those cells in the group must be filled by one of the digits in the Hidden Subset. It follows that none of cells can contain any digit other than those in the Hidden Subset. Knowing this, if any of the cells in the group have candidates which are not in the Hidden Subset, those candidates can be eliminated from the candidate lists belonging to those cells.

Examples Of Hidden Subsets.



Hidden Pair



Hidden Triple

Candidates in Hidden Subset.

Candidates that can be eliminated.

Building on the researched material as before, the possible lengths of a Hidden Subset will be investigated. Similarly to a Naked Subset, by the above definition of a Hidden Subset, the length (X) of the

subset in a given house will be in the range: $1 \leq X \leq E$ (where again, E is the number of empty cells in that house). Like for a Naked Subset, some cases will be uninteresting. Clearly if $X = E$ nothing can be deduced. If $X = 1$ the only candidate in the Hidden Subset will be a Hidden Single for its only resident cell. As well as these, there is another useless case for the length of a Hidden Subset; when $X = E - 1$ there will be one cell in the house that is not spanned by the Hidden Subset. This cell can by definition only have one candidate thus making is a Naked single. Concluding, we have the length range of possible Hidden Subsets to be: $2 \leq X \leq (E - 2)$

When seeking Hidden Subsets computationally some are easier to find than others. It is notably more simple to find a subset where all of its digits are present in every cell spanned. For subsets of length $3 \leq X$ this is not always the case. For example, if a Hidden Subset is say: $[1, 2, 3]$, the three spanned cells could have candidate lists; $[1, 2, 4]$, $[1, 3, 4]$ and, $[2, 3, 4]$ (The reason for this difficulty will be explained later in the report). Interestingly this same complication does not make it harder to find Naked Subsets.

2.4 Trial And Error Methods

Both trial and error methods described below use a common idea, they try placing one of the possible values for a cell in to that cell, then they observe how this allocation affects the rest of the puzzle. Placing a number in a cell will yield one of three possible outcomes:

1. The Sudoku will be solved;
2. The filled value will be the wrong number for that cell;
3. Cells are still empty and the placed value is not known to be false.

When considering a case where no logical methods are known; the candidate list for every cell will be $[1, \ldots, n^2]$, outcome 1 can only happen if the cell being filled is the last remaining empty cell of the puzzle, and outcome 2 will only occur if the value placed in the cell is already filled into one of the cell's three resident houses. In this case where logical methods are not considered; 3 will be to most common outcome to guesses, and solving a Sudoku with this idea will take a very long time. Thankfully there is a better way.

When a 'guess' value is allocated to a 'trial-cell', the candidate lists of the cells in that trial-cell's three resident houses can all have that value removed from them. Once this is done, other logical methods may also be able to act on this new version of the Sudoku; more candidate lists could be reduced, or maybe more values can be placed in cells. Basically, applying logical methods after a guess is made can fast forward the puzzle to a solution or at least helpfully progress it. Doing this may also bring the Sudoku to a position where some cells have no possibilities, in this case, the guess must have been false. Both methods described bellow use logical methods in this way.

Elimination By Contradiction

[5] Proof by contradiction is a well known procedure in mathematics to deduce whether a statement is true. It works by first, assuming that the opposite of the statement is true then, if some contradiction is found with that assumption, the statement must be true.

A somewhat similar process can be applied to the solving of puzzles. To aid the solving of a Sudoku, the candidate lists of empty cells can be reduced by the elimination of possibilities. To justly the removal of a value from a given candidate list, it is required to prove that the value cannot fill that cell. To do this, the method of proof by contradiction can be applied. First, assume that the number can fill the cell by placing it in that cell, next seek a contradiction by applying logical methods. If one is found, it has been proven that the chosen value cannot fill that cell. Following this, the trial-value can be eliminated from the candidate list of that cell.

The Elimination By Contradiction method works in this way by repeatedly placing trial-values in cells until one is found to cause an error.

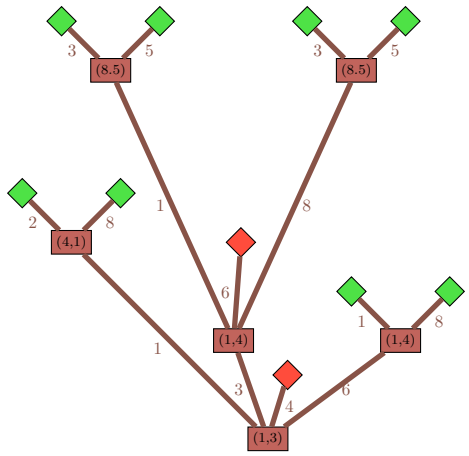
Branching Tree

Compared to Elimination By Contradiction, this method employs a more direct approach. The Branching Tree process works by choosing trial-values and applying logical methods repeatedly until every possible combination of candidate arrangements is considered. Doing this will find the solution to any given Sudoku, it will also find all the solutions for a Sudoku with more than one.

The potential combinations of trial-values in empty cells can be visualized as a tree. To look through all of the possible branches in this tree, the method follows a systematic procedure. As mentioned above, when a trial-value is allocated to a cell, one of three outcomes are possible; a solution is found, an error is found or, the puzzle is stuck with more empty cells. In this method, when the outcome to a guess is that the puzzle is stuck, another trial-value is allocated to a different empty cell. This is repeated until a solution or an error is found, solutions and errors are like the leaves of the tree and each trial-value is like a branch. Some branches end in leaves, others end in forks leading to more branches. When this method arrives at a leaf, it moves its attention to the previous fork, which is representative of the empty cell where the last guess was made. In that cell, one of the other possible candidates is placed as a trial-value. This branch is followed until all of its ends are leaves, then another trial-value is tried, and another. This is repeated until all the possibilities for that cell are exhausted. When this happens, the procedure 'backtracks' to the fork before that one, all of its possibilities are then tried as well. This process of exhausting and backtracking is repeated until all possible combinations of candidates for cells in the Sudoku have been tested, thus, all the leaves of the tree are found.

For uniquely solvable Sudokus there will be only one leaf in this tree which corresponds to a solution, the rest are errors. However, for a Sudoku with X solutions there will be X leaves on the tree which represent solutions.

Here is an example of a tree that was used to solve a real Sudoku with eight solutions:



Green diamonds represent solutions, and red diamonds represent errors. The order of guesses that were made is as follows:

try 1 in (1, 3) outcome: stuck,
try 2 in (4, 1) outcome: solution,
try 8 in (4, 1) outcome: solution,
try 3 in (1, 3) outcome: stuck,
try 1 in (1, 4) outcome: stuck,
try 3 in (8, 5) outcome: solution,
try 5 in (8, 5) outcome: solution,
try 6 in (1, 4) outcome: error,
try 8 in (1, 4) outcome: stuck,
try 3 in (8, 5) outcome: solution,
try 5 in (8, 5) outcome: solution,
try 4 in (1, 3) outcome: error,
try 6 in (1, 3) outcome: stuck,
try 1 in (1, 4) outcome: solution,
try 8 in (1, 4) outcome: solution.

You may notice that the order in which the algorithm works is similar to the order used when converting a propositional logical tree into polish notation.

Before the process of trying values is begun, the order in which cells will be considered is decided. This is useful when a trial-value leads to more guesses because the next cell to look at is already chosen. With the use of logical methods, after a trial-value is applied, it is often the case that other empty

cells are filled. When this happens, those cells are skipped in the order when a cell is being chosen. As well as this, the use of logical methods can act to deduce that a trial-value leads to an error long before it would have been known if it was not for logical methods. Doing this reduces the number of total combinations to test, its a bit like pruning the tree.

3 Structure And Data Types

Solving a Sudoku by hand is a simple case of; looking at the puzzle, thinking, penciling in numbers, and repeating till the Sudoku is solved. Computers are not good at looking and thinking, so when programming a Sudoku solving algorithm a different approach must be taken. A given puzzle must be converted into form that a computer can understand and manipulate. With this in mind, the Python program carries out the following steps to solve a given puzzle:

1. **Receives a Sudoku from the user in the form of a list of lists.**
   Each inner list corresponds to a row of the given puzzle with every value in those lists corresponding to a cell. If a value is 0 then the cell is empty.
2. **Converts the list of lists into dictionaries that contain more specific information.**
   Detailed of possible candidates for empty cells are contained in `cell_dict`. Sets of filled values in each house are held in `ref_dict`.
3. **Manipulates the dictionaries using several logical Sudoku solving methods.**
   Methods are applied with functions that are used repeatedly in a specific order. This goes on until the problem is solved or, no further progress can be made.
4. **The Sudoku is converted out of dictionary form and gets printed to the console.**
   The result will either be a solved puzzle or an unsolved puzzle representing the progress made (if any).

The bulk the program is largely dedicated to the functions which apply solving methods. Each of these functions works in a different way, but most of them share the same pattern. The majority create a dictionary, separate from `cell_dict` and `ref_dict` then, consider its contents with some logical ideas. The dictionary used for solving can be different for each function, however many of them require the same one, which is `cell_cand_pos`.

3.1 The Specific Form Of Dictionaries Used

`cell_dict`

Contains information for each cells in the Sudoku grid. Every cell on the puzzle has its own corresponding element of `cell_dict`. Each element is of the form:

$$Cell: \ Value$$

If the cell is filled and contains a value, or

$$Cell: \ [C_1, \ldots, C_m]$$

If the cell is empty. C_i for $i \in \{1, \ldots, m\}$ are the candidates for the empty cell. The key of each element ($Cell$) is an index for the cell. It is a pair of numbers corresponding to the cell's row and column taking the form: (r, c) for $r, c \in \{1, \ldots, n^2\}$.

ref_dict

Acts as a reference for which values have been confirmed in each house. Every house in the Sudoku has a corresponding element in `ref_dict` of the form:

$$House: \ [V_1, \ldots, V_m]$$

V_i for $i \in \{1, \ldots, m\}$ are the values confirmed in the house. The keys ($House$) are of the form: (*house type*, *house index*), where *house type* is a string; `'row'`, `'column'` or, `'box'`. *house index* is the index identifying the specific house. As specified in §1.1, it is either an number (for rows and columns) or, a pair of numbers (for boxes).

cell_cand_pos

As mentioned above this dictionary is used by some functions in the program to employ their solving method. `cell_cand_pos` holds information on; which houses contain empty cells, which empty cells they contain and, what the candidates for those cells are. It is useful as an an aid to applying solving methods because, it allows the functions to specifically target only the houses which contain empty cells, rather than looking at all houses. This can save on computing time. Each house which contains empty cells has an element in `cell_cand_pos` of the form:

$$House: \ \{Q_1, \ldots, Q_E\}$$

where,

$$Q_i = Cell_{rc_i}: \ [C_1, \ldots, C_{m_i}]$$

E is the number of empty cells in that house and the details of $House$ and $Cell$ are provided above.

4 Noteworthy Code

Finding n

To allow the program to keep its functioning consistent with Sudokus of different sizes, some functions use n as a variable. At the beginning of those that do, there is one of these simple lines of code to define n.

```
n = int(sqrt(sqrt(len(cell_dict))))
```

or,

```
n = int(sqrt(len(list_of_lists)))
```

The first works using the fact that an n^2 by n^2 Sudoku has n^4 cells. Taking a double square root yields n. Similarly the second uses the fact that there are n^2 lists in `list_of_lists`.

Converting Between Cell And Box Indices

For a specific cell (r, c), what box is it in? Or, for a given box (i, j), what are the indices of its n^2 cells?

For some processes in the program it is important to obtain the answer to one or both of those questions. For example, to fill `ref_dict` with the places values in a given box, it is important to know which cells are in that box. This is done by determining the ranges of indices for both rows and columns that overlap the box. Another case is when an empty cell is to be added to a box element in `cell_cand_rcb`, The box that the cell lies in needs to be determined.

In each case, the problem can be reduced by looking at a single line of n^2 cells divided into n 'sub-lines' each of length n. With this the questions above simplify towards; determining the sub-line of a cell from its position, or finding the range of cells in any given sub-line. When it is known how to deal with a line, the methods can be adapted into two-dimensions for a Sudoku.

Cell Index → Box Index

For a line of n^2 cells split into n ordered sub-lines. Provided the position (X) of one of the cells, the statement below will determine which sub line the cell lies in.

$$\left\lfloor \frac{(X+n-1)}{n} \right\rfloor$$

It is possible to apply this concept to the two dimensional case of a Sudoku by taking two of these sub-lines with equal lengths. One can be called a row and have cell position R; the other may be called a column with cell position C. For this case the box index for a cell (R, C) will be a combination of the relevant 'sub-row' and 'sub-column' for that cell. With this we have the box index to be:

$$\left(\left\lfloor \frac{(R+n-1)}{n} \right\rfloor, \left\lfloor \frac{(C+n-1)}{n} \right\rfloor \right)$$

Finally, the code to find this box index is as follows:

```
box_index = ((row + n-1)//n, (col + n-1)//n)
```

Box Index → Cell Index

Considering an ordered line of cells as before, we wish to obtain a range of cells which are in a given sub-line. To do this, an upper and lower limit of the range must be found. For a sub-line in position Y along the line, the lowest cell index in the group in the group is:

$$(Y-1) \times n + 1 \tag{1}$$

and the highest cell index is:

$$Y \times n \tag{2}$$

So the range of cell positions for a given sub-line is $[(1), (2)]$, which is given by `range((1), (2)+1)` in Python. To find all of the cells indices in a box, this range must be considered for the sub-row and sub-column of that box. For example, to accomplish that this double `for` loop can be used:

```
for cell_row in range( (i-1)*n+1 , i*n+1 ):
        for cell_column in range( (j-1)*n+1 , j*n+1 ):
```

where (i, j) is the box index.

`empty_cells`

The solving methods stated in §2 don't require information on every cell in the Sudoku. Specifically they only need to consider cells which are empty. For this reason the program keeps track of those cells in a list called, `empty_cells`. The list is filled with this code:

```
empty_cells = []
for cell in cell_dict:
    if type(cell_dict[cell]) == list:
        empty_cells.append(cell)
```

Fill `cell_cand_rcb`

The dictionary, `cell_cand_rcb` (described in detail in §3.1) is filled by this sequence of code:

```
for cell in empty_cells:
    row = cell[0]
    col = cell[1]
    box = ( row +n-1)//n, ( col +n-1)//n
    houses = {'row': row, 'column': col, 'box': box}
    for house in houses:
        if not((house, houses[house]) in list(cell_cand_rcb.keys())):
            cell_cand_rcb[(house, houses[house])] = {}
        cell_cand_rcb[(house, houses[house])][cell] = cell_dict[cell]
```

For each empty cell in the Sudoku, this code adds an element for that cell into all three 'sub-dictionaries' in `cell_cand_rcb` that correspond to the cell's three houses. In those three cell elements, the candidate list for that cell is attached.

5 Input And Output Functions

5.1 `input_square`

Given a list of lists which represent a Sudoku puzzle, `input_square` converts the problem into dictionaries. It produces `cell_dict` and `ref_dict`. However the `cell_dict` that is produced is not complete, for each empty cell it holds a non reduced list ($[1, \ldots, n^2]$). This list does not yet represent the possible candidates for the empty cell. After filling the dictionaries, the function checks that there are no errors in the input this will be discussed further.

To fill the dictionaries `input_square` first adds an empty list to `ref_dict` for every house. The rows and columns are done first in a single `for` loop. The function then cycles with a quadruple `for` loop, filling `ref_dict` with empty lists for each boxes with the first two loops. With the second two loops, every cell in each box is addressed. `if cell_val == 0` checks whether that cell is empty, if it is `cell_dict` is given a list. If the cell is not empty, `cell_dict` and `ref_dict` are given the cell's value.

Pseudocode

Input: (Sudoku as a list of lists)

Fill `cell_dict` and `ref_dict`.

for house in the Sudoku**:**
- Make a `ref_dict` element containing an empty list.

for cell in the grid**:**

if that cell is empty **then**
- Make a `cell_dict` element for the cell containing the list: $[1, \ldots, n^2]$.
- Add that value to the lists in `ref_dict` corresponding to each of the 3 houses that the cell occupies.

- Look for Input errors.

Output: (`cell_dict, ref_dict`)

When inputting the Sudoku problem there could be mistakes made. This can cause problems processing the puzzle, if there are false entries in the input some of the functions will not work properly. For this reason, it is important that there are mechanisms in place to deal with and make the user aware of these mistakes.

5.1.1 Catching Input Errors

All input errors are picked up by the `input_square` function. Possible invalid entries include:

Wrong size
This error occurs when the side lengths of the grid are not square numbers. Caught by:

```
if not(sqrt(len(list_of_lists)) % 1 == 0):
    return('False entry. Your puzzle length is not a square number.')
```

Wrong shape
This error occurs when the grid entered is not square in shape. Caught by:

```
for row in list_of_lists:
    if len(row) != len(list_of_lists):
        return('False entry. Your puzzle is not square.')
```

Inappropriate value
Since the only numbers in a Sudoku are elements of the set $\{1 \leq x \leq n^2 \,|\, x \in \mathbb{Z}\}$ an inappropriate value to enter in a cell would be one that is not in this set. To check for this type of error the program needs to look for values that are; too big, too small or not an integer.

Repeated values
The rules of Sudoku state that each house cannot contain more than one of any value. So a false entry would occur when the user inputs more than one of a number into any row column or box. Inappropriate value and repeated values are both caught by this bit of code.

```
for house in ref_dict.keys():
    for value in ref_dict[house]:
        if value != 0:
            if value > n**2:
                return('Value: %s is too big.'% str(value))
            if value < 0:
                return('Value: %s is too small.' % str(value))
            if not(type(value) == int):
                return('Value: %s is not an integer.'% str(value))
            if ref_dict[house].count(value) > 1:
                return('%s is repeated' %str(value))
```

5.2 print_puzzle

Provided the `cell_dict` for a given Sudoku, `print_puzzle` will print the grid to the Python console in a form recognizable as a Sudoku.

This is accomplished by first, converting the dictionary back into a list of lists with each interior list corresponding to a row. Then, the function adds vertical and horizontal lines to divide the boxes and contain the grid. Finally, `print_puzzle` sequentially prints out each list in the main list representing a row or a boundary.

`print_puzzle` will scale its output appropriately depending on the size of the Sudoku, and how long the entries in the cells are. Say for example, a Sudoku is sized $n = 4$ (a 16 by 16 grid). Some of the filled values could be 2 digits long. In this case, for all the cells filled by a single digit number a space is inserted in front of the number. This ensures that all the rows in the grid line up correctly. More generally for a Sudoku, if the longest entry in the grid is X characters long, for a given entry in a cell that is Y characters long, $X - Y$ spaces will be inserted in front of that entry.

When given an incomplete Sudoku grid, `print_puzzle` will by default represent unfilled cells as white space in the output. However, the option is available to change this by setting the variable, `hide_candidates` at the beginning of the function to be false. Doing so will change the output for an empty cell from white space to the candidate list for that cell. As with numbers of more than one digit the function will scale the output to accommodate lists aesthetically.

Pseudocode

Input: `(cell_dict)`

- Find the maximum character length to scale up to.

Fill a main list with lists corresponding to rows and horizontal boundary's.

for row in the Sudoku**:**

- Make a list for that row.

for cell in the row**:**

- Scale that cell's entry to the appropriate size (considering the maximum character length).
- Add that entry to the row list.

- Add vertical lines in appropriate places along the row.

- Define appropriate looking horizontal boundaries with; '+'s at the intersections between boundaries, and vertical lines at the far right and left.
- Define a plain horizontal line with the right length to bound the top and bottom of the print out.
- Add the horizontal boundaries at relevant intervals between rows in the main list.

Print to the console.

- Print the top boundary.

for row list in the main list **:**

- Print the elements of that list (using `' '.join(list)`).

- Print the bottom boundary.

Output: Printed Sudoku

To insert the horizontal boundaries between boxes the function uses this `for` loop:

```
y=0
for i in range(1,n):
    main_list.insert(i*n+y, horizontal_boundary)
    y+=1
```

It works by using `insert()` to add the boundaries at specific indexes in the main list. The specific indexes that require boundaries can be initially determined by $i \times n$ for $i = 1, \ldots, n$. The problem is, when a boundary is added to the main list the required index of the next boundary (if there is one) will change by $+1$. To amend this issue the iterative variable `y` is used and the corrected index becomes $i \times n + y$.

6 Logical Solving Functions

6.1 Value Allocation Functions

Functions that apply numbers to cells using logical methods stated in §2.2. Each of these functions output a list of cells that they have filled. This list is important as it is used to apply the mark up method to appropriate empty cells after values have been placed. Both of these functions act specifically on the empty cells in the unsolved Sudoku.

6.1.1 naked_single

Individually looks at the candidates of each empty cell in the unsolved Sudoku. If the cell has only one candidate, the function allocates that value to the cell.

To do this `naked_single` uses a `for` loop to cycle through `empty_cells`. For each of those cells it checks `if len(cell_dict[cell]) == 1` (whether the length of the candidate list is 1). If this is the case for a cell, the function redefines the `cell_dict` element for that cell as the one candidate left in

the list. Also, `ref_dict` is updated accordingly by adding the number to lists in the row, column, and box elements in `ref_dict` that correspond to the row, column, and box which the cell is in.

Pseudocode

Input: `(cell_dict, ref_dict, empty_cells)`

Look for naked singles and allocate them to their cells.

for cell in `empty_cells`**:**
if the candidate list for that cell (held in `cell_dict`) is of length 1 **then**
- Confirm the lone value in that list as the number for that cell in `cell_dict`.
- Update the `ref_dict` appropriately.
- Add the cell to the list of `filled` cells.

Output: `(cell_dict, ref_dict, filled)`

6.1.2 hidden_single

Considering only houses that have empty cells. The function looks for Hidden Single candidates in the empty cells of those relevant houses. When it finds one, `hidden_single` fills the resident cell with its definite value.

Conveniently, `cell_cand_rcb` contains information on all the houses that have empty cells. The function works by considering the appropriate houses in `cell_cand_rcb` one by one. For each of those houses `hidden_single` makes a list, `house_candidates`. This is filled with all candidates for empty cells in that house. Repeated candidates are included. So for example, if two empty cells in a house both have 5 as a candidate, `house_candidates` will be filled with two 5s. This is vital as the function is seeking candidates that only appear once in the house. With `house_candidates` filled, `hidden_single` considers every candidate of each empty cell in the house. For the candidate it checks: `house_candidates.count(candidate) == 1`. If this is true, the candidate is a Hidden Single and is applied to its cell.

Pseudocode

Input: `(cell_dict, ref_dict, cell_cand_rcb)`

for houses containing empty cells **:**
- Fill `house_candidates`.

for each empty cell in the house**:**
if one of that cell's candidates appears only once in the house. **then**
- That candidate is a Hidden Single.
- Fill the cell with that value.
- Add the cell to `filled`.

Output: `(cell_dict, ref_dict, filled)`

There is one problem with the method used by this function. It arises from the fact that each cell occupies three houses. Since `hidden_single` works with `cell_cand_rcb` and this dictionary is not updated when a cell is filled. After a Hidden Single is found for a cell in one house, the same cell will be considered as if it were not filled in its other two houses. This is bad for a couple of reasons. Firstly, it will cause the function to cycle over objects which are irrelevant to it, which wastes time. Also, an empty cell may have a resident candidate which satisfies the condition to be a Hidden Single in more than one house. In this case the function will try to fill a cell multiple times. It will also add that cell to the `filled` list more than once, causing issues in other functions. To fix this problem and stop

`hidden_single` completing unnecessary computations the function has a simple check. In the `for` loop which cycles over cells to look for Hidden Singles, before doing anything else the function completes this `If` check:

```
if cell in filled:
    continue
```

Which amends the issue.

6.2 Candidate Elimination Functions

6.2.1 markup_specific

Applies the mark up method to cells that are in a provided list.

For each of those cells in the '`cells_to_markup`' list. This involves removing values from its candidate list if that value is already filled somewhere else in one of the three houses of which that cell is a resident

`markup_specific` does this for a cell by first determining what houses the cell occupies. Then, using `ref_dict`, it finds the values that are already placed in those three houses. Finally, the function checks if any of those values are in the candidate list for the cell, if one is it is removed.

There is a good reason for why the function is written to only mark up specific cells as opposed to all of them. In several cases not many cells need to be marked up. The highest number of cells that will ever be required to be marked up for a uniquely solvable 9 by 9 Sudoku is 64 (81 total cells minus the minimum number of initial clues, 17). And that will only need to happen once when solving the puzzle. Many of the instances where the markup method is required is when a cell or cells have been filled. If for example, one cell is filled (in a 9 by 9 grid), the most cells that will be affected and hence need to be marked up in the case is 20 (if all the other cells in the three common houses are empty). However it will often be less. In these cases marking up all the cells would waste a substantial amount of computation time over many iterations.

Pseudocode

Input: `(cell_dict, ref_dict, cells_to_markup)`

for cell in the `cells_to_markup` list provided**:**
Figure out which values cannot be in that cell
(based on if they are assigned somewhere else in one of the cell's three houses).

for each of those values that cannot fill the cell**:**
if that value is in the cell's candidate list **then**
- Remove that candidate from the candidate list for that cell in `cell_dict`.

Output: `(cell_dict)`

6.2.2 markup_relevant

When cells are filled, the remaining empty cells in the same houses need to be marked up. This function, takes a list of cells that have been `filled` and marks up the empty cells that need to be.

This is done by, finding the appropriate empty cells and making a list of them, then using `markup_specific` on that list. Most of the function is dedicated to finding the right cells. To do this the function uses a dictionary called `relevant_houses`. This dictionary is constructed by the function and contains information on which houses are occupied by the cells that have been filled. Once those houses are known, `markup_relevant` makes a list of the empty cells that are in at least one of the relevant houses.

Pseudocode

Input: `(cell_dict, ref_dict, filled, empty_cells)`

Figure out witch empty cells share a house with a newly filled cell.

Fill the `relevant_houses` dictionary.

for each cell that has been filled**:**
for each of that cells three houses**:**
if that house is not in `relevant_houses` **then**
- Add it to the dictionary.

Use `relevant_houses` to determine which cells need marking up.

for every empty cell in the Sudoku**:**
if one of its three spanned houses is in `relevant_houses` **then**
- Add that cell to a list of cells that are to be marked up (`to_markup`).

Markup all of the relevant cells

- Apply `markup_specific(cell_dict, ref_dict, to_markup)`

Output: `(cell_dict)`

6.2.3 box_line_intersection

Seeks all relevant intersections between houses where the conditions to use Box-Line Intersection method are present. As discussed earlier these intersections occur between either, a row and box or, a column and box. If one is found `box_line_intersection` will remove candidates appropriately.

On a regular 9 by 9 Sudoku there are many intersections between lines and boxes (line again meaning row or column). Each box has 3 rows and 3 columns overlapping it. That makes 6 intersections per box, so $9 \times 6 = 54$ total. In that case, to find all the condition satisfying intersections `box_line_intersection` would need to consider all 54 of these. However, some intersections can be ignored. Firstly, if either the box, or the line of the intersection are completely full, the necessary conditions cannot be satisfied. Second (†), in either the box, or the line. If there are no empty cells which are simultaneously inside that house, and outside the overlapping region. Then, again, the required conditions cannot be satisfied. `box_line_intersection` uses these two ideas to ignore pointless intersections in the interest of efficiency.

The function works by using a dictionary called `intersection_dict`. This is first filled with information on intersections in the Sudoku. As mentioned, intersections are only considered if both of their houses are not full. To determine which houses are not full `cell_cand_rcb` is used. Once filled, some cases are trimmed from the dictionary using the second argument above (†). Finally, each potentially useful intersection is examined and tested for *Type 1* and *Type 2* Box-Line intersection conditions (details in §2.3).

Specifications of intersection_dict

For each intersection in `intersection_dict` information is contained on which cells are empty in the two intersecting houses. More Specifically, which cells are empty in three particular zones within the span of those houses. These zones are as follows. One is the overlapping region of the line and the box. Another is the area which is within the box but outside the overlap. The last is the part of the row that is also outside the overlap. In the dictionary, for each intersection, these zones have their own lists. This is the form of `intersection_dict`:

$$\{Box_{ij}: \ \{Row_r: \ (Zone\ Lists),\ Column_c: \ (Zone\ Lists)\}\}$$

Where,

$$(Zone\ Lists) = \{'in': \ [\ ],\ 'out': \ \{'sameline': \ [\ ],\ 'samebox': \ [\ ]\}\}$$

Here like in `ref_dict`, Box_{ij}, $Column_c$, and Row_r have the exact form of: $('box',\ (i,j))$, $('row',\ r)$ and $('column',\ c)$ respectively. Each intersection in the dictionary is saved as its row or a column which is nested in the element corresponding to the intersecting box. For each of the three zones in the intersection there is a list in $(Zone\ Lists)$ that is to contain indices of the empty cells in that

zone. The element *'in'* : [ ] is the overlapping region. The other element *'out'* : {...}is for two zones outside the overlap. It has interior elements; *'sameline'* : [ ], and *'samebox'* : [ ] which have lists to hold appropriate empty cells.

Pseudocode

Input: `(cell_dict, cell_cand_rcb)`

Fill `intersection_dict`.

- Fill the dictionary with boxes that contain empty cells.

for each line that that is not full.**:**

for each of the boxes that the line shares empty cells with**:**

- Add that intersection to the dictionary.
- For the empty cells which are in the overlapping region of the intersection. Add them to the `'in'` part of the dictionary for that intersection.

for all of the intersections in the dictionary**:**

if it is a row-box intersection **then**

- Fill 'sameline'
- Fill 'samebox'

if it is a column-box intersection **then**

- Fill 'sameline'
- Fill 'samebox'

if by argument (†), the intersection is useless **then**

- Add it to a list to be ignored.

- Delete all the useless intersections from `intersection_dict`.

Inspecting `intersection_dict`**, attempt to eliminate candidates.**

for every intersection left in `intersection_dict`**:**

for each present candidate in the overlapping region**:**

- Considering whether that candidate is a possibility for the empty cells in the 'sameline' or 'samebox' lists of the the intersection.

if conditions for the *Type 1* case are satisfied **then**

- Eliminate the candidate from cells in `'samebox'`

if conditions for the *Type 2* case are satisfied **then**

- Eliminate the candidate from cells in `'sameline'`

Output: `(cell_dict)`

To find cells to fill the `'sameline'` list for an intersection. The function uses the 'box range' mentioned in §4 to find the boundaries of the box, and then it seeks empty cells outside those boundaries.

6.2.4 naked_subset

This function seeks Naked Subsets in individual houses and when one is found, it eliminates candidates where possible. To do this it only considers houses that contain empty cells.

As discussed in §2.3, for a Naked Subset to be helpful it must have a quantity of numbers, X which is within the range: $2 \leq X \leq (E-1)$, where E is the number of empty cells in the house and $X, E \in \mathbb{Z}$. It follows from this that if an house contains less than three empty cells ($E < 3$) there can be no useful Naked Subsets in that house. To maximize efficiency, the function specifically looks for Naked Subsets that have a length in that range, and it only seeks them in houses with three or more empty cells.

To find what it is looking for, `naked_subset` uses a dictionary called `subsets`. For each empty cell in every relevant house, it makes an element in `subsets` of this form:

$$Subset: \ Spanned\ Cells$$

where,

$$Subset = [C_1, \ldots, C_X]$$

and

$$Spanned\ Cells = [Cell_1, \ldots, Cell_m]$$

The key of that element (*Subset*) is the candidate list for the specific empty cell, this is the potential Naked Subset. The value of the element (*Spanned cells*) is a list containing indices of some other empty cells in the house. For a cell to be in this list its candidate list must contain no number which is not in *Subset*. When this single dictionary element is filled, the function inspects it to see if the list of candidates forms a valid Naked Subset, if it does, candidates are removed from other empty cells where possible. The list will form a valid Naked Subset if and only if the number of candidates in *Subset* is exactly equivalent to the number of empty cells in *Spanned Cells*.

Pseudocode

Input: `(cell_dict, cell_cand_rcb)`

for every house that contains empty cells**:**
if the house contains less than 3 empty cells **then**
- Skip that house.

for each empty cell in the house**:**
if that cell has exactly E candidates **then**
- Skip that cell.

Fill an element of `subsets` for the empty cell.

- Add the candidate list for the empty cell as the key.

for each empty cell in the house**:**
if that cell has no candidates which don't appear in *Subset* **then**
- Add that cell to *SpannedCells* in the dictionary element.

Inspect that dictionary element to see if holds a Naked Subset.

X = (number of candidates in *Subset*)
S = (number of cells in *SpannedCells*)
if $Y < X$ **then**
- The subset is not a Naked Subset.

if $Y = X$ **then**
- The subset is a Naked Subset.
- Remove candidate in the subset from empty cells not in the cell span of the subset.

Output: cell_dict

When solving a Sudoku, it is not uncommon to see multiple empty cells in a house which all have exactly the same candidates (a Naked Pair is an explicit example of this). With the process `naked_single` uses, it will decide whether a set of candidates is a Naked Subset the first time it considers that set. For this reason, once the function tests a set of candidates for a cell, there is no reason for it to test the same candidates for another cell in the same house, so it doesn't. To realize this, a simple check is employed for each cell that is considered in a given house, it is as follows:

```
elif candidate_group in ignore:
    continue
```

`ignore` is a list that is filled with candidate sets that have already been considered in the specific house.

6.2.5 hidden_subset

This function looks for Hidden Subsets in houses that contain empty cells, on finding one it will eliminate appropriate candidates where possible.

From §2.3 we know that for a Hidden Subset to be of any use, it must be X candidates long, where $2 \leq X \leq (E-2)$. Since the length must be in that range, if a house has less than four empty cells ($E < 4$), there is no length X that satisfies the range, and so that house cannot contain a useful Hidden Subset. With this in mind `hidden_subset` ignores houses with less than four empty cells.

The function seeks two types of Hidden Subset using different methods for each. Here, the two types will be referred to as: Basic Hidden Subsets, and Complex Hidden Subsets. These variations are described in detail below:

Basic Hidden Subsets (BHS)
A BHS is a Hidden Subset where the candidate lists of each empty cell that is spanned by the Subset contains every candidate in the Hidden Subset. A Hidden Pair is explicitly a Basic Hidden Subset.

Complex Hidden Subsets (CHS)
To be a CHS a Hidden Subset simply needs to not satisfy the condition to be a BHS. That is to say, one or more of the candidate lists of empty cell spanned by the Subset must contain not all of the candidates in the Hidden Subset. The example in §2.3 where the Subset is $[1, 2, 3]$ is an example of a Complex Hidden Subset.

The function seeks these two variations independently because there is a nice efficient way to find Basic Subsets that is not capable of finding Complex ones.

To complete its task, `hidden_subset` fills and then consults a dictionary called `cand_pos_rcb`. This dictionary contains information on houses in the Sudoku, specifically, houses that contain empty cells. For each of these houses `cand_pos_rcb` has two sets of information; it knows which cells are empty and what the candidates for those cells are (just like `cell_cand_rcb`), it also holds information on every candidate that appears in the house, and for each candidate it has a list of cells that have the candidate as a possibility. The exact form of an element corresponding to a house in `cand_pos_rcb` is as follows:

$$House: \ \{'cells': \ \{Q_1, \ldots, Q_E\}, \ 'cand\_pos': \ \{P_1, \ldots, P_H\}\}$$

where,

$$Q_i = Cell_i: \ [C_1, \ldots, C_{m_i}]$$

and

$$P_i = C_i: \ [Cell_1, \ldots, Cell_{s_i}]$$

C_i represent candidates for empty cells, H is the number of unique candidates for empty cells in the house, and s_i is the number of cells spanned by candidate C_i.

After filling `cand_pos_rcb`, the function tests every relevant house in the Sudoku with the aim of finding Basic Hidden Subsets. If none are found `hidden_subset` shifts its attention to look for Complex Hidden Subsets. It seeks BHSs before CHSs because, as mentioned above, the method of finding BHSs is more efficient.

Finding Basic Hidden Subsets

To seek Basic Hidden Subsets, the function inspects the lists of cells that are spanned by candidates held in

the $'cand\_pos'$ part of the dictionary. It uses the fact that: if X candidates all span exactly the same cells and they all span X cells then those candidates form a BHS across the spanned cells.

The Function uses this by considering each candidates cell span one by one. For each, `hidden_subset` looks in the *'cand_pos'* part of the dictionary to count how many candidates have the exact same cell span as the one being considered. If that count results in a number equivalent to the length of the cell span, that group of cells must contain a Basic Hidden Subset. Once this is known, the function determines which candidates make up that Subset.

Finding Complex Hidden Subsets

Searching for Complex Hidden Subsets is a notably less trivial task than seeking their Basic counterparts. This is because the candidates in a CHS can be in various arrangements across the candidate lists of the spanned cells, it is not possible to seek a single pattern. Overcoming this challenge, the function uses a less elegant approach than that used to look for Basic Hidden Subsets. To seek Complex Hidden Subsets, the function finds all the appropriate combinations of the unique candidates that are in `house_candidates`. Once found, the function individually checks each of these candidate combinations to determine whether it is a CHS. To ensure that as little time as possible is wasted, only specific combinations are generated. Hidden Subsets are limited to be X candidates long where $2 \leq X \leq (E-2)$. But remembering that if a Hidden Subset is of length 2, it is automatically Basic, only combinations in the length range $3 \leq X \leq (E-2)$ are generated. As well as this, there are no two combinations produced that have the same digits in a different order. This is the specific code that is responsible for generating the combinations:

```
hc = house_candidates
if len(hc) < 5:
    continue
hc.sort()
combos = {3:[]}
all_combos = []
for i in range(len(hc)):
    for j in list(range(i+1,len(hc))):
        for k in list(range(j+1,len(hc))):
            combos[3].append([hc[i],hc[j],hc[k]])
            all_combos.append([hc[i],hc[j],hc[k]])

for length in range(4,len(hc) - 1):
    combos[length] = []
    for combo in combos[length - 1]:
        biggest = combo[-1]
        for i in range(len(combo), len(hc)):
            if hc[i] > biggest:
                combos[length].append(combo + [hc[i]])
                all_combos.append(combo + [hc[i]])
```

It fills the list `all_combos` by initially finding all the combinations of length three, then it finds the longer lengths by adding single numbers to the end of previously found lists. Knowing that the range of CHS lengths is $3 \leq X \leq (E-2)$, candidate sets of length four or less can be ignored, this is the reason for the check `if len(hc) < 5`.

Once all the combinations are found, the function tests them one by one to confirm or deny if each is a valid Hidden Subset. To do so, it first finds the group of cells that spanned by the candidate set then, the function compares the number of spanned cell to the length of the candidate set, if they are the same, the set is a Hidden Subset. This method will actually find both Complex and Basic Subsets however, at least for its implementation in this function, it only considers candidate sets with three or more digits, and so it will not find Hidden Pairs.

Pseudocode

Input: `(cell_dict, ref_dict, cell_cand_rcb)`

1. Fill `cand_pos_rcb`.

for every house which contains empty cells**:**
if if contains less than four empty cells **then**
- Ignore that house

- Make an element for that house in `cand_pos_rcb`.
- Fill the `'cells'` part of that element using `cell_cand_rcb`.

for each of those houses in with an element in `cand_pos_rcb`**:**
Fill the *'cand_pos'* part.
- Fill `house_candidates` with all the candidates appearing in the house.
for each of those candidates in `house_candidates`**:**
- Make a list in the *'cand_pos'* part of the dictionary element, fill it with all the cells in the house that have the candidate as a possibility.

2. Seek Basic Hidden Subsets.
for every candidate in `house_candidates` for the house**:**
- Let `cell_set` be the list of cells which that candidate is a possibility for.
X = The number of times that exact cell set appears in the `'cand_pos'` part of the dictionary element for that house.
Y = The number of cells in the cell set
if $X = Y$ **then**
- That cell set contains a Hidden Subset.
Find the candidates which are in the Hidden Subset.
for each candidate in `house_candidates`**:**
if it has the same cell set as the one spanned by the Hidden subset **then**
- That candidate is in the Hidden subset

- Remove candidates that are not in the Hidden Subset from the cells that are spanned by it.

if a basic Hidden Subset was found **then**
- Stop the function and output cell_dict.

3. Seek Complex Hidden Subsets.
- Find all suitable combinations of the candidates which are in `house_candidates`
for each of those sets of candidates**:**
for each candidate in the set**:**
if that candidate appears in more cells than the length of the candidate set **then**
- That candidate set cannot form a Hidden Subset.
- Skip to the next combination of candidates.

- Find all the cells that are spanned by this candidate set.
X = The number of cells that are spanned by the set.
Y = The number of candidates in the set.
if $X = Y$ **then**
- The candidate set is a Hidden Subset.
- Remove appropriate candidates where possible.

Output: `(cell_dict)`

7 Systematically Applying Logical Methods

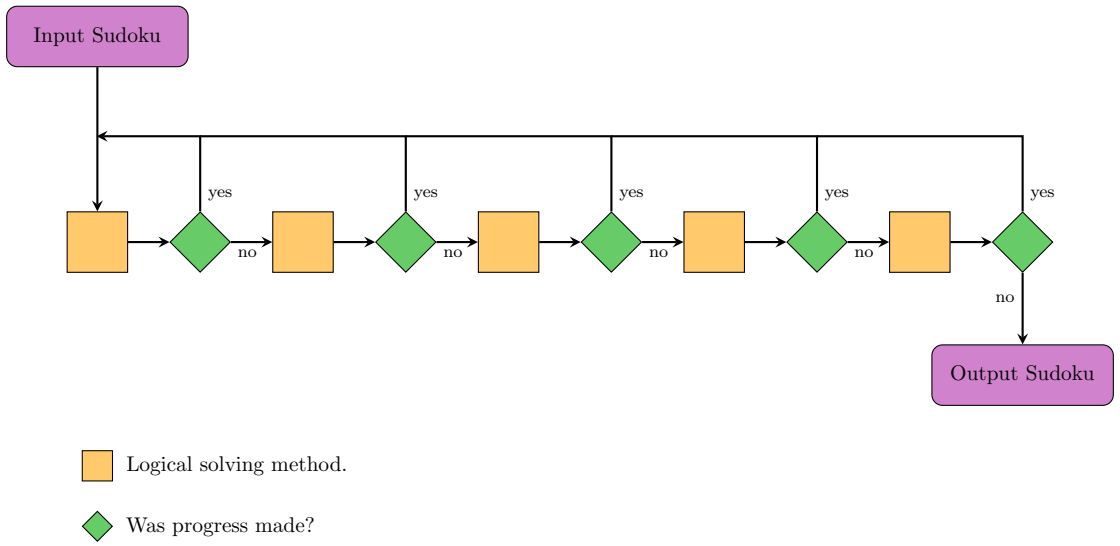
7.1 Algorithmic Process

We now have five logic based solving functions to use on Sudokus, but what is the best order to apply them? The goal is to solve puzzles as fast as possible, so the best process of application will be the one which works fastest.

With the task of quickly progressing a sudoku towards its solution, some of the logical solving functions will be more effective than others. Conscious of this, it is clear that the best process will use the more efficient functions before the slower ones. For that purpose, the functions are ordered in a chain starting with the fastest method, ending with the slowest. The choosing of the order in that chain will be discussed in the next subsection, for now, the examination will focus on finding the best algorithmic process to apply methods in the chain.

It is not likely controversial to suggest that a good algorithm would simply apply each method in the chain sequentially from the first to the last, then repeat until no further progress can be made. This would would solve Sudokus just fine, but a better way exists. There is no point in administering complex solving methods when simpler ones can make progress. Knowing this, an inefficiency can be sighted in the algorithm stated above, and that is; after a simple method is used to allocate values or eliminate candidates, it is not impossible or even uncommon that the same simple method can be implemented to make progress again, without changing the puzzle further.

In the interest of not unnecessarily sending the puzzle through slow complicated solving functions a better process is formulated: Starting with the first, apply a method in the chain, if it doesn't make progress, apply the next method, if it does, return to the start of the chain, if you use the last method in the chain and it cannot make progress, output the result. Here is a visual representation of this algorithmic process:



This is the process which is utilized by the program.

7.2 Optimal Order Of The Chain

In the previous section, the utilized process of administering logical solving methods in a chain was defined. To employ this process it is necessary to decide on the order of that chain. As mentioned previously, the desired order is that which favours the efficiency of the methods, starting with the fastest.

Conscious of this, the next step is to figure out how efficient the five solving functions are in comparison to each other. This task is not trivial because when it comes to progressing a Sudoku towards its solution, the efficiency of a solving function will depend on these two things; the speed at which that function does its job, and, how much progress that job will make. The latter becomes a complicated question when candidate elimination functions are considered.

Because of the complexity this choice of order, I opted to make it via experimental methods. Instead of going through the complex analysis of each function's effectiveness, I simply tested several different orders to see which was fastest. At first this may seem like a daunting exercise; with five methods there is $5! = 120$ ways to arrange them, with each test taking say 2 minutes, it would be a bit of a grind. Thankfully, a couple of assumptions can be made to simplify the task.

Firstly, with some contemplation it is clear that the value allocation functions; `naked_single`, and `hidden_single` will be the most efficient methods. They are comparatively much more basic in structure that the others, also, filling cells will progress a Sudoku faster than eliminating candidates.

Secondly, `naked_single` must sit before `hidden_single` in the chain. This is strongly arguable for two reasons; inspection of the functions shows that `naked_single` will require less time to run, it contains a single `for` loop where as `hidden_single` uses five, as well as this, `hidden_single` uses `cell_cand_rcb` which needs to be filled in each solving loop before it is used, which requires time (the solving loop will be clarified in a later section).

With these assumptions it is robust to assume that the first method in the optimal chain is naked Single and the second is Hidden Single. This now leaves only three methods to be ordered, thus reducing the number of tests from 120 to a more manageable $3! = 6$.

To test each of the 6 orders, I ran 1011 Sudokus through the algorithmic process. As expected, each chain arrangement solved the same number of Sudokus, 319. For each test, the average time to solve a sudoku was recorded. The same order will give a slightly different average solve time each time it's run, to counter this I completed each of the 6 tests three times and took the mean average. Here are the results ordered with time.

Order	Time (s)
n-sin, h-sin, b-lin, n-sub, h-sub	0.006289
n-sin, h-sin, n-sub, b-lin, h-sub	0.007428
n-sin, h-sin, n-sub, h-sub, b-lin	0.008121
n-sin, h-sin, b-lin, h-sub, n-sub	0.008694
n-sin, h-sin, h-sub, b-lin, n-sub	0.010683
n-sin, h-sin, h-sub, n-sub, b-lin	0.010701

From this, the best chain order is clear; Naked Single, Hidden Single, Box-line Intersection, Naked Subset, Hidden Subset. Looking at the above table, it is visible that the further Hidden Subset is to the back of the chain, the faster the time.

Order	Time (s)
n-sin, h-sin, b-lin, n-sub, h-sub	0.006289
n-sin, h-sin, n-sub, b-lin, h-sub	0.007428
n-sin, h-sin, n-sub, h-sub, b-lin	0.008121
n-sin, h-sin, b-lin, h-sub, n-sub	0.008694
n-sin, h-sin, h-sub, b-lin, n-sub	0.010683
n-sin, h-sin, h-sub, n-sub, b-lin	0.010701

This is understandable as; the closer a method is to the front of the chain, the more it is used, and Hidden Subset employs the slow approach of testing combinations. After noticing this pattern in the data, I ran the fastest order again but without Hidden Subset, 3 less Sudokus were solved and the times remained about the same. But then I tried removing Hidden Subset from the trial and error methods, and this induced significant time improvements. Specifically it reduced the solving time of a 16 by 16 Sudoku from 34 seconds to 9. The effect will be enhanced for 16 by 16 puzzles as the combination method of Hidden Subset will potentially have a lot more cases to consider. However, less drastic but still evident time reductions were present for 9 by 9 Sudokus, but of course, only those which required Trial And Error methods to solve.

7.3 Solving Loop

The 'Solving Loop' is a bit of code that is used to implement solving functions in a procedure which is analogous to the algorithmic process outlined above.

The program uses it in two of its functions, in each case the details of the loop vary slightly but the underlying structure is identical. The loop works by running each method in the chain one by one, after each is applied, the question is imposed; was progress made? when that answer is yes, the loop is started again at the first method. To impose such a question it is required to know the state of the Sudoku's `cell_dict`; before, and after a solving function is applied. To enable this, a copy of `cell_dict` needs to be made at the start of each loop. This may seem as simple as saving it as another variable, unfortunately, doing this will not work, the dictionary and the variable will become intrinsically linked.

One valid option for saving a copy is to use `deepcopy()`, imported from the copy module. This would work fine but it is not what the loop uses. Speed testing proved another solution to be faster. This was to, instead of saving the whole dictionary, just save its values in a string format. This data type is not linked to `cell_dict`, and so serves as a useful copy.

When the value allocation methods are applied and cells are filled, `markup_relevant` is applied to those filled cells immediately after to update the appropriate candidate lists. With the exception of `naked_single`, all logical solving function use `cell_cand_rcb`. For this reason, that dictionary is filled with appropriate contents in every loop of the Solving Loop, but this is done after `naked_single` has been applied. Doing this avoids the wasting of time when `naked_single` is all that is required.

Pseudocode

while progress can be made**:**

- Apply `naked_single`.
if it made progress **then**
  - Markup appropriate cells.
  `continue`

-Fill `cell_cand_rcb`.

- Apply `hidden_single`,
if it made progress **then**
  - Markup appropriate cells.
  `continue`

- Apply `box_line_intersection`.
if it made progress **then**
  `continue`

- Apply `naked_subset`.
if it made progress **then**
  `continue`

- Apply `hidden_subset`.
if it made progress **then**
  `continue`

---

8 Trial And Error Based Solving Functions

8.1 Functions To Facilitate Trial And Error Methods

8.1.1 check_for_errors

When testing a trial-value in a trial-cell, it is vital to determine whether the guess made is false. To enable this the program has a short function called `check_for_errors`. It uses two ideas, the first is that no house can contain more than one of the same number, the second is that if an empty cell has no possible candidates, something must be wrong. The function searches the whole of a given Sudoku for each of these two error signalling cases. This is the form of `check_for_errors`:

```
def check_for_errors(ref_dict, cell_dict):
    for house in ref_dict:
        for value in ref_dict[house]:
            if ref_dict[house].count(value) != 1:
                return True
    for cell in cell_dict:
        if cell_dict[cell] == []:
            return True
```

8.1.2 apply_logic

As mentioned, filling a trial-value into a cell can result in one of three outcomes. Given puzzle data for a Sudoku that has had a trial-value inserted, `apply_logic` will enlist logical solving methods to determine the outcome to that guess. The outcome will depend on what the logical methods accomplish. Methods are used in a solving loop almost identical to what is outlined in §7.3, although there are a couple of differences.

To determine if a provided guess results in an error, `check_for_errors` is ran at the beginning of each cycle of the solving loop, if this finds some contradiction, `apply_logic` will terminate and output `'error'`.

In the case that the solving loop finishes with no interruptions, the Sudoku will be examined to see if it is complete; this is done by checking if there are empty cells remaining or not. If this inspection shows that the Sudoku is solved, the function will stop and output `'solution'` along with the `cell_dict` of the completed puzzle. Alternatively, if the inspection finds that there are empty cells remaining, the function will output `'stuck'` along with the `cell_dict`, `ref_dict` and, `empty_cells` of the puzzle.

Through experimentation, it has been found that when `hidden_subset` is removed from the solving loop in `apply_logic`, `branching_tree` functions more quickly. For this reason, `hidden_subset` is removed. I also tried removing `naked_subset` from the loop, this curiously made 16 by 16 puzzles take less time to solve, but made 9 by 9 Sudokus take longer in general. Because it made solving 9 by 9s slower this change was not kept.

8.2 elimination_by_contradiction

This function implements a variant of proof by contradiction to eliminate candidates from empty cells. It sequentially tests potential values for empty cells, if a test leads to some contradiction, it can be deduced that the value in question can be eliminated from the candidate list of the empty cell that it was tried in.

When given an unsolved Sudoku, `elimination_by_contradiction` first fills a dictionary called `options`. This contains the same information that is held in `cell_dict` for empty cells. each element in `options` is for an empty cell and the value of that element is the candidate list for that cell.

Once filled, the function tries every candidate for each empty cell one by one. It does this to seek candidates that cannot fill their cells and so can be removed from their candidate lists. After each candidate is tried, that option is removed from `options`. If `options` becomes empty, the function stops and returns `cell_dict`.

Pseudocode

Input: `(cell_dict, ref_dict, empty_cells)`

- Make copies of the Sudoku data.
- Fill `options`

while there are untested candidates left in `options`**:**

- Reset the copied Sudoku data back to the original.
- Chose a trial-cell.
- Chose a trial-value.
- Remove this case from `options`.

if `options` is now empty **then**

- Break the `while` loop.

- Update the copied Sudoku data to contain the trial-value and cell by using `fill_a_cell`. Use `apply_logic` to see how the trial affects the Sudoku.

if the outcome of this test is an error **then**

- The trial-value cannot fill the trial-cell.
- Eliminate the trial-value from the candidate list of the trial-cell in the original `cell_dict`

Output: (`cell_dict`)

To try a value in a cell, the dictionaries representing the puzzle need to be changed, however it is important to be able to return to a point before the trial-value was placed. For this to be possible, a copy is made of the puzzle date and the trial is made in the copied data. To facilitate this for the dictionaries an imported function, `deep_copy` is used, this stops the copy dictionary being linked to the original.

8.3 branching_tree

Given an unsolved Sudoku, this function will seek and find every possible solution. It works by inspecting every 'branch' in the tree of possibilities for that puzzle. In this model empty cells will represent the forks in branches, they will also be referred to as nodes. The function follows each path one by one in a systematic order similar to that used when writing a statement in polish notation (as described in §4.3). If the given Sudoku only has one solution however, there is no point in looking down every branch to find another. Conscious of this, there is an option within `branching_tree` to output the first solution found, thus stopping the function doing unnecessary work and reducing the run time.

When exploring the tree of possibilities it is important to keep track of the current position within the tree. The function does this by assigning an identity to each node it has traveled, this is saved in the form of a list called `branch_ID`. Every single node in the tree has its one unique `branch_ID`, each one of these lists contains all of the nodes (cells) that have been passed (filled) to arrive at the current position in the tree, it also contains the node at the current position. As well as the position in the tree, two more pieces of information (listed below) are required to successfully traverse all of the branches.

First, when testing candidates at a node, it is necessary to know what the candidates of that empty cell are at that specific position in the tree, equivalent to saying: which new branches can be traversed.

Second, after testing branches, following them, maybe testing more branches, it is vital to not get lost. When the possibilities of a particular branch have been exhausted, the function needs to return the state of the Sudoku back to what it was at a previous node. To enable this, the puzzle data needs to be saved and stored for each of the traversed nodes.

To contain the potential candidates and the puzzle data for each node, the function uses a dictionary called `branch_cands`. This has an element filled for every node that is traveled. Each element of `branch_cands` is of the form:

$$Branch\_ID: \ \{'progress': \ (Puzzle \ \ Data), \ 'possibilities': \ [C_1, \ldots, C_{m_i}]\}$$

where,

$$(Puzzle \ \ Data) = (cell\_dict, \ \ ref\_dict, \ \ empty\_cells)$$

C_i represent candidates for the relevant empty cell as usual, and each element of $(Puzzle \ \ Data)$ is saved as a copy made with *deepcopy*().

Branching Tree uses a `while` loop to progress the exploration of possibilities. Each loop; a trial-value is chosen, then it is tested with `apply_logic`. Testing will result in one of three outcomes; an `error`, `'stuck'`, or a `'solution'`.

In the case where the outcome is that the Sudoku still has empty cells (`'stuck'`), the function will progress its attention to the next node. To do this the function seeks the first cell in the ordered list of nodes which has not already been filled. The form of this list defining what order nodes will be progressed to, is predetermined at the start of the function. Once this is chosen, the `while` loop skips to its next iteration.

In the other two cases where the outcome is either a `'solution'` or an `'error'`, the function will, if it is necessary, backtrack to the appropriate node and reset the puzzle date to what it was before any guesses were made at the node. When backtracking if a node has all its possibilities exhausted the previous node will be traveled to, this is repeated until one is found with candidates left to try. If the first node is returned to and it has no more possible values, the exploration has reached every leaf of the tree, every solution will be found and so, the function halts.

If the outcome is specifically a `'solution'`, before back tracking begins, that solution is saved to a list as its `cell_dict`.

Pseudocode

Input: `(cell_dict, ref_dict, empty_cells)`

- Decide on the order in which the empty cells (nodes) will be considered.
- Fill the first `branch_cands` element.

while there are more combinations of candidates to explore**:**

- Choose a trial-value for the current node.
- Apply that choice to the puzzle data.
- Using `apply_logic` test the choice for one of the three outcomes.

if the outcome is `'stuck'` **then**

- Find the next node in the node order that hasn't been filled by `apply_logic`.
- Fill `branch_cands` for this new node.

`continue`

if the outcome is not `'stuck'` **then**

if the outcome is not `'solution'` **then**

- Add that solution to the list.

if seeking a single solution **then**

- Output that first and only solution, stopping the function.

if the current node is exhausted **then**

- Back-track to the last one that is not

if all nodes are exhausted **then**

- The function has finished its task.

`break`

- Using the `'progress'` part of `branch_cands`, return the Sudoku to its state before guesses were made at the current node.

Output: (List of solutions, as `cell_dicts`)

As mentioned, the order at which the nodes are progressed to is predetermined at the beginning. Initially, this was just the order that the list of empty cells found its self, but out of curiosity I tried

changing that order. Two experimental orders were tried, both depended on the same thing, how many candidates the empty cells have. the first was starting with empty cells with the least possibilities, the second was staring with those that had the most. The results of my testing different orders were both mixed, but generally it was found that the ordering of least to most candidates would bring the best time improvements, and in some cases these were significant. For this reason, the final program uses that order. Despite this generally speed boost, the utilized order confusingly makes some Sudokus slower to solve, significantly in a few cases. The only explanation I can think of for these cases is that the order which empty cells naturally falls into is one that is somehow intrinsically beneficial to solving time.

9 Combining All Methods

9.1 `solve`

At this point all of the functions that are necessary to complete Sudokus have been defined, we can input the puzzle to the program and manipulate it with an arsenal of solving methods. Equipped with these functions, `solve` will produce the solution to any given Sudoku.

To carry out its mission, `solve` executes a few steps: First, it fills the initial puzzle data, `cell_dict`, `ref_dict`, and `empty_cells`; then after an initial markup, the function iterates solving methods in a `while` loop of the same structure stated in §7.3; if this loop does not complete the puzzle, `branching_tree` is employed to finish the job; finally, `solve` outputs the solution or a list of solutions to the given Sudoku.

It is possible to utilize `elimination_by_contradiction` when solving, this can be done best by including it at the end of the `while` loop. Doing so would enable the loop alone to solve all Sudokus, without the need for `branching_tree`, however, since `branching_tree` completes the same task faster. `elimination_by_contradiction` is not used. If the user wishes to apply only logical methods to a Sudoku, there is an option within `solve` disable `branching_tree`.

When giving a Sudoku to `solve`, there could be a mistake in the input. If so, this will be picked up by `input_square` at the beginning of the line of operation. `input_square` would usually output a `cell_dict` and `ref_dict` which would be passed to the solving functions, but in the case of a false input, it will output a string representing the error. If nothing is done to avoid it, this string will be passed to solving functions which will not be able to process it. To avert this problem, `solve` has this check just after `input_square` is ran:

```
if type(puzzle_data) == str:
    print(puzzle_data)
else:
    (rest of the function)
```

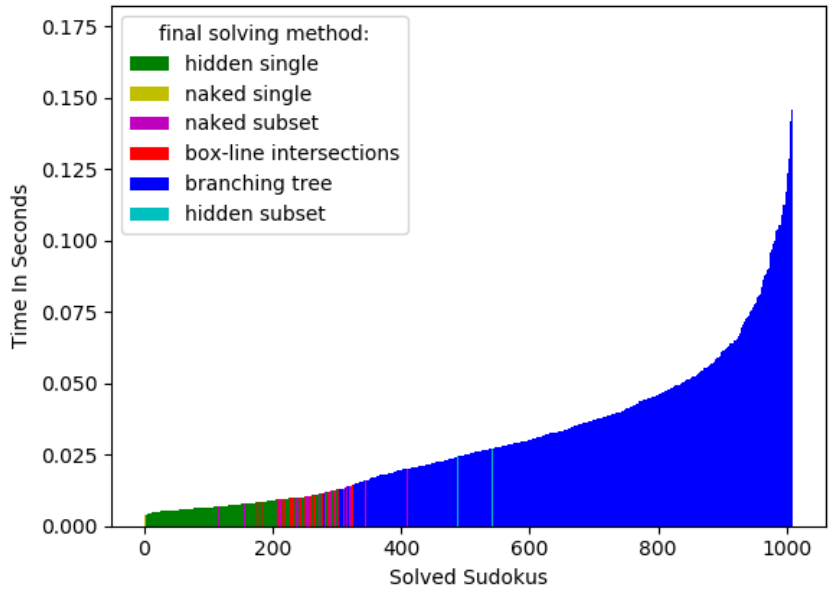
It simply checks to see if `puzzle_data`, the output of `input_square`, is a string. If it is, the string is printed, if not the function continues.

10 Benchmarking Function

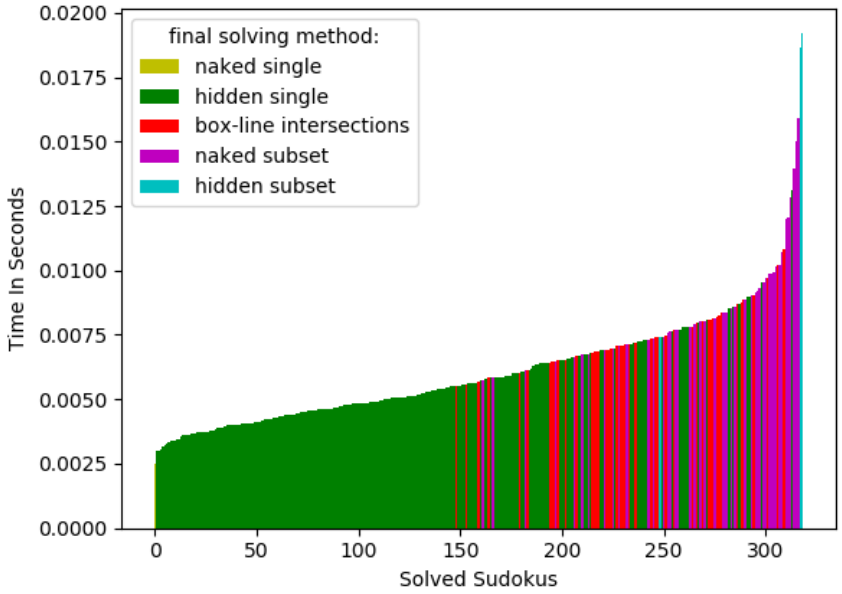
Provided a set of Sudokus, for each puzzle, `benchmark` will apply `solve`, it will record; whether the the puzzle was solved, the time taken, and the most advanced solving method used. To clarify, this is referring to the furthest method along the defined chain that has been able to progress the puzzle. With the stated data collected for every Sudoku in the given set, the function will print the average solving time to the console; it will also produce a bar chart with each bar representing the time to solve a Sudoku. The bars are arranged in order of minimum to maximum time. Each bar's colour corresponds to the most complicated solving method used on that specific Sudoku.

By varying the methods and orders of methods employed in `solve`, the output of `benchmark` will change. For example here is the output for a set of 1011 Sudokus with and without the use of branching tree:

With `branching_tree`



Without `branching_tree`



While running various tests it has been insightful to observe the average solve time and graphical figures outputted by `benchmark`. When making changes to the program, it was by considering the average solve time that the best configuration was chosen. Specifically, this approach informed the decision to remove `hidden_subset` from `apply_logic`, it also showed that ordering the nodes in branching tree was beneficial to performance.

Aside from efficiency, it can be of interest to run benchmark with solving functions disabled in `solve`. Doing so can give insight into which methods are the most powerful, solving more sudokus independently. Running the three logical candidate elimination functions alone with `naked_single`

on the set of 1011 shows that `naked_subset` and `hidden_subset` are considerably more powerful than `box-line_intersection`. It also unveils that despite not being used much in the above figures, `hidden_subset` can solve the most Sudokus independently (see appendix for the graphical outputs of these tests).

The big sets of Sudokus are held in text files obtained from [5]. Every line of the files is made up of a string characters, each character represents the state of a cell with full stops corresponding to empty cells. To work with these sets, `benchmark` employs a couple of other functions: `get_sukokus_from_txt` is used to transfer the information from the text file to Python, it makes a list of all the lines in the file; `string_to_nxn_sudoku_up_to_25x25` is used to convert each of those lines of characters into list of list form, which the program can handle.

As well as the functions to manage the Sudoku sets, `benchmark` also uses a function called `get_data` which will provide information on a given Sudoku by running it through `solve`. Specifically, it reports; whether the Sudoku can be solved, the most advanced solving method that was used, and the time taken. The function saves the time and furthest method for every solved Sudoku in the set. The information is kept in a dictionary called `solve_times`, with this, it can produce the bar chart. A dictionary is used so that the elements can be ordered.

Occasionally, a solving time will arise that is exactly the same as one already recorded. Since the times are saved as keys in the dictionary, this copy time would overwrite the other time when it is found, effectively removing it from the data set. To mediate this issue, avoiding the loss of data, when a time is found that is the same as one previously saved, the copy time and the corresponding furthest method are both saved into lists, `rare_copy_times` and `rare_copy_methods`. Doing this allows the copy data entries to still be used in the output.

References

[1] Crook, J.F. Notices of the American Mathematical Society, vol 56, iss 4, (pp. 460-468).

[2] Jones, S.K., Roach, P.A. and Perkins, S. Properties of Sudoku puzzles. in: *Proceedings of the 2nd Research Student Workshop*. 2007, (pp. 7-11).

[3] McGuire, G., Tugemann, B., Civario, G. There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem. in: CoRR.

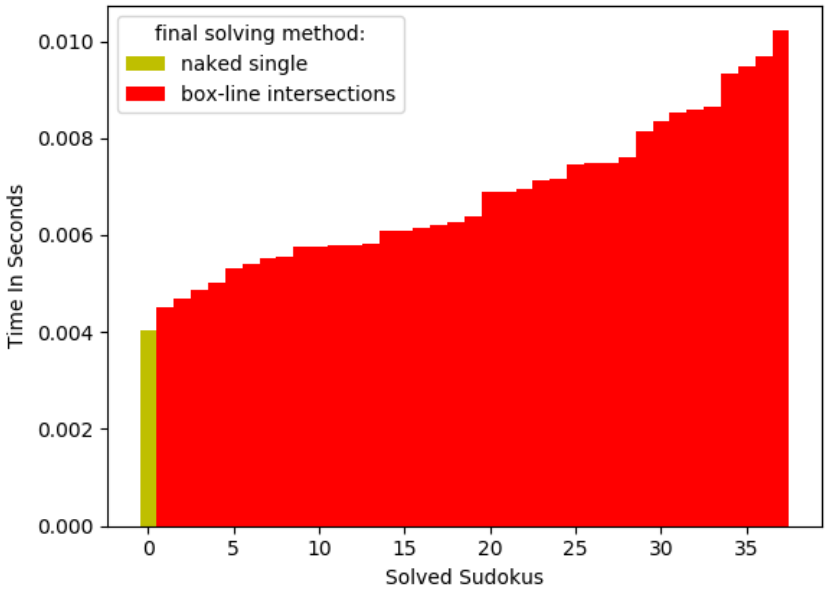
[4] sudoku9981. How to solve sudoku - Solving sudoku strategies. [Online]. 2008. [Accessed January 2018]. Available from: `http://www.sudoku9981.com/sudoku-solving/`

[5] Warwick University. Solving Sudoku puzzles with Python. [Online]. [Accessed Febuary 2018]. Available from: `https://warwick.ac.uk/fac/sci/moac/people/students/peter_cock/python/sudoku/`

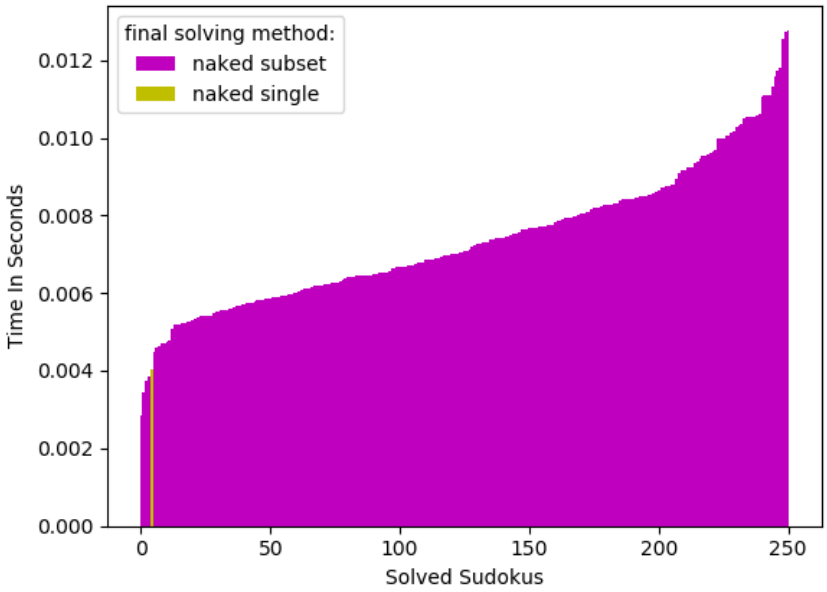
11 Appendix

Experiments Mentioned in § 10

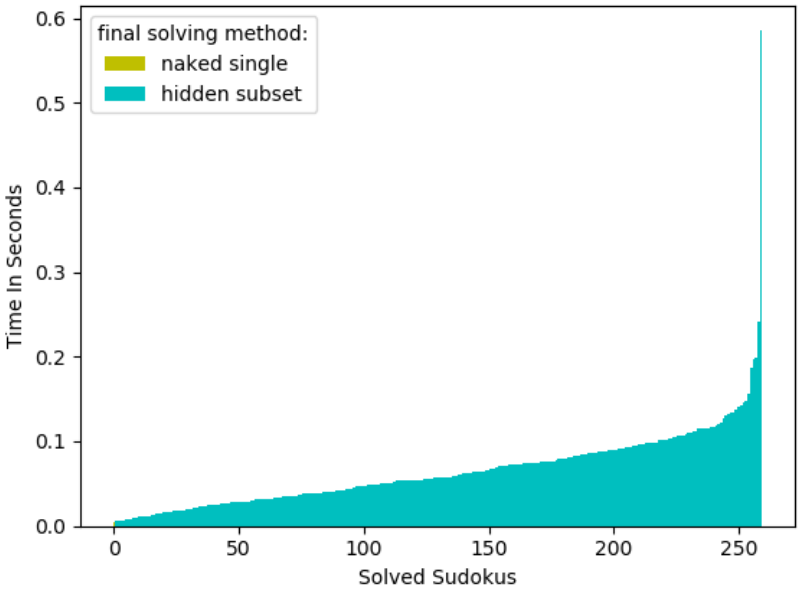
Only `box-line_intersection` And `naked_single`



Only `naked_subset` And `naked_single`



Only `hidden_subset` And `naked_single`



Input And Output Functions

Input Square

```
def input_square(list_of_lists):
    n = int(sqrt(len(list_of_lists)))
    cell_dict = {}
    ref_dict = {}
    #Error Catching
    if not(sqrt(len(list_of_lists)) % 1 == 0):
        return
        ('False entry. Your puzzle dimension is not a square number.')
    for row in list_of_lists:
        if len(row) != len(list_of_lists):
            return('False entry. Your puzzle is not square.')

    '''fills cell dict and ref_dict with values'''
    for i in range(1, n**2+1):
        ref_dict[('row', i)] = []
        ref_dict[('column', i)] = []
    for box_row in range(1, n+1):
        for box_col in range(1, n+1):
            br, bc = box_row, box_col
            ref_dict[('box', (br, bc))] = []
            for r in range((br-1)*n +1, br*n +1):
                for c in range((bc-1)*n +1, bc*n +1):
                    cell = (r,c)
                    cell_val = list_of_lists[r-1][c-1]
                    cell_box_index = (br, bc)
                    if cell_val == 0:
                        cell_dict[cell] = list(range(1,n**2+1))
                    else:
                        cell_dict[cell] = cell_val
                        ref_dict[('row',r)].append(cell_val)
                        ref_dict[('column',c)].append(cell_val)
                        ref_dict[('box',cell_box_index)].append(cell_val)
    #Error Catching
    for house in ref_dict.keys():
        for value in ref_dict[house]:
            if value != 0:
                if value > n**2:
                    return('False entry, value: %s is too big.'
                           % str(value))
                if value < 0:
                    return('False entry, value: %s is too small.'
                           % str(value))
                if not(type(value) == int):
                    return('False entry, value: %s is not an integer.'
                           % str(value))
                if ref_dict[house].count(value) > 1:
                    return('False entry, %s is repeated in '
                           %str(value)+house[0]+' '+str(house[1]))
    return(cell_dict, ref_dict)
```

Print Puzzle

```
def print_puzzle(cell_dict):
    ##########################
    hide_candidates = True
    ##########################
    n = int(sqrt(sqrt(len(cell_dict))))
    sudoku = []
    character_lengths = []


    for cell in cell_dict:
        character = cell_dict[cell]
        if hide_candidates:
            if type(character) == list:
                character = '␣'
        character_lengths.append(len(str(character)))
    max_len = max(character_lengths)

    for r in range(1, n**2+1):
        row = []
        for c in range(1, n**2+1):
            character = cell_dict[(r,c)]
            if hide_candidates:
                if type(character) == list:
                    character = '␣'
            length = len(str(character))
            fill = max_len - length
            row.append(fill*'␣'+str(character))

        x=0
        for i in range(n+1):
            row.insert(i*n+x,'|')
            x+=1
        sudoku.append(row)

    horizontal_boundary =
    '|'+
    ('-'*(max_len*n+n+1)+'+')*(n-1) +
    '-'*(max_len*n+n+1)+
    '|'

    top_bottom = '␣'+'-'*(len(horizontal_boundary) - 2)
    y=0
    for i in range(1,n):
        sudoku.insert(i*n+y, horizontal_boundary)
        y+=1

    print(top_bottom)
    for row in sudoku:
        if type(row) == list:
            print('␣'.join(row))
        else:
            print(row)
    print(top_bottom)
```

Fill A Cell, To Facilitate LSMs

```
def fill_a_cell(cell_dict, ref_dict, cell, value):
    n = int(sqrt(sqrt(len(cell_dict))))
    row    = cell[0]
    column = cell[1]
    box    = (row+n-1)//n, (column+n-1)//n
    cell_dict[cell] = value
    ref_dict[('row', row)].append(value)
    ref_dict[('column', column)].append(value)
    ref_dict[('box', box)].append(value)
```

Markup Specific

```
def markup_specific(cell_dict, ref_dict, cells_to_markup):
    n = int(sqrt(sqrt(len(cell_dict))))
    for cell in cells_to_markup:
        #Information corresponding to the cell.
        row    = cell[0]
        column = cell[1]
        box    = (row+n-1)//n, (column+n-1)//n
        #Find impossible candidates.
        row_vals    = ref_dict[('row', row)]
        column_vals = ref_dict[('column', column)]
        box_vals    = ref_dict[('box', box)]
        markup_vals = row_vals + column_vals + box_vals
        #Remove impossible candidates.
        for val in markup_vals:
            if val in cell_dict[cell]:
                cell_dict[cell].remove(val)
    return(cell_dict)
```

Markup Relevant

```
def markup_relevant(cell_dict, ref_dict, filled, empty_cells):
    n = int(sqrt(sqrt(len(cell_dict))))
    to_markup = []
    relevant_houses = {'rows':[], 'columns':[], 'boxes':[]}
    #fills relevant_rcb
    for cell in filled:
        box_index = (cell[0]+n-1)//n, (cell[1]+n-1)//n
        if not(cell[0] in relevant_houses['rows']):
            relevant_houses['rows'].append(cell[0])
        if not(cell[1] in relevant_houses['columns']):
            relevant_houses['columns'].append(cell[1])
        if not( (box_index) in relevant_houses['boxes']):
            relevant_houses['boxes'].append((box_index))
    #Find cells to be marked up
    for cell in empty_cells:
        row    = cell[0]
        column = cell[1]
        box    = ( row +n-1)//n, ( column +n-1)//n
        if row in relevant_houses['rows']:
            to_markup.append(cell)
            continue
        elif column in relevant_houses['columns']:
            to_markup.append(cell)
            continue
        elif box in relevant_houses['boxes']:
            to_markup.append(cell)
            continue
    #Mark up appropriate cells
    markup_specific(cell_dict, ref_dict, to_markup)

    return(cell_dict)
```

Naked Single

```
def naked_single(cell_dict, ref_dict, empty_cells):
    filled = []
    for cell in empty_cells:
        if len(cell_dict[cell]) == 1:
            n_single = cell_dict[cell][0]
            fill_a_cell(cell_dict, ref_dict, cell, n_single)
            filled.append(cell)
    return(cell_dict, ref_dict, filled)
```

Hidden Single

```
def hidden_single(cell_dict, ref_dict, cell_cand_rcb):
    filled = []
    for house in cell_cand_rcb:
        house_candidates = []
        for cell in cell_cand_rcb[house]:
            for cand in cell_cand_rcb[house][cell]:
                house_candidates.append(cand)
        for cell in cell_cand_rcb[house]:
            if cell in filled:
                continue # Stops the function considering cells that
                         it has already filled.

            for cand in cell_cand_rcb[house][cell]:
                if house_candidates.count(cand) == 1:
                    h_single = cand
                    fill_a_cell(cell_dict, ref_dict, cell, h_single)
                    filled.append(cell)
    return(cell_dict, ref_dict, filled)
```

Box-Line Intersection

```
def box_line_intersection(cell_dict, cell_cand_rcb):
    n = int(sqrt(sqrt(len(cell_dict))))
    intercection_dict = {}
    ##Fills intercection_dict
    #Fills intercection_dict with boxes that have empty cells
    for house in cell_cand_rcb:
        if house[0] == 'box':
            intercection_dict[house] = {}
    #Fills intercection_dict with rows and columns that have empty cells
    for house in cell_cand_rcb:
        if house[0] != 'box':
            for empty_cell in cell_cand_rcb[house]:
                row = empty_cell[0]
                col = empty_cell[1]
                box = ( row +n-1)//n, ( col +n-1)//n
                if not(house in intercection_dict[('box', box)]):
                    intercection_dict[('box',box)][house] =
                    {'in':[],'out':{'sameline':[],'samebox':[]}}

                intercection_dict[('box',box)][house]['in']
                .append(empty_cell)

    #Fills intercection_dict with candidates for cells outside the
    #intersections
    for box in intercection_dict:
        horizontal_box_range =
        range( (box[1][1]-1)*n +1, box[1][1]*n +1 )

        verticle_box_range   =
        range( (box[1][0]-1)*n +1, box[1][0]*n+1 )
```

```
        dummy_lines = []
        for line in intercection_dict[box]:
            #Row intersections
            if line[0] == 'row':
                #Cells in the same row as intersection, not the box
                for empty_cell in cell_cand_rcb[line]:
                    col = empty_cell[1]
                    if not(col in horizontal_box_range):
                        intercection_dict[box][line]['out']['sameline']
                        .append(empty_cell)

                #Cells in the same box as intersection, not the same row
                for empty_cell in cell_cand_rcb[box]:
                    row = empty_cell[0]
                    if not(row == line[1]):
                        intercection_dict[box][line]['out']['samebox']
                        .append(empty_cell)

            #Column intersections
            if line[0] == 'column':
                #Cells in the same column as intersection, not the box
                for empty_cell in cell_cand_rcb[line]:
                    row = empty_cell[0]
                    if not(row in verticle_box_range):
                        intercection_dict[box][line]['out']['sameline']
                        .append(empty_cell)

                #Cells in the same box as intersection, not the column
                for empty_cell in cell_cand_rcb[box]:
                    col = empty_cell[1]
                    if not(col == line[1]):
                        intercection_dict[box][line]['out']['samebox']
                        .append(empty_cell)

            #Removes obviously usless rows from slicing_dict with no
            #empty cells in the line or box outside the overlap
            if (len(intercection_dict[box][line]['out']['samebox']) ==
            0 or
            len(intercection_dict[box][line]['out']['sameline']) ==
            0):

                dummy_lines.append(line)
        for line in dummy_lines:
            del intercection_dict[box][line]

    #Uses intersection_dict to implement the box-line intersection method
    for box in intercection_dict:
        for line in intercection_dict[box]:
            #Candidates for empty cells in the overlap
            intercection_candidates = []
            for cell in intercection_dict[box][line]['in']:
                intercection_candidates += cell_cand_rcb[line][cell]
            #Candidates for empty cells that are not in the overlap
            #but are in the same line
            sameline_candidates = []
            for cell in intercection_dict[box][line]['out']['sameline']:
```

```
            sameline_candidates += cell_dict[cell]
        #Candidates for empty cells that are not in the overlap
        #but are in the same box
        samebox_candidates = []
        for cell in intercection_dict[box][line]['out']['samebox']:
            samebox_candidates += cell_dict[cell]
        #Checks each candidate in every intersection overlay to see
        #if it satisfies the right conditions
        for cand in intercection_candidates:
            #Candidate is in the same line
            in_line   =   (sameline_candidates.count(cand) != 0)
            #Candidate is in the same box
            in_box   =   (samebox_candidates.count(cand) != 0)
            #Checks for Type 1 (see section 2.3 of report)
            if in_box and not(in_line):
                for cell in
                intercection_dict[box][line]['out']['samebox']:

                    if cand in cell_dict[cell]:
                        cell_dict[cell].remove(cand)
            #Checks for Type 2 (see section 2.3 of report)
            if in_line and not(in_box):
                for cell in
                intercection_dict[box][line]['out']['sameline']:

                    if cand in cell_dict[cell]:
                        cell_dict[cell].remove(cand)
return(cell_dict)
```

Naked Subset

```
def naked_subset(cell_dict, cell_cand_rcb):
    subsets = {}
    #Looks for Naked Subsets
    for house in cell_cand_rcb:
        #Skips houses that contain less that 3 empty cells
        if len(cell_cand_rcb[house]) < 3:
            continue
        ignore = []
        for cell in cell_cand_rcb[house]:
            candidate_group = cell_cand_rcb[house][cell]
            #Skips groups of candidates that have the same number of
            #elements as there is empty cells
            if len(candidate_group) == len(cell_cand_rcb[house]):
                continue
            #Checks if the subset has already been looked at in the house
            elif candidate_group in ignore:
                continue
            else:
                ignore.append(candidate_group)
                #Fills a single element of subsets.
                other_cells = [] #cells not in the Naked Subset
                subsets[str(candidate_group)] = []
                for cell_1 in cell_cand_rcb[house]:
                    subset = True
                    for cand in cell_cand_rcb[house][cell_1]:
                        if not(cand in candidate_group):
                            subset = False
                            other_cells.append(cell_1)
                            break
                    if subset:
                        subsets[str(candidate_group)].append(cell_1)

                #Checks if a group of candidates is a Naked Subset for
                #the house.
                number_of_cands = len(candidate_group)
                number_of_cells = len(subsets[str(candidate_group)])
                #The group is not a Naked Subset.
                if number_of_cells < number_of_cands:
                    del subsets[str(candidate_group)]
                #The group is a Naked Subset.
                elif number_of_cells == number_of_cands:
                    naked_subset = candidate_group
                    for cand in naked_subset:
                        for cell_1 in other_cells:
                            if cand in cell_cand_rcb[house][cell_1]:
                                cell_dict[cell_1].remove(cand)
    return(cell_dict)
```

Hidden Subset

```
def hidden_subset(cell_dict, ref_dict, cell_cand_rcb):
    n = int(sqrt(sqrt(len(cell_dict))))
    cand_pos_rcb = {}
    #fills cand_pos_rcb
    for house in cell_cand_rcb:
        ################################## Only works properly if
        #hidden_single is also used, comment out if not
        if len(cell_cand_rcb[house]) < 4:
            continue
        ##################################
        cand_pos_rcb[house] =
        {'cells': cell_cand_rcb[house] ,'cand_pos':{}}

    for house in cand_pos_rcb:
        #Finds which candtdates are in that house
        house_candidates = list(range(1,n**2+1))
        for cand in ref_dict[house]:
            house_candidates.remove(cand)
        #Fills 'cand_pos' part of cand_pos_rcb
        for cand in house_candidates:
            cand_pos_rcb[house]['cand_pos'][cand] = []
            for cell in cand_pos_rcb[house]['cells']:
                if cand in cand_pos_rcb[house]['cells'][cell]:
                    cand_pos_rcb[house]['cand_pos'][cand].append(cell)
        ###Finds Hidden Subsets
        ##Finds Basic Hidden Subsets
        before = str(list(cell_dict.values()))
        ignore = []
        for cand in house_candidates:
            #Skips useless candidates.
            if cand in ignore:
                continue
            #The cells that have 'cand' as a candidate.
            cell_set = cand_pos_rcb[house]['cand_pos'][cand]
            #The number of candidates in the house with exactly the
            #same cell set.
            cands_same_cells_set = (
            list(cand_pos_rcb[house]['cand_pos'].values())
            .count(cell_set))

            #Checks whether the cell set contains a Hidden Subset.
            if cands_same_cells_set == len(cell_set):
                #Finds the candidates in the Hidden Subset.
                Hidden_Subset = []
                for cand1 in house_candidates:
                    if cand_pos_rcb[house]['cand_pos'][cand1] ==
                    cell_set:

                        Hidden_Subset.append(cand1)
                        ignore.append(cand1)
                #Removes candidates that are not in the Hidden Subset
                #from cells in the span
                for cell in cell_set:
                    for cand1 in cand_pos_rcb[house]['cells'][cell]:
```

```
                    if not(cand1 in Hidden_Subset):
                        cell_dict[cell].remove(cand1)
    #If candidates have been removed, return cell_dict
    after = str(list(cell_dict.values()))
    if before != after:
        return(cell_dict)
    ##Finds Complex Hidden Subsets
    hc = house_candidates
    if len(hc) < 5:
        continue
    combos = {3:[]}
    hc.sort()
    all_combos = []
    ##finds all suitable combinations of candidates
    #Finds initial length 3 combos.
    for i in range(len(hc)):
        for j in list(range(i+1,len(hc))):
            for k in list(range(j+1,len(hc))):
                combos[3].append([hc[i],hc[j],hc[k]])
                all_combos.append([hc[i],hc[j],hc[k]])
    #Finds combos longer that length 3
    for length in range(4,len(hc) - 1):
        combos[length] = []
        for combo in combos[length - 1]:
            biggest = combo[-1]
            for i in range(len(combo), len(hc)):
                if hc[i] > biggest:
                    combos[length].append(combo + [hc[i]])
                    all_combos.append(combo + [hc[i]])
    #For every combo, check if its a Hidden Subset.
    for cand_set in all_combos:
        try:
            span = []
            for cand in cand_set:
                #Checks if cand_set not a Hidden Subset
                if len(cand_pos_rcb[house]['cand_pos'][cand]) >
                len(cand_set):

                    raise StopIteration
                #Finds all the cells that are spanned by the set.
                for cell in cand_pos_rcb[house]['cand_pos'][cand]:
                    if not(cell in span):
                        span.append(cell)
            #Checks if the candidate set is not a Hidden Subset.
            if len(span) != len(cand_set):
                raise StopIteration
            #Otherwise the candidate set is a Hidden Subset.
            else:
                #Removes appropriate candidates.
                for cell in span:
                    for cand in cand_pos_rcb[house]['cells'][cell]:
                        if not(cand in cand_set):
                            cell_dict[cell].remove(cand)
        except StopIteration:
            continue
return(cell_dict)
```

Trial And Error Solving Methods

Functions To Facilitate T and E SMs

```
def check_for_errors(ref_dict, cell_dict):
    #checks for repeated values in a house
    for house in ref_dict:
        for value in ref_dict[house]:
            if ref_dict[house].count(value) != 1:
                return True
    #checks for cells with no posibilitys
    for cell in cell_dict:
        if cell_dict[cell] == []:
            return True


def apply_logic(cell_dict, ref_dict, filled_cells, empty_cells):
    n = int(sqrt(sqrt(len(cell_dict))))
    cd, rd = cell_dict, ref_dict
    #initial markup
    cd = markup_relevant(cd, rd, filled_cells, empty_cells)
    #solving loop
    not_stuck = True
    while not_stuck:
        if check_for_errors(rd, cd):
            return(['error'])
        cell_cand_rcb = {}
        filled = []
        before = str(list(cd.values()))

        '''Naked Single Method'''
        new = naked_single(cd, rd, empty_cells)
        cd, rd, filled = new[0], new[1], new[2]
        if len(filled) != 0:
            for cell in filled:
                empty_cells.remove(cell)
            cd = markup_relevant(cd, rd, filled, empty_cells)
        after = str(list(cd.values()))
        not_stuck = (before != after)
        if not_stuck:
            continue

        #Fills cell_cand_rcb.
        for cell in empty_cells:
            row = cell[0]
            col = cell[1]
            box = ( row +n-1)//n, ( col +n-1)//n
            houses = {'row': row,'column': col,'box': box}
            for house in houses:
                if not((house, houses[house]) in
                list(cell_cand_rcb.keys())):
                    cell_cand_rcb[(house, houses[house])] = {}
                cell_cand_rcb[(house, houses[house])][cell] = cd[cell]
```

```
        '''Hidden Single Method'''
        new = hidden_single(cd, rd, cell_cand_rcb)
        cd, rd, filled = new[0], new[1], new[2]
        if len(filled) != 0:
            for cell in filled:
                if cell in empty_cells:
                    empty_cells.remove(cell)
            cd = markup_relevant(cd, rd, filled, empty_cells)
        after = str(list(cd.values()))
        not_stuck = (before != after)
        if not_stuck:
            continue

        '''Box-Line Intersection Method'''
        cd = box_line_intersection(cd, cell_cand_rcb)
        after = str(list(cd.values()))
        not_stuck = (before != after)
        if not_stuck:
            continue

        '''Naked Subset Method'''
        cd = naked_subset(cd, cell_cand_rcb)
        after = str(list(cd.values()))
        not_stuck = (before != after)
        if not_stuck:
            continue

        '''Hidden Subset Method'''
        '''Not used here for reasons discussed at the end of
           section 7.2'''
#       cd = hidden_subset(cd, rd, cell_cand_rcb)
#       after = str(list(cd.values()))
#       not_stuck = (before != after)
#       if not_stuck:
#           continue

    if len(empty_cells) == 0:
        return( ('solution', cd) )

    return( ('stuck', (cell_dict, ref_dict, empty_cells)) )
```

Elimination By Contradiction

```
def elimination_by_contradiction(cell_dict, ref_dict, empty_cells):
    options = {}
    for cell in empty_cells:
        options[cell] = cell_dict[cell][:]

    options_left = True
    while options_left:
        #reset Sudoku data
        copy_cd = deepcopy(cell_dict)
        copy_rd = deepcopy(ref_dict)
        copy_ec = deepcopy(empty_cells)
        #choose a value to apply
        trial_cell = list(options.keys())[0]
        trial_value = options[trial_cell][0]
        options[trial_cell].remove(trial_value)
        if len(options[trial_cell]) == 0:
            del options[trial_cell]
        if len(options) == 0:
            options_left = False
        #apply trial to Sudoku data
        fill_a_cell(copy_cd, copy_rd, trial_cell, trial_value)
        copy_ec.remove(trial_cell)
        filled_cell = [trial_cell]

        data = apply_logic(copy_cd, copy_rd, filled_cell, copy_ec)
        outcome = data[0]

        if outcome == 'error':
            cell_dict[trial_cell].remove(trial_value)
            continue

    return(cell_dict)
```

Branching Tree

```
def branching_tree(cell_dict, ref_dict, empty_cells):
    solutions = []
    #ordered empty cells
    ordered_ec = []
    #Initial puzzle data
    copy_cd = deepcopy(cell_dict)
    copy_rd = deepcopy(ref_dict)
    n = int(sqrt(sqrt(len(cell_dict))))
    ##############################################
    #sorts least candidates to most candidates

    for x in range(2,n**2+1):
        for node in empty_cells:
            if len(cell_dict[node]) == x:
                ordered_ec.append(node)
    ##############################################
```

```
    ##########################################
    #sorts most candidates to least candidates

#     for x in reversed(range(2,n**2+1)):
#         for node in empty_cells:
#             if len(cell_dict[node]) == x:
#                 ordered_ec.append(node)
    ##########################################
    #doesn't sort

#     ordered_ec = empty_cells[:]
    ##########################################

    #Fills the first branch_cands element
    branch_ID = [ordered_ec[0]]
    branch_cands = {str(branch_ID):
                    {'progress': (copy_cd, copy_rd, ordered_ec),
                     'posibilitys': cell_dict[ordered_ec[0]]}}

    more_combos = True
    while more_combos:
        #Choose guess
        trial_cell  = branch_ID[-1]
        trial_value = branch_cands[str(branch_ID)]['posibilitys'][0]
        #Remove guess from posibilitys.
        branch_cands[str(branch_ID)]['posibilitys'].remove(trial_value)
        #Add to the data form of the sudoku.
        fill_a_cell(cell_dict, ref_dict, trial_cell, trial_value)
        empty_cells.remove(trial_cell)
        #Determins the outcome of the trial.
        filled_cells = [trial_cell]
        data =
        apply_logic(cell_dict, ref_dict, filled_cells, empty_cells)

        #Outcome of guess.
        outcome = data[0]
        #print(trial_value, trial_cell)

        if outcome == 'stuck':
            cell_dict   = data[1][0]
            ref_dict    = data[1][1]
            empty_cells = data[1][2]

            #decide on next node
            for node in ordered_ec:
                if node in branch_ID:
                    continue
                elif node in empty_cells:
                    next_node = node
                    break
                else:
                    continue
            # add next node
            branch_ID.append(next_node)
            current_node = branch_ID[-1]
```

```
            #fills posibilitys
            branch_cands[str(branch_ID)] = {}
            branch_cands[str(branch_ID)]['posibilitys'] =
            cell_dict[current_node]

            #fills progress
            CD_prog = deepcopy(cell_dict)
            RD_prog = deepcopy(ref_dict)
            EC_prog = deepcopy(empty_cells)
            branch_cands[str(branch_ID)]['progress'] =
            (CD_prog, RD_prog, EC_prog)

        else:
            if outcome == 'solution':
                cell_dict    = data[1]
                solution = deepcopy(cell_dict)
                ##########################################
                #Make false to find single solutions faster

                find_all_solutions =  True
                ##########################################
                if find_all_solutions:
                    solutions.append(solution)
                else:
                    return([solution])

            node_posibilitys =
            branch_cands[str(branch_ID)]['posibilitys']

            #Backtracks to the apropriate node if this one is exauhsted
            while len(node_posibilitys) == 0:
                if branch_ID == [ordered_ec[0]]:
                    more_combos = False
                    break
                del branch_ID[-1]
                node_posibilitys =
                branch_cands[str(branch_ID)]['posibilitys']

            if not(more_combos):
                break
            #finds current progression of the puzzle for
            #the specific branch id
            progress = branch_cands[str(branch_ID)]['progress']
            cell_dict    = deepcopy(progress[0])
            ref_dict     = deepcopy(progress[1])
            empty_cells = deepcopy(progress[2])

    return(solutions)
```

Solve

```
def solve(sudoku):
    puzzle_data = input_square(sudoku)
    #handles errors
    if type(puzzle_data) == str:
        print(puzzle_data)
```

```
else:
    methods_dict = {}
    methods = []
    empty_cells = []
    cd = puzzle_data[0]# cd refers to cell_dict
    rd = puzzle_data[1]#rd refers to ref_dict
    n = int(sqrt(sqrt(len(cd))))
    #determines initial empty cells
    for cell in cd:
        if type(cd[cell]) == list:
            empty_cells.append(cell)
    #initial markup
    cd = markup_specific(cd, rd, empty_cells)
    #solving loop
    not_stuck = True
    while not_stuck:
        cell_cand_rcb = {}
        before = str(list(cd.values()))

        '''Naked Single Method'''
        new = naked_single(cd, rd, empty_cells)
        cd, rd, filled = new[0], new[1], new[2]
        if len(filled) != 0:
            for cell in filled:
                empty_cells.remove(cell)
            if len(empty_cells) == 0:
                break
            cd = markup_relevant(cd, rd, filled , empty_cells)
        after = str(list(cd.values()))
        not_stuck = (before != after)
        if not_stuck:
            if not('n-sin' in methods_dict.values()):
                methods_dict[0] = 'n-sin'
            continue

        #fills cell_cand_rcb
        for cell in empty_cells:
            row = cell[0]
            col = cell[1]
            box = ( row +n-1)//n, ( col +n-1)//n
            houses = {'row': row,'column': col,'box': box}
            for house in houses:
                if not((house, houses[house]) in
                list(cell_cand_rcb.keys())):

                    cell_cand_rcb[(house, houses[house])] = {}
                cell_cand_rcb[(house, houses[house])][cell] =
                cd[cell]

        '''Hidden Single Method'''
        new = hidden_single(cd, rd, cell_cand_rcb)
        cd, rd, filled = new[0], new[1], new[2]
        if len(filled) != 0:
            for cell in filled:
                empty_cells.remove(cell)
            if len(empty_cells) == 0:
```

```
                    break
                cd = markup_relevant(cd, rd, filled, empty_cells)
                after = str(list(cd.values()))
            not_stuck = (before != after)
            if not_stuck:
                if not('h-sin' in methods_dict.values()):
                    methods_dict[1] = 'h-sin'
                continue

            '''Box-Line Intersection Method'''
            cd = box_line_intersection(cd, cell_cand_rcb)
            after = str(list(cd.values()))
            not_stuck = (before != after)
            if not_stuck:
                if not('b-l-int' in methods_dict.values()):
                    methods_dict[2] = 'b-l-int'
                continue

            '''Naked Subset Method'''
            cd = naked_subset(cd, cell_cand_rcb)
            after = str(list(cd.values()))
            not_stuck = (before != after)
            if not_stuck:
                if not('n-sub' in methods_dict.values()):
                    methods_dict[3] = 'n-sub'
                continue

            '''Hidden Subset Method'''
            cd = hidden_subset(cd, rd, cell_cand_rcb)
            after = str(list(cd.values()))
            not_stuck = (before != after)
            if not_stuck:
                if not('h-sub' in methods_dict.values()):
                    methods_dict[4] = 'h-sub'
                continue

            '''Elimination By Contradiction Method'''
            '''Intentionally not used because it is slow.'''
#           cd = elimination_by_contradiction(cd, rd, empty_cells)
#           after = str(list(cd.values()))
#           not_stuck = (before != after)
#           if not_stuck:
#               if not('cont' in methods_dict.values()):
#                   methods_dict[5] = 'cont'
#               continue

        ##################################
        use_branching_tree = True
        ##################################
        if use_branching_tree:
            '''Branching Tree Method'''
            if not(len(empty_cells) == 0):
                solutions = branching_tree(cd, rd, empty_cells)
                methods_dict[6] = 'tree'
            else:
                solutions = [cd]
```

```
        else:
            solutions = [cd]


        #fills list of methods used
        for i in range(7):
            if i in methods_dict.keys():
                methods.append(methods_dict[i])
        #outputs solution(s)
        if len(solutions) == 1:
            solution = solutions[0]
            return(solution, methods)
        else:
            return(solutions, methods)
```

Extra Functions

```
def string_to_nxn_sudoku_up_to_25x25(string):
    n = int(sqrt(sqrt(len(string))))
    numbers = ['1','2','3','4','5','6','7','8','9']
    letters = ['A','B','C','D','E','F','G','H',
    'I','J','K','L','M','N','O','P']

    sudoku = []
    x = 0
    for i in range(n**2):
        temp = []
        for j in range(n**2):
            if string[x] == '.':
                temp.append(0)
                x += 1
            elif string[x] in numbers:
                temp.append(int(string[x]))
                x += 1
            elif string[x] in letters:
                value = 10 + letters.index(string[x])
                temp.append(value)
                x += 1
        sudoku.append(temp)
    return(sudoku)
```

```
def solve_show(sudoku):
    start = perf_counter()
    data = solve(sudoku)
    end   = perf_counter()
    if not(type(data) == tuple):
        return
    else:
        solutions = data[0]
        time = end - start
        if type(solutions) == dict:
            print_puzzle(solutions)
            if data[0]:
                print('Solution found in', round(time,4), 'seconds.')
            elif not(data[0]):
                print('This was found in', round(time,4), 'seconds.')
        elif type(solutions) == list:
            for solution in solutions:
                print_puzzle(solution)
            print(len(solutions), 'solutions. Found in',
            round(time,4), 'seconds.')
```

```
def solve_show_str(string):
    sudoku = string_to_nxn_sudoku_up_to_25x25(string)
    solve_show(sudoku)
```

```
def get_data(sudoku):
    start = perf_counter()
    data = solve(sudoku)
    end = perf_counter()
    cell_dict = data[0]
    can_solve = True
    for value in cell_dict.values():
        if type(value) == list:
            can_solve = False
            break
    time = end - start
    methods_used = data[1]
    return(can_solve, time, methods_used)
```

```
def get_sukokus_from_txt(filename):
    if not(filename[-4:] == '.txt'):
        filename += '.txt'
    sudokus = []
    with open(filename,'r') as file:
        while True:
            line = file.readline().replace('\n','')
            if line == '':
                break
            else:
                sudokus.append(line)
    return sudokus



def print_puzzle_str(string):
    list_of_lists = string_to_nxn_sudoku_up_to_25x25(string)
    cell_dict = input_square(list_of_lists)[0]
    print_puzzle(cell_dict)



def mean(lst):
    total = 0
    for i in lst:
        total += i
    if len(lst) != 0:
        return(total/len(lst))
    else:
        return(0)
```

Benchmark

```
def benchmark(filename):
    total = 0
    solved = 0
    solve_times = {}
    rare_copy_times = []
    rare_copy_methods = []

    for string in get_sukokus_from_txt(filename):
        sudoku = string_to_nxn_sudoku_up_to_25x25(string)
        data = get_data(sudoku)
        its_solved  = data[0]
        time    = data[1]
        methods = data[2]
        if its_solved:
            #deals with the case where you get a time exactly
            #the same as another
            if time in solve_times.keys():
                rare_copy_times.append(time)
                rare_copy_methods.append(methods)
            solve_times[time] = methods
            solved += 1
        total += 1
```

```
times_list = list(solve_times.keys())
times_list.sort()
mean_time = mean(list(solve_times.keys()))
t = rare_copy_times[:]
m = rare_copy_methods[:]
x = 0
used = []
c = {'n-sin':'y',
     'h-sin':'g',
     'b-l-int':'r',
     'n-sub':'m',
     'h-sub':'c',
     'cont':'k',
     'tree': 'b'
     }
l = {'n-sin':'naked single',
     'h-sin':'hidden single',
     'b-l-int':'box-line intersections',
     'n-sub':'Naked Subset',
     'h-sub':'Hidden Subset',
     'cont':'trial and contradiction',
     'tree':'branching tree'
     }

plt.figure(filename + '. Average time: ' +
str(mean_time) + ' seconds.')

for time in times_list:
    #plots data for the rare copy times
    if time in rare_copy_times:
        while time in t:
            place = t.index(time)
            plt.bar(x,t[place], color = c[m[place][-1]], width = 1)
            if not(m[place][-1] in used):
                used.append(m[place][-1])
            x += 1
            del t[place]
            del m[place]
    #plots data for the regular times
    plt.bar(x,time, color = c[solve_times[time][-1]], width = 1)
    if not(solve_times[time][-1] in used):
        used.append(solve_times[time][-1])
    x += 1
#to make the legend
for method in used:
    plt.bar(0,0,color = c[method], label = l[method])
plt.ylabel('Time in Seconds')
plt.xlabel('Each bar is a solved puzzle')
plt.title('The respective times taken to solve '+str(solved)+
' out of a '+'set of '+str(total)+
' sudokus \n aranged in order of time')

plt.legend(loc = 2, title = 'final solving method:')
print('Solved ',solved,' out of ',total, 'Sudokus.')

print('Average solve time:', round(mean_time,4), 'seconds.')
```