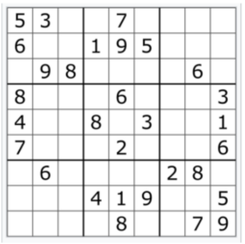
1. Introduction

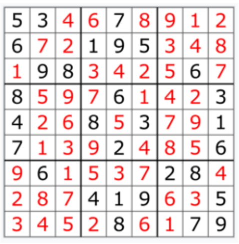
1.1 The Problem of SUDOKU

Sudoku is a logic-based, number-placement puzzle. Basically speaking, all cubes in the 9×9 Sudoku grid are filled with a digit from 1-9. To have a completely filled grid with no spaces, we should make sure that each digit from 1-9 only appear once in every row, column, and block.



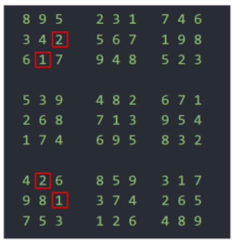
5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Fig 1.1 Partially filled Sudoku Grid (Sudoku Puzzle)



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

Fig 1.2 Completely filled Sudoku Grid



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

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

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

Fig 1.3 Bad Digging Case

Some digits are then dug out to generate a partially filled grid, which require players to fill in to recover the fully filled grid.

1.2 Rules of Sudoku

Here are some rules for Sudoku for players and the quiz setting, basically:

1. Players must fill with **digits in 1-9**;
2. Players are **not allowed to repeat** filling any digits in its own row, column, and box;
3. The well-defined quiz is supposed to **have only one solution**. We should avoid bad cases like digging digits in red rectangle in ***Fig 1.3***, which leads to multiple solutions.

We need to design reasonable solving and generating algorithm that do not violates the restrictions.

1.3 Goals and Achievements Our Project

Our goal is to build algorithms that solve Sudoku problems in and end-to-end manner:

1. Generate well-defined Sudoku grid given number of blanks.
2. Solve quizzes with algorithms from 1, or from reality.
3. Propose metric that quantifies given sudokus' difficulty.

To solve Sudoku, we propose solvers in 3.1, and evaluate their efficiency under randomly generated quizzes in 4.1; as for generating, some brand-new techniques are proposed, along with the naïve one in 3.3; regarding difficulty evaluation, we prove that the average blanks' candidates (ABC) for a sudoku can intuitively, yet effectively quantify the degree of difficulty in 3.5 and 4.3.

1.4 Terminology and Notation

Cell: Each small square on the sudoku board, with or without a digit from 1-9.
Grid: The overall number-filling structure of Sudoku has a total of 9×9=81 cells.
Puzzle: The partially filled Sudoku grid.
Box: Every 3×3 block of cells, complete if containing 9 unique digits from 1-9 in its cells.
Clue: The number hints given in the original Sudoku for reasoning.
Space/blanks: The dug cells in the grid that require players to fill in.
Peer: For the given cell, peers are other cells in the same row, column, and box.
Candidates: All possible solutions for a cell, excluding digits appeared in cell's peers'.

2. Methodology for SUDOKU

2.1 Solving Algorithms

The solving algorithms are fundamental, since we greatly rely on solvers to diagnose properties of quizzes, and to generate well-defined puzzles. Hence, it will be discussed first.
Regarding techniques, we mainly focused on data structure related algorithms, e.g. pure **brutal force** (3.1.1), **depth first search** based (3.1.2), and **dancing links** based methods (3.1.3). Also, we make certain trials in the heuristic methods like **genetic algorithm** to solve Sudoku (3.1.4).

2.2 Diagnose Techniques

The diagnose of a given puzzle contains 3 steps: 1) if a sudoku is valid; 2) if valid, whether it can be solved; and 3) if solvable, whether it is well-defined.
We proposed **rule-based validation check** (3.2.1), and defined **solver-based diagnose techniques** to evaluate if a puzzle has solution (3.2.2), and the exact number of solutions (3.2.3).

2.3 Generating Algorithms

Sudoku grids are limited online, and we are looking forward to make our own well-designed puzzles based on solvers and diagnose techniques. Intuitively, key steps include 1) create a complete grid; 2) keep removing cubes under rules in ***1.2***, and stop if the grid has enough blanks.
We firstly **generate the complete grid based on Random DFS** seeds (3.3.1), and then **improve the naïve digging algorithms via DLX** (3.3.2).

2.4 Quantitative Difficulty Evaluation Metrics

Though many statistics can depict certain characteristics of given puzzles, but they can not necessarily evaluate their difficulties. We reasonably propose **Average Blanks' Candidates (ABC) as our metric** in (3.4).

3. Implementation Details

3.1 Solving Algorithms

1) Pure Brutal

The first solver to consider is the pure brutal search algorithm, which untiredly search possible guesses in candidate spaces for all blanks, and will not stop before find the solution, as in ***Alg 3.1***.

```
Algorithm 1: Pure Brutal Force Search
  Input: Sudoku_Grid, Target_Cubes, Candidates_for_Cubes
  Output: Completed_Grid
products := generate all targets' combinations of candidate for prod in products do
    if prod is a solution to Sudoku_Grid then
        Completed_Grid := Fill prod into Sudoku_Grid RETURN Completed_Grid
RETURN None
final ;
return Recommendlist and Evaluation;
```

Alg 3.1 Pure Brutal Force Search

Regarding advantage, it is intuitive and easy enough, especially with *itertools.product* in python,. However, its **time complexity is unacceptable**, increasing geometrically as the blanks go up. (e.g. search 15*10^12 times when blanks=35 with average candidate numbers as 2.38).

2) Depth First Search based method (DFS)

We also build DFS-based algorithm to tackle the puzzle by traversing all candidate, and dynamically retrieving possible guesses as far as possible. It will backtrack until find the solution.

i) Classical DFS in SUDOKU

Firstly, we deployed a classical DFS algorithms as in ***Alg 3.2***, where we directly go over all blanks by their natural order in the Sudoku grid.

```
Algorithm 1: Depth_First_Search
  Input: Target_ix
if Target_ix == 81 then
    return True;
if Sudoku_Grid[Target_ix] is blank then
    for candidate in Get_Candidates(Sudoku_Grid[Target_ix]) do
        Fill(Sudoku_Grid[Target_ix], candidate) if Deep_First_Search(Target_ix+1) then
            return True;
        else
            Fill(Sudoku_Grid[Target_ix], 0)
else
    if Deep_First_Search(Target_ix+1) then
        return True;
return False;
```

Alg 3.2 Depth First Search

DFS with dynamic candidate searching is comparatively more efficient than brutal search, since it **keeps shrinking the candidate space at each step**. However, since the searching frequency is quite high, some optimization must be made.

The first hypothesis we made is **whether the searching order affects the speed?** Say, if we search from the blanks with candidates from the lowest to the highest, will the searching speed increase? In fact, this modification merely changes the order without any pruning or computation optimization. Hence, the expected time complexity will not be affected much.

ii) Boot with Bit Computation

Another trial is to change the way of computation. As we have learned in database, the bitmap indexing is the fastest indexing, since the CPU computes super-fast for Boolean computation by nature. So, **we can do the computation in bit way!**

We build binary representation on each row, column, or box, for example, the vector (011000100) indicates 3, 7, 8 has already appeared. Moreover, further accelerates can be achieved via tricks as:

i. **Searching the candidate set via bitwise inversion and bitwise & operation with (111111111)**. For example, the inverse of the above vector (100111011) indicates that 1,2,4,5,6,9 can be chosen as candidate digits.

ii. **Turning the bit i of the number b from 1 to 0 or from 0 to 1 via b ^ (1<<i)** where '^' means XOR operation and '<<' means left shift operation.

iii. **Getting the position of the lowest bit** whose value is 1 in the binary representation of number b via 'bin(b & (-b)).count("0") – 1'.

3) Dancing Links (DLX)

i) Exact Cover Problem

The exact cover problem is a decision problem to find an exact cover. Given a set X and another set where each element is a subset to X. We want to select a set of subsets such that every element in X exist in exactly one of the selected sets, for example:

$$S_1 = \{3, 9, 17\}$$
$$S_2 = \{1, 9, 17\}$$

$$S_3 = \{1, 6, 119\}$$
$$S_4 = \{6, 9, 119\}$$
$$X = \{1, 3, 6, 9, 17, 119\}$$

Then (S_1, S_3) is a set of valid solutions. This problem can be visualized as a binary matrix M and the goal is to find a set of rows such that every column in the rows contain exactly one 1.

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

ii) Algorithm X

Algorithm X is a nondeterministic, recursive algorithm that uses DFS for backtracking, designed by Donald Knuth, and widely applied to find all solutions to the exact cover problem.

Before handling this algorithm, let me introduce some simple preliminaries. Given a node x that points to two elements in a doubly linked list. L[x] points to the left element of x and R[x] points to the right element of x. The following shows how to remove x from the doubly linked list and how to insert x back into the doubly linked list.

$$L[R[x]] \leftarrow L[x], R[L[x]] \leftarrow R[x]$$
$$L[R[x]] \leftarrow x, R[L[x]] \leftarrow x$$

The first operation is well-known, but the second operation is not well-known enough. The insertion works though the left and right elements of x cannot refer to it, x still has the references it had. The most notable usage of insertion is for backtracking.

iii) Dancing Links

Dancing Links, or DLX for short, is the technique for implementing Algorithm X efficiently. Given a binary matrix, DLX will represent the 1s as nodes, with each node has four member variables left, right, up, and down, which allow linkage to the 4 directions.

Another member variable in node link to the column node which is a special node that has one additional member variable, count of row, which represents the number of rows in this column. Since each row and column are a circular doubly linked list, if a node is removed from the column, the count of row is decreased. ***Fig 3.1*** shows how to represent a binary matrix in DLX structure.

$$\begin{array}{c} \begin{array}{ccccccc} A & B & C & D & E & F & G \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

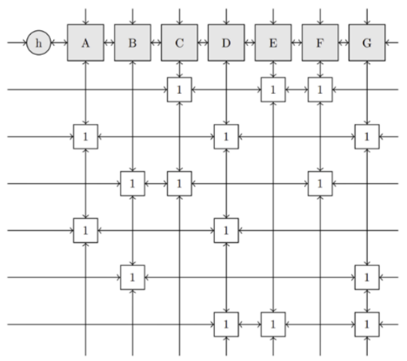


Fig 3.1 Binary matrix in DLX manner

iv) Reducing Sudoku to Exact Cover Problem

What is the meaning of rows and columns of the binary matrix visualized from exact cover problem? In fact, rows represent decisions because each row corresponds to a set, which includes choose or not choose. Columns represent constraints, every rule of Sudoku can be described as a constraint.

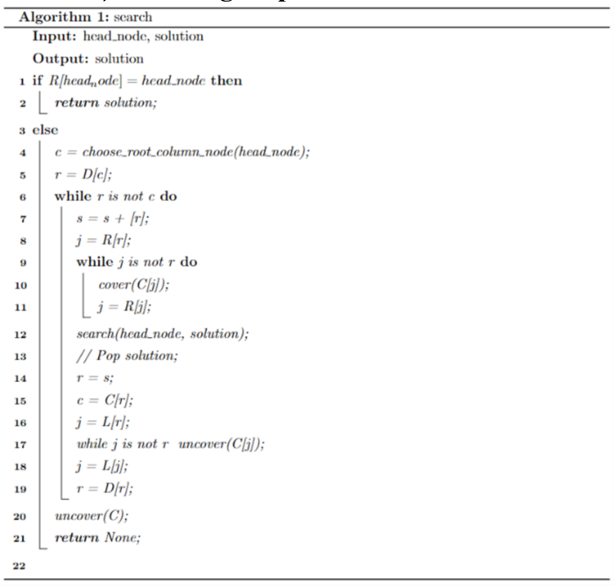
Consider a Sudoku, each decision can be represented by an ordered tuple (r, c, w). r is row r, c

is column c and w is integer w. Therefore, there are $9 \times 9 \times 9 = 729$ rows. Then consider the constraints. Let's think about what impact will the decision (r, c, w) have. Note that (r, c) locates in box b.

1. Row r includes an integer w (indicated by $9 \times 9 = 81$ columns)
2. Column c includes an integer w (indicated by $9 \times 9 = 81$ columns)
3. Box b includes an integer w (indicated by $9 \times 9 = 81$ columns)
4. Fill in an integer in (r, c) (indicated by $9 \times 9 = 81$ columns)

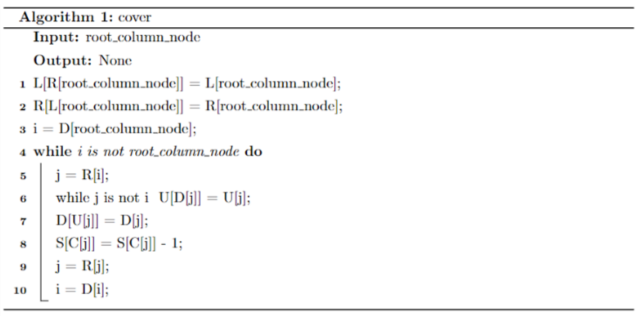
There are 81×4=324 columns. So far, we have successfully reduced a Sudoku to an exact cover problem.

v) Solving Steps



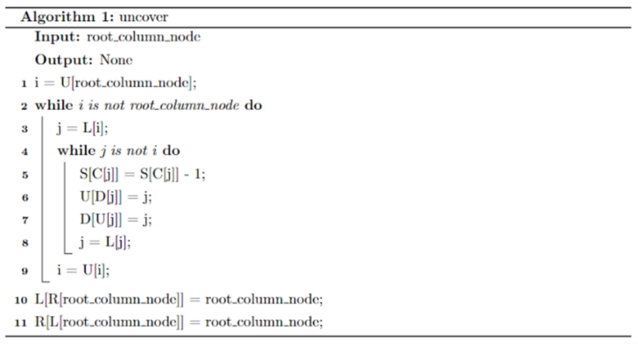
```
Algorithm 1: search
Input: head_node, solution
Output: solution
1  if $R[head_node] = head\_node$ then
2      return solution;
3  else
4      $c$ = choose_root_column_node(head_node);
5      $r = D[c]$;
6      while $r$ is not $c$ do
7          $s = s + [r]$;
8          $j = R[r]$;
9          while $j$ is not $r$ do
10             cover($C[j]$);
11             $j = R[j]$;
12         search(head_node, solution);
13         // Pop solution;
14         $r = s$;
15         $c = C[r]$;
16         $j = L[r]$;
17         while $j$ is not $r$  uncover($C[j]$);
18         $j = L[j]$;
19         $r = D[r]$;
20     uncover($C$);
21     return None;
22
```

Alg 3.4 search in DLX



```
Algorithm 1: cover
Input: root_column_node
Output: None
1  L[R[root_column_node]] = L[root_column_node];
2  R[L[root_column_node]] = R[root_column_node];
3  i = D[root_column_node];
4  while i is not root_column_node do
5      j = R[i];
6      while j is not i  U[D[j]] = U[j];
7      D[U[j]] = D[j];
8      S[C[j]] = S[C[j]] - 1;
9      j = R[j];
10     i = D[i];
```

Alg 3.5 cover in DLX



```
Algorithm 1: uncover
Input: root_column_node
Output: None
1  i = U[root_column_node];
2  while i is not root_column_node do
3      j = L[i];
4      while j is not i do
5          S[C[j]] = S[C[j]] - 1;
6          U[D[j]] = j;
7          D[U[j]] = j;
8          j = L[j];
9      i = U[i];
10 L[R[root_column_node]] = root_column_node;
11 R[L[root_column_node]] = root_column_node;
```

Alg 3.6 uncover in DLX

We firstly define the function search. $L[x]$ points to the left element of x. $R[x]$ points to the right element of x. $U[x]$ points to the up element of x. $D[x]$ points to the down element of x. $C[x]$ represents current node's root column node. $S[x]$ represents the count of row of x.

On row 4 in ***Alg 3.4***, the column node is chosen which can be implemented in two different ways, either by choosing the first column object after the root column node, or by choosing the column with the fewest number of 1s occurring in a column. I choose the latter since it can minimize the branching factor. Another two functions *cover* and *uncover* in ***Alg 3.5*** and ***Alg 3.6*** make use of the removal and insertion introduced before along with Algorithm X.

DLX is invoked with *search(head_node, [])*, since *head_node* is not equal to *R[head_node]*, still in the example in Fig 3.1, column A is chosen as the next column to be covered. We know that r points to the first node in column A. This will also affect column D and column G since these two columns also have node on this row. Therefore, column A, column D and column G will be covered in this step, as in ***Fig 3.1***.

Next, the branch uses *search(head_node, solution)* to search after all nodes that were covered in the first row in A. This will cover column B and leave no node in column E, which in turn will leave *search(head_node, solution)* without solution. So the algorithm will return and proceeds to the first row in column A. Finally, the solution is found in this dancing links' manner.

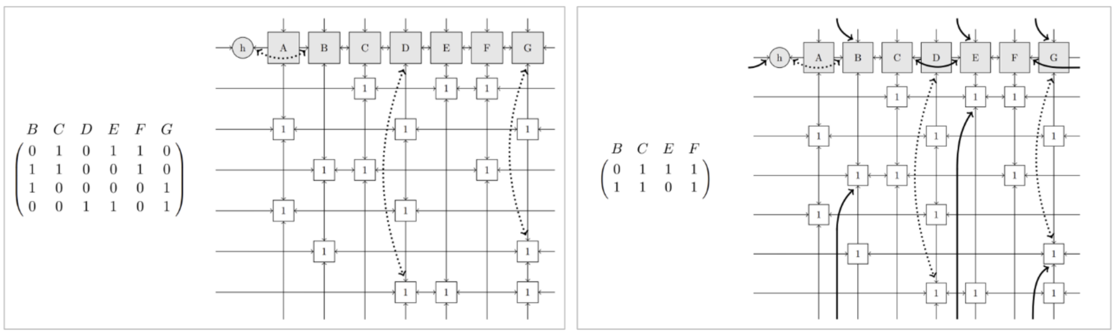


Fig 3.2 Quick Example of DLX
(continued from Fig 3.1)

4) Genetic Algorithm

Besides data structure related algorithms, we also tested heuristic methods. The genetic algorithm mimics the process of natural selection, and try to find the fittest guess. It involves generating individuals (guesses) up to certain size to form the population, randomly picking parents, and producing their offspring (mixing guesses), and filter the whole population for next iteration (randomly by evaluated scores). Among this process, mutations while intersection is considered to produce more possibilities. To be specific:

i. We randomly generate individuals by randomly picking candidate for all targets, as in ***Alg 3.7***, We create the initial population by repeating generating individuals up to the population size.

```
Algorithm 1: Random_Init_Individual
   Input: Target_ix, Target_length
   Output: Guess
 1 Individual := ''
 2 for ix in Target_length do
 3     Individual += Randomly_Pick(Get_Candidates(Sudoku_Grid[Target_ix]))
 4 return Individual;
```

Alg 3.7 Randomly generate individual in GA

ii. With the population, we rate individual based on the fitness to the Sudoku. That is, the guess's repeated digits among its peers, as in ***Alg 3.8***. The lower the score, the more fit to the puzzle.

```
Algorithm 1: Evaluate_Individual
   Input: Guess, Target_ix
   Output: Score
 1 score = 0;
 2 new_grid := Fill_Guess(Sudoku_Grid, Guess);
 3 for ix in Target_ix do
 4     score += Count repeated digits with ix ;
 5 return score/2;
```

Alg 3.8 Individual Evaluation

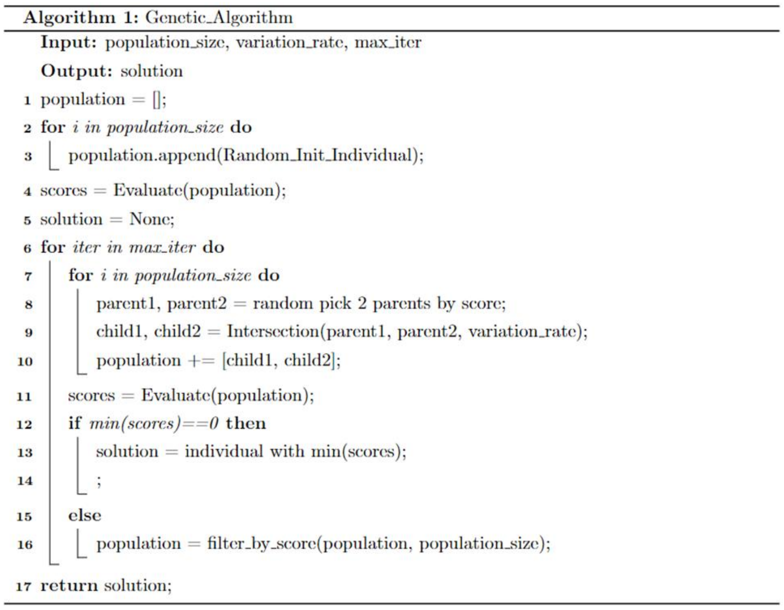
iii. Then, parents are randomly picked by scores for intersection, indicating that those fit more will have priority of intersection. Variants add new guesses among the intersection, as in ***Alg 3.9***.

```
Algorithm 1: Intersection
   Input: parent1, parent2, variation_rate
   Output: child1, child2
 1 r1 = random digit between (0,1);
 2 r2 = random digit between (0,1) & greater than r1;
 3 child1 = parent1[:r1] + variation(parent2[r1:r2], variation_rate) + parent1[r2:];
 4 child2 = parent2[:r1] + variation(parent1[r1:r2], variation_rate) + parent2[r2:];
 5 return child1, child2;
```

Alg 3.9 Intersection of Parents

iv. Afterwards, we rate all individuals' scores again, and filter the whole population by their scores, such that guesses higher fitness will have higher possibility to be passed to the next generation. Note that we applied the Alias Sampling to accelerate the weighted sampling process, which can reduce the time complexity to O(1) while drawing choice.

The whole process of Genetic Algorithm is available in ***Alg***.



```
Algorithm 1: Genetic_Algorithm
  Input: population_size, variation_rate, max_iter
  Output: solution
population = [];
for i in population_size do
    population.append(Random_Init_Individual);
scores = Evaluate(population);
solution = None;
for iter in max_iter do
    for i in population_size do
        parent1, parent2 = random pick 2 parents by score;
        child1, child2 = Intersection(parent1, parent2, variation_rate);
        population += [child1, child2];
    scores = Evaluate(population);
    if min(scores)==0 then
        solution = individual with min(scores);
        ;
    else
        population = filter_by_score(population, population_size);
return solution;
```

Alg 3.10 Whole Process of GA

3.2 Diagnose Techniques

1) Valid

First step is check the digits, shape of puzzles based on rule. It is the very beginning for filtering invalid Sudoku quizzes from various sources, like invalid parsing result from online sudoku quizzes.

2) Solvable

To judge whether a sudoku is solvable, we can easily apply solvers on the puzzle to see if the solver can return with a solution. This judgement greatly relies on the solver we built in 3.1. For example, we implemented DLX and DFS solver core to test the solvability.

3) Solution Uniqueness and Multi-Solution Test

As mentioned in 3.2 and 3.3, the DFS and DLX core immediately stop if they find the solution. In order to find all possible solutions, we can easily change logic by extending the constraint on stop flag, so that they can go over all the feasible solutions.

The extended algorithm can see if the puzzle has multiple solutions, even with the exact number.

3.3 Generating Algorithms

1) Competed Matrix Generation

Before generating a Sudoku puzzle, we should have a complete grid with no blanks.

To achieve this, we can modify a solver to solve the grid with 9x9 blanks, with random picking candidates and filling into each space. **The wisdom behind this is that we think generate a complete grid equals to find one solution on the given grid**! As a result, we design Random DFS picking candidate randomly to fill the spaces., as shown in the first two columns in ***Fig 3.3***. Not only that, this architecture by nature can generate complete grid with zero-seed, or with some given seeds. The two rows in ***Fig 3.3*** demonstrated these 2 kinds of ideas.

2) Digging Holes:

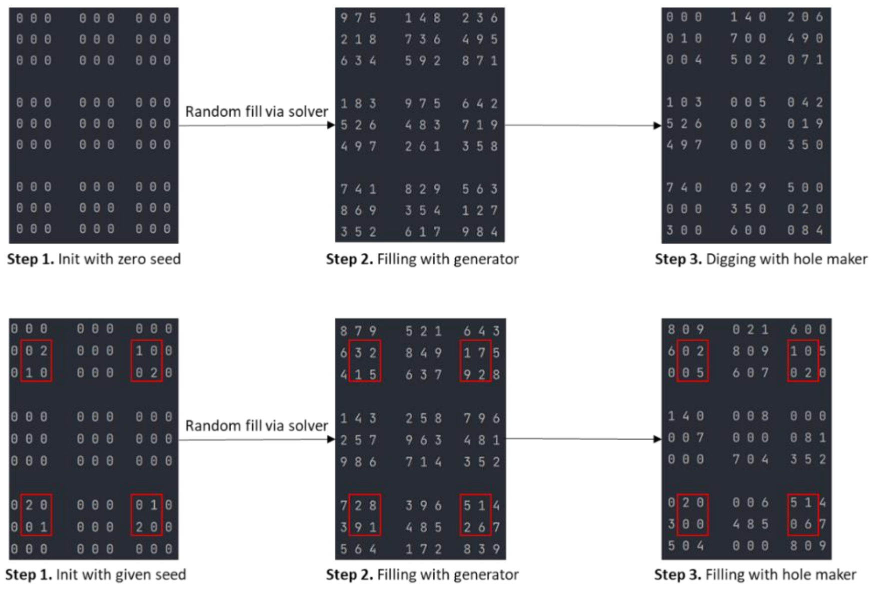

Random fill via solver
Step 1. Init with zero seed
Step 2. Filling with generator
Step 3. Digging with hole maker
Random fill via solver
Step 1. Init with given seed
Step 2. Filling with generator
Step 3. Filling with hole maker

Fig 3.3 Process of the Generating Algorithm

The only target for the hole digger is the number of blanks of the result puzzle, with the constraint in ***1.2***. We introduce two algorithms as follows.

i) Naïve Digging Algorithm

The first hole digging algorithm is quite naïve. With the processes as:

i. Mark every cube as not visited;
ii. Randomly visit unvisited cubes, and try to remove the digits;
iii. Fill the rest 8 digits besides the removed one, and see if it has solution;
iv. If it has, rollback and mark the cube as visited, since that removal violates the restrictions;
v. Otherwise, keep that removal, mark that cube as visited and repeat steps ii.-iv.
vi. The algorithm stops if all cubes are marked as visited.

This naïve digging is prevailing, nearly all websites suggest us to do so. However, its disadvantage is obvious: **it's time consuming and cannot always find the puzzle**, since it repeats to find solution 8 times before moving to the next step. The digging time required to get a well-defined puzzle increases exponentially after 58 spaces.

ii) Boost with DLX core

The 8-time trialing of the naïve digging method is unacceptable, but the question is, how can we dig wholes while testing if it will cause multiple solutions? Remind that, we have made some progress in 3.2.3 to test certain properties through solver cores, these **methods for solution uniqueness test are exactly what we want for hole digger** method!

By merging the DLX for multi-solution test into the generation process, we get a 5-time faster hole digger algorithm compare with the naïve one, with detailed experiments in ***4.2***.

3.4 Difficulty Evaluation Metrics

Many statistics can depict characteristics of the given puzzle. For example, some may think that the number of blanks might have relationship with difficulty. Similar metrics are the largest search size in brute force, indicating the probability of acquiring a solution via random guess. We think they just evaluate the Sudoku grid from very limited viewpoints compared with ours.

In order **to quantify the difficulty reasonably yet intuitively, we propose our metric as the**

Average Blanks' Candidates (ABC), which can be computed as the average number of candidate numbers for all blanks in a given puzzle.

To begin with, when tackling a Sudoku as a player, the nervousness and doubt originates from the uncertainty while filling the blank cubes, because we have many choices, but only one is correct, and each trial will exhaust us a lot in the end! Imagine a different story, if we have blank cubes with only few candidates, we can confidently fill them in and make some concrete guesses! Hence, the **multi-choices property should be captured in metric design, and ABC directly reflects this important property in an intuitive yet effective manner**.

Besides, ABC is comprehensively reasonable compared with other peers. As for the number of blank cubes, it does not necessarily relate to the difficulty of the sudoku. Also, while randomly generated sudoku with spaces at given scale, the average candidate does not diverse significantly in the first 30 spaces, and difficulties resemble within this range, as in ***Fig 3.3***. To our experience, the simplest methodology naked single can be recursively used if the digging strategy is kind enough. Hence, number of blanks is not a good metric compared with ABC.

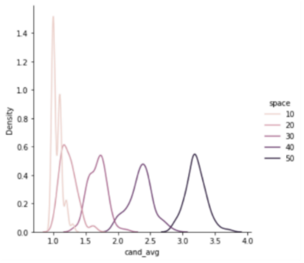


Fig 3.4 Distribution of ABC by number of spaces generated by DLX

On the other hand, as for the maximum search space, it is too big to form a cognition, and the increasing trend is too big to scale, so that it is not cognitively suitable in this case.

Frankly speaking, although there exist more scientific ways to evaluate the difficulty of a given Sudoku grid, for example, comprehensively rating on the most difficult necessary tactic used in solving Sudoku. However, for all human tactics, HODOKU mentioned more than 58 kinds, and we think it is not a economic enough choice as its too difficult to realize them all within the limited time, yet achieve limited further progress,

3.5 Methodology Organization

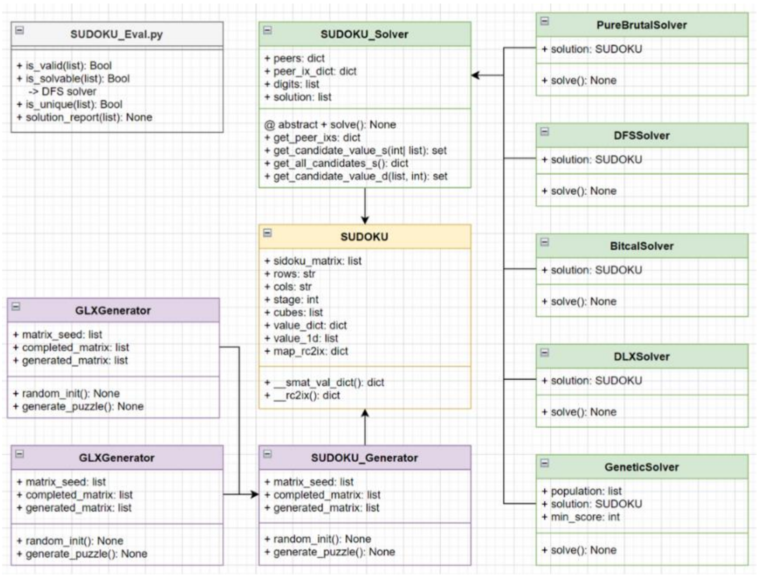


Fig 3.5 Design of the Class Relation of the sudoku_toolkits

For all the techniques mentioned, we realized them as a Sudoku solving module *'sudoku_toolkits'*, with reasonable class structure. ***Fig 3.3*** shows the detailed relation of all the classes, with classes in yellow as our base class for Sudoku demonstration and retrieving, green ones as our solver, purple ones as generator and grey ones as the evaluator.

4. Experiments

4.1 Solving Efficiency Evaluation

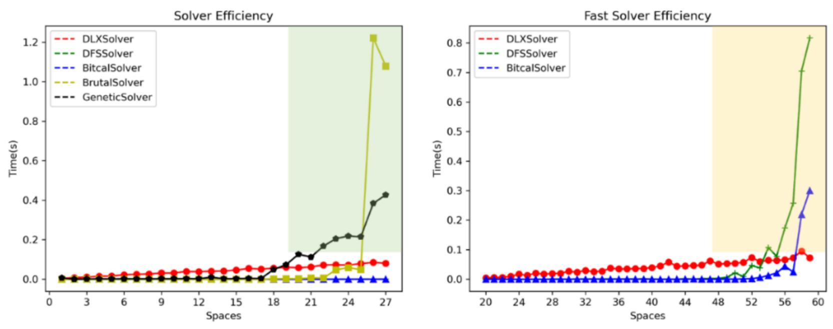


Fig 4.1 Solving efficiency of all proposed models on randomly generated Sudoku grids
We remove brutal and genetic solvers after 30 spaces due to low efficiency

To evaluate the efficiencies, we randomly generate 10 well-defined puzzles for each space from 1-60, and repeat the solving for each solver for 10 times to avoid random disturbance of CPU. By counting the average solving time in second as our metric, as in ***Fig 4.1***, we find that:

- Brutal search and genetic algorithm perform well in puzzle with spaces under 17, but sudden deteriorates afterwards, and increases exponentially, so we exclude them for further tests.
- DFS-based solver is efficient for puzzles under 53 spaces, and the bit computation can indeed accelerate speed by 3 times on average. However, there solving time suddenly increase after 56.
- DLX solver is the most stable one under all circumstances, with almost linear complexity as the number of spaces increases, which is one of the reasons why we pick it as the main solver for generator and diagnose core.

4.2 Generation Efficiency Evaluation

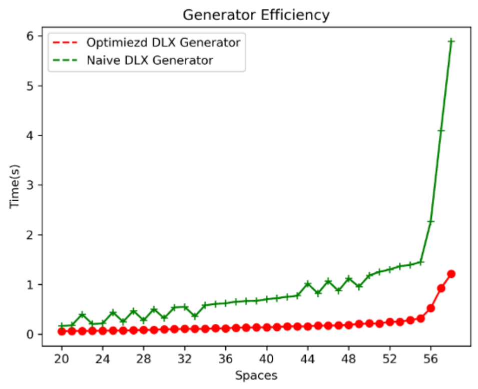


Fig 4.2 Generation Efficiency between Naïve and DLX Generator

As mentioned in 3.3.2, we developed a brand-new generator by diagnose the solution uniqueness of the grid via DLX multi-solution test, instead of solving Sudoku with the other 8 candidate digits. We compared the two generators' efficiency by counting the average time for generating 100 grids with spaces from 20 to 58. The ***Fig 4.2*** denotes the results, where the optimized dlx generator is 5 times faster than the naïve one on average. In this case, we can confidently say that our generator is more efficient than the prevailing one.

4.3 Difficulty Evaluation

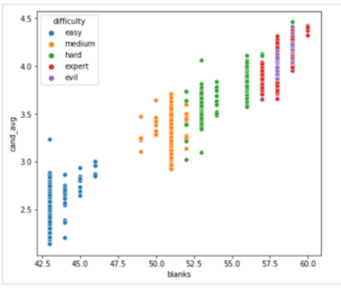


Fig 4.3 Distribution of ABC by spaces

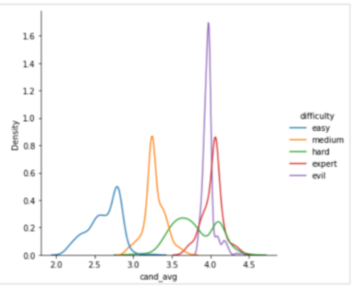


Fig 4.4 Distribution Density Plot of ABC

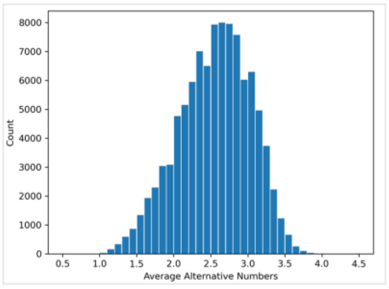


Fig 4.5 Distribution Histogram of ABC

Note:
1. *Fig 4.3 and Fig 4.4 are puzzles scraped and parsed from images in sudoku.com, with sample size as 1500.*
2. *Fig 4.5 is generated from puzzles in newspapers stored in Kaggle with sample size as 100000.*

To further evaluate our metric ABC, we scraped 1500+ Sudoku grids' image from sudoku.com, one of the most popular Sudoku App, and parse them into numbers via OCR, and downloaded 100000+ Sudoku grids published in newspaper on Kaggle. Based on ABC, we found that:

i. Two different media design sudoku for different perspectives. Since newspapers' audience are the general public, the overall difficulty distribution in ***Fig 4.5*** is easier than the puzzles designed for online sudoku players in ***Fig 4.4***. The latter's customers usually get in touch by either searching or recommendation, both reflects certain degree of proficiencies.

ii. For difficulty design in sudoku.com, although easy and medium grids have large intervals about blank numbers, their ABC designs are more considerable. But the design for hard, expert and evil shows little bit discrepancy; same pattern appears in the solving rate among these 3 difficulties, indicating their design for hardcore players may not be that good.

5. Conclusions

In this program, we have so far evaluated 5 solving algorithms' efficiency, and find that 1) DFS has the highest solving efficiency for puzzles with spaces under 53 spaces on average, but sudden increases later; and 2) DLX performs consistently stabler under all circumstances; and 3) Brutal and genetic algorithm perform is not suitable for puzzles with more than 25 spaces.

For diagnosis, we applied rule-based and solver-based methods checking if a sudoku is well defined. The proposed multi-solution diagnose algorithm can not only tells if the puzzle is well defined, but also tells how many solutions that puzzle have.

Regarding generation, we first create a compete grid with Random DFS, and further improved the naïve hole digging algorithm via DLX multi-solution diagnose by nearly 5 times faster.

We also formed an intuitive difficulty evaluation metric, Average Blanks' Candidates (ABC), which can be computed as the average number of candidate numbers for all blanks in a given puzzle. It turns out that this metric is reasonable compared with other proposed ones.

For detailed codes, you are warmly welcomed to our git-hub space:
https://github.com/georgeDuSH/sudoku_toolkit