



# SUDOKU SOLVER

## Study and Implementation

### Abstract

In this project we studied two algorithms to solve a Sudoku game and implemented a playable application for all major operating systems.

Qizhong Mao, Baiyi Tao, Arif Aziz  
{qzmao, tud46571, arif.aziz}@temple.edu

## Table of Contents

|   |           |
|---|-----------|
| <b>Introduction .....</b>                                       | <b>3</b>  |
| <b>Variants .....</b>   | <b>3</b>  |
| Standard Sudoku .....   | 3         |
| Small Sudoku .....  | 4         |
| Mini Sudoku .....   | 4         |
| Jigsaw Sudoku .....   | 4         |
| Hyper Sudoku .....  | 5         |
| Dodeka Sudoku.....  | 5         |
| Number Place Challenger .....                                   | 5         |
| Giant Sudoku .....  | 6         |
| Killer Sudoku (Not Supported) .....                             | 6         |
| Greater Than Sudoku (Not Supported).....                        | 7         |
| <b>Sudoku Solving Algorithm.....</b>                            | <b>7</b>  |
| Brute Force Search.....   | 7         |
| Heuristic Search .....  | 8         |
| Filling Empty Grid.....   | 9         |
| Comparison between Brute Force Search and Heuristic Search..... | 9         |
| <b>Experiments .....</b>  | <b>10</b> |
| Filling Empty Grid (Worst Scenario) .....                       | 10        |
| Solving Standard Sudoku.....                                    | 10        |
| <b>Optimization .....</b>                                       | <b>11</b> |
| Concurrently Processing.....                                    | 11        |
| Stochastic Value Selection.....                                 | 11        |
| <b>Application.....</b>   | <b>11</b> |
| Screenshots .....   | 12        |
| Add a New Board Variant.....                                    | 14        |
| Create a Preset File .....                                      | 14        |
| <b>Summary.....</b>   | <b>15</b> |
| <b>References .....</b>   | <b>15</b> |

## Introduction

Sudoku (数独) is a logic-based, combinatorial number placement puzzle game. The objective of this game is to fill a  $N \times N$  grid with digits so that every row, column and inner box is filled with number from 1 to  $N$  without duplicates. The shape of inner boxes may vary, but all must have exactly  $N$  blocks. In most situations, a Sudoku game should have a unique solution. However, there can be cases that a Sudoku game has multiple solutions. Usually a game with multiple solution is considered to be more difficult than the one that has a unique solution.

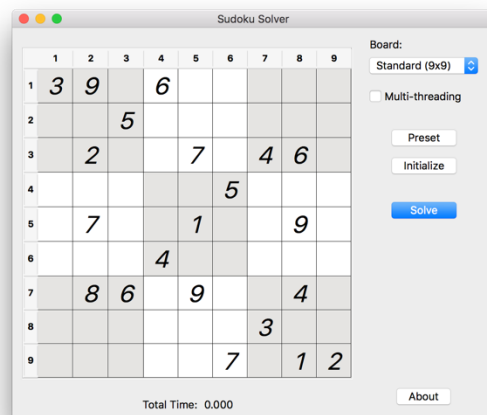
Sudoku was originated in the late 16<sup>th</sup> century. La France developed the embryonic form of the modern Sudoku of 9 by 9 grid with 9  $3 \times 3$  inner boxes, and later, it was believed that the modern Sudoku was mostly designed anonymously by Howard Garns in 1979. However, this game became a worldwide phenomenon not until Nikoli introduced it to Japan in 1984 and named it as 数字は独身に限る. The name was later abbreviated to 数独 (Sudoku) by Maki Kaji. Sudoku is a trademark in Japan and the puzzle is generally referred to as Number Place, or NumPla in short.

## Variants

Though the  $9 \times 9$  grid with  $3 \times 3$  inner boxes is by far the most common Sudoku game, many other variants exist. Our program natively implemented 8 variants including the standard one. In the following variants, different inner boxes are represented by different colors. In general, 2 colors are needed to mark all inner boxes. Also the number of inner boxes is  $N$  in a  $N \times N$  game for most variants, there can be exceptions.

### Standard Sudoku

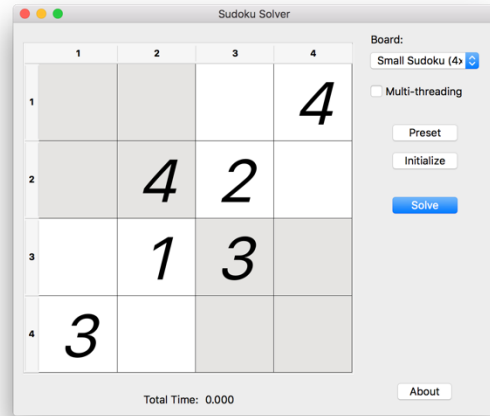
- Grid size:  $9 \times 9$
- Number of inner boxes: 9
- Inner box shape:  $3 \times 3$  Square
- Number range: 1 to 9



### Small Sudoku

Small Sudoku is probably the smallest and easiest Sudoku variant.

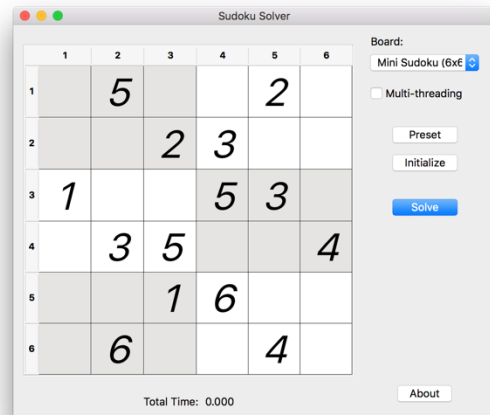
- Grid size:  $4 \times 4$
- Number of inner boxes: 4
- Inner box shape: Square
- Number range: 1 to 4



### Mini Sudoku

Mini Sudoku is a relatively easy variant with non-square inner boxes.

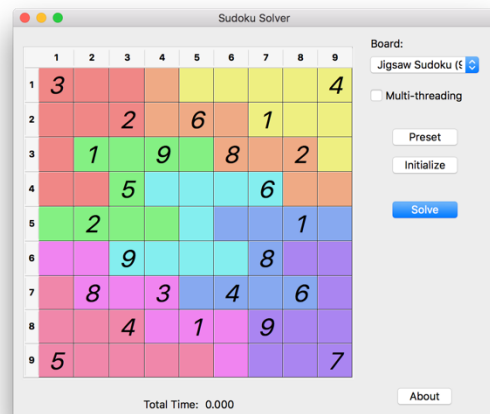
- Grid size:  $6 \times 6$
- Number of inner boxes: 6
- Inner box shape:  $3 \times 2$  Rectangle
- Number range: 1 to 6



### Jigsaw Sudoku

Jigsaw Sudoku (or Nonomino Sudoku) is almost the same as the standard Sudoku. The only difference to the standard Sudoku is the shape of the inner blocks. The difficulty is considered to be the same as the standard Sudoku. There are also several other variants like the Jigsaw Sudoku, where the inner boxes are not rectangles. These variants usually use multiple colors to distinguish different inner boxes.

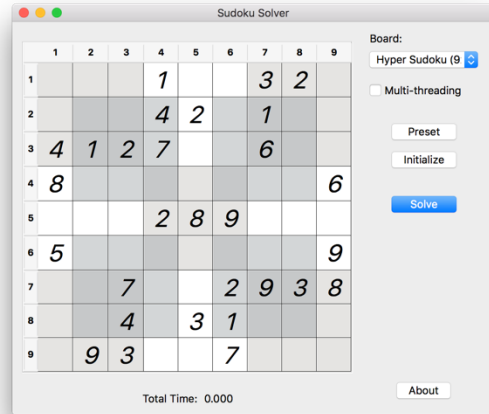
- Grid size:  $9 \times 9$
- Number of inner boxes: 9
- Inner box shape: Nonomino (polyomino of order 9)
- Number range: 1 to 9



### Hyper Sudoku

Hyper Sudoku is one of the most popular variant. It is very similar to the standard Sudoku, but with 4 extra interior squares. There are some overlaps, which can logically reduce the number of values for those blocks. In general, all inner boxes can be represented by 2 colors, where the overlapping blocks will use an average color of 2 colors.

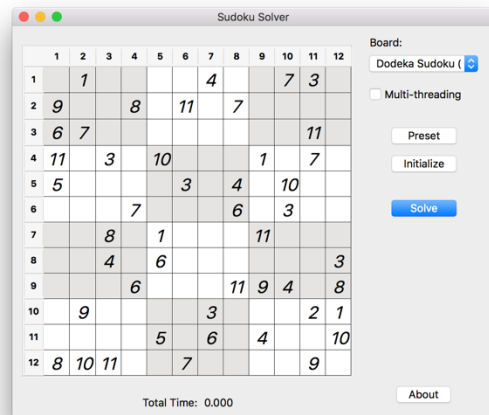
- Grid size:  $9 \times 9$
- Number of inner boxes: 13
- Inner box shape:  $3 \times 3$  Square
- Number range: 1 to 9



### Dodeka Sudoku

Dodeka Sudoku can be considered as a larger version of Mini Sudoku. As the maximum number allowed is greater than 9, alphabetic letter may be used instead of digits. For example, 10 can be replaced by *A*, 11 can be replaced by *B*, 12 can be replaced by *C*.

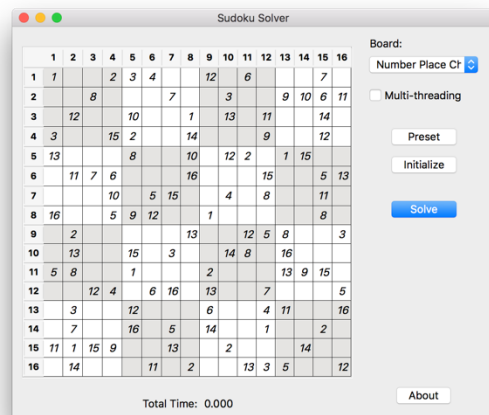
- Grid size:  $12 \times 12$
- Number of inner boxes: 12
- Inner box shape:  $4 \times 3$  Rectangle
- Number range: 1 to 12 or 1 to 9 and *A* to *C*



### Number Place Challenger

As the maximum number allowed is greater than 9, alphabetic letter may be used instead of digits. For example, 10 to 16 can be replaced by *A* to *G* respectively.

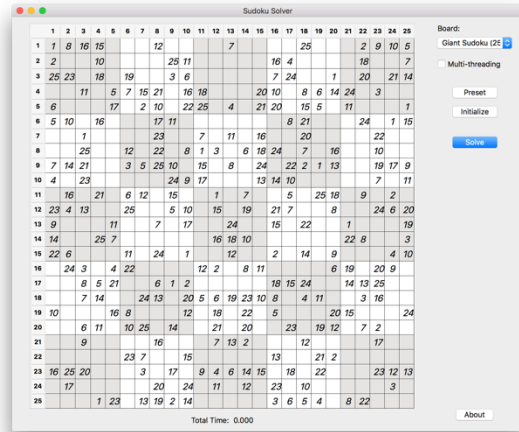
- Grid size:  $16 \times 16$
- Number of inner boxes: 16
- Inner box shape:  $4 \times 4$  Square
- Number range: 1 to 16 or 1 to 9 and *A* to *G*



### Giant Sudoku

As the maximum number allowed is greater than 9, alphabetic letter may be used instead of digits. For example, 10 to 25 can be replaced by *A* to *P* respectively.

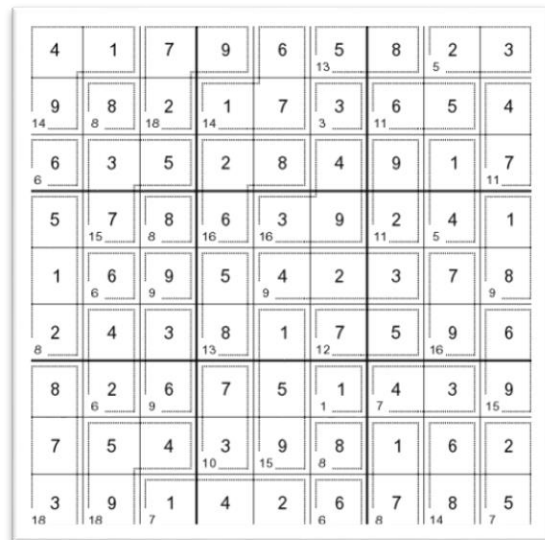
- Grid size: 16 × 16
- Number of inner boxes: 16
- Inner box shape: 4 × 4 Square
- Number range: 1 to 25 or 1 to 9 and *A* to *P*



### Killer Sudoku (Not Supported)

In this 9 × 9 Sudoku, with the Sudoku strategy, Kakuro strategy is also required. In the dotted lines, which is also called cages, contain a set of non-repeating digits. The sum of the numbers in the cages will give the number that is shown in the cage.

- Grid size: 9 × 9
- Number of inner boxes: 9
- Inner box shape: 9 × 9 Square
- Number range: 1 to 9



### Greater Than Sudoku (Not Supported)

In this  $9 \times 9$  Sudoku, a greater – than or a smaller – than sign will appear between any two blocks in the  $3 \times 3$  sub blocks, that indicates the relationship between the numbers in those blocks.

- Grid size:  $9 \times 9$
- Number of inner boxes: 9
- Inner box shape:  $9 \times 9$  Square
- Number range: 1 to 9

|           |           |           |
|-----------|-----------|-----------|
| 2 < 6 < 7 | 1 < 3 < 9 | 8 > 5 > 4 |
| 1 < 9 > 3 | 5 > 4 < 8 | 7 > 6 > 2 |
| 8 > 5 > 4 | 6 < 7 > 2 | 3 < 9 > 1 |
| 9 > 3 < 8 | 2 > 1 < 5 | 6 > 4 < 7 |
| 6 > 2 > 1 | 7 < 9 > 4 | 5 < 3 < 8 |
| 4 < 7 > 5 | 3 < 8 > 6 | 2 < 1 < 9 |
| 5 < 1 < 9 | 8 > 2 < 3 | 4 < 7 > 6 |
| 7 < 8 > 6 | 4 < 5 > 1 | 9 > 2 < 3 |
| 3 < 4 < 2 | 9 > 6 < 7 | 1 < 8 > 5 |

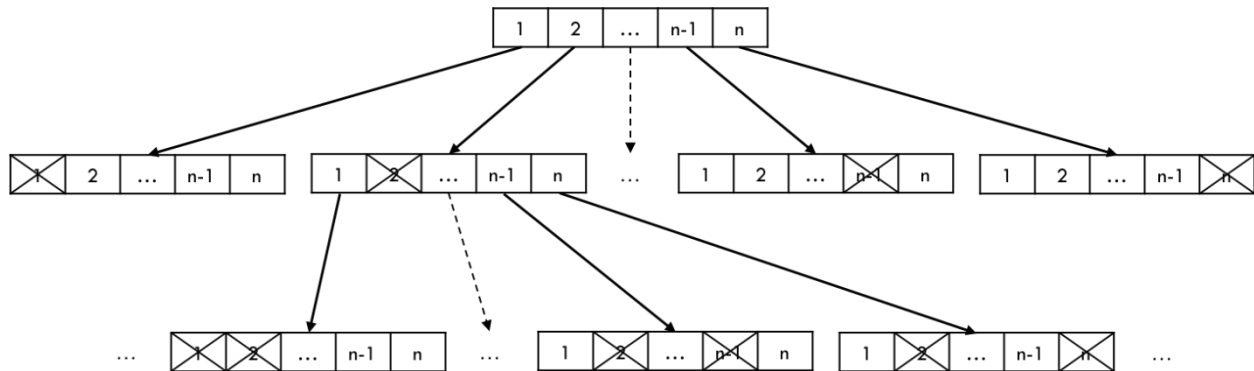
## Sudoku Solving Algorithm

### Brute Force Search

The brute force search algorithm is probably the most fundamental method to solve any type of puzzle game. To solve a Sudoku game by brute force search, the following steps are performed:

1. Find the first block that is empty
2. List all values 1 to  $N$  for that block
3. Check all the other blocks in the **row** that contains this block, remove any used values
4. Check all the other blocks in the **column** that contains this block, remove any used values
5. Check all the other blocks in the **inner boxes** that contains this block, remove any used values
6. If there is only one value remained, assign the value to the block, then go back to step 7. If there are multiple remaining values, choose one of the values and assign to it, and keep a record of which value is chosen for the block and all the remaining values for that block, then go to step 7. If there is no value remained, go to step 8.
7. Find the next empty block, and repeat the above steps. If no empty block can be found, then the game is solved.
8. If in any previous steps, there is one block that has been assigned a value from a list, choose the last block that satisfy the condition, assign another value from the remaining list and remove all the values assigned after the block, then repeat the above steps. If all remaining values have been tested, choose second last block that has multiple remaining values, and test another value, until a solution is found, or no solution if all blocks having multiple remaining values are checked.

Based on the explanations above, we can observe that the brute force search is to build a tree that with maximum height of  $N^2$ , as every block needs to be checked. A abstract tree structure is shown below (each level is a block).



However, if we look the game row by row, for each row, we have  $N$  possible values for the first block, and  $N - 1$  possible values for the second block,  $N - 2$  possible values for the third block, and 1 value for the last block. Hence the computational complexity for filling one row is  $O(N \times (N - 1) \times \dots \times 1) = O(N!)$ . As there are  $N$  rows, the total computational complexity is  $O((N!)^2)$ . In fact, this can be further reduced, but it is hard to find a short formula to represent it, so we will just keep this as the final computational complexity of brute force search.

### Heuristic Search

An alternative way to solve a Sudoku game is heuristic search. In every step, we try to find a block that has least possible values. The detailed steps are given as follows:

1. Scan all empty blocks, for every empty blocks, maintain a list of possible values by checking the row, column and inner boxes that contain the block.
2. Find a block that has least possible values.
3. If the block has only one possible value, assign that value to the block, and repeat from step 1. If the block has multiple possible values, assign one value from the list, and keep a record of which value has been assigned and all the other remaining values, then repeat from step 1. If the block has no possible value, go to step 4.
4. If there is one block that has multiple possible values, assign another value in the list and remove all the values after the block. If all values have been tested, check the second last block that has multiple possible values, until a solution is found, or no solution if all blocks having multiple possible values are checked.

It is easy to see the steps of heuristic search share some commons with brute force search. The key difference is that heuristic search tries to minimize the cost of testing possible values by introducing more tests on all empty blocks. The introduced cost may be ignored because it is marginal. Therefore, if every empty block has exactly 0 or 1 possible value, it is the best scenario with computational complexity  $O(N^2)$  as every block needs to be checked once only. But the



worst scenario where all blocks are empty will have computational complexity the same as brute force search, which is  $O((N!)^2)$ .

### Filling Empty Grid

There is a special algorithm that can fill an empty board quickly with computational complexity  $O(N^2)$ , if the game satisfy the following conditions:

- The grid must be completely empty
- The width and height of the grid must be  $N = n^2$
- There must be exactly  $N$  inner boxes
- Every inner box must be a  $n \times n$  square
- No overlapping blocks

The pseudocode of this algorithm is shown on the right (we did not implement this algorithm in our program, since this algorithm is too restricted, which is not very useful for our purpose.

```
final int n = 3;
final int[][] field = new int[N][ N];
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    field[i][j] = (i* n + i/ n + j) % N + 1;
```

### Comparison between Brute Force Search and Heuristic Search

The comparison is shown in the following table

| $N$ | Complexity ( $c = 1$ ) | Complexity ( $c = 3$ ) | Complexity $O((N!)^N)$ |
|-----|------------------------|------------------------|------------------------|
| 4   | 16                     | $1.30 \times 10^3$     | $3.32 \times 10^5$     |
| 6   | 36                     | $4.67 \times 10^4$     | $1.39 \times 10^{17}$  |
| 9   | 81                     | $1.01 \times 10^7$     | $1.09 \times 10^{50}$  |
| 12  | 144                    | $2.18 \times 10^9$     | $1.46 \times 10^{104}$ |
| 16  | 256                    | $2.82 \times 10^{12}$  | $1.35 \times 10^{213}$ |
| 25  | 625                    | $2.84 \times 10^{19}$  | $5.84 \times 10^{629}$ |

Where  $c$  is the average number of possible values for every empty block. Hence, the second column in the table is the best scenario of heuristic search, the third column is a heuristic search with 3 possible values for every empty block on average, and the last column is the brute force search, where  $c$  can be consider to be  $N$ . Obviously a larger  $c$  will significantly increase the computational complexity in an order of magnitude.

However, brute force search has an advantage that it will always take constant time to solve all games of one variant, though the time can be extremely long.

## Experiments

### Filling Empty Grid (Worst Scenario)

We have tested the time to solve an empty grid of different grid sizes. We can see that the time for small and mini Sudoku are quite short, but the ratio does not correspond to the ratio calculated by the computational complexity. This may be caused by the overhead of internal functions and CPU scheduling. But the time to fill an empty standard grid took a whole day to run and not yet finished. Larger grid size causes an exponential grow in the complexity. Larger grid size such as  $12 \times 12$  and even larger variants should have much longer time to complete.

|      | Small        | Mini         | Standard     |
|------|--------------|--------------|--------------|
| Size | $4 \times 4$ | $6 \times 6$ | $9 \times 9$ |
| Time | 5 ms         | 12 ms        | $\infty$     |

### Solving Standard Sudoku

As the algorithm to solve all supported Sudoku variants is the same, we mainly test the time, CPU and memory usage for 200 preset values for standard Sudoku. Among the 200 presets, 100 are easy level and 100 are hard level. The presets are crawled from the Internet, but the difficulty levels are not quite uniform. We found that there could be some relatively hard presets in the easy level set, and there are also some extremely hard presets in the hard level set.

|            | Time     | CPU | Memory |
|------------|----------|-----|--------|
| Easy Level | 815 ms   | 5%  | 100 MB |
| Hard Level | 6,189 ms | 17% | 1.5 GB |

Since easy levels generally have a smaller  $c$  value, the time needed for solving an easy level standard Sudoku is far more less than solving a hard level. The CPU and memory usage are not very precise, but the CPU usage for easy levels is usually below 8%, and above 13% for hard levels. Also the maximum memory usage we recorded for easy levels is less than 100 megabytes, but for hard levels, the memory usage can easily go beyond 1.5 gigabytes, sometimes even 10 gigabytes.

## Optimization

Exponential grow is a natural of Sudoku, which means there is no way that can really reduce the complexity to solve a badly designed game. However, there are still 2 methods may be used.

### Concurrently Processing

If a game has multiple solutions, usually these valid solutions and any other invalid solutions are independent with each other. Hence concurrently processing or multi-threading can be applied to speed up the searching process. Also possible values of any empty block can be checked concurrently. For example, the row, column, and all inner boxes that contain the block can be checked concurrently though multiple threads (1 thread for checking the row, 1 thread for checking the column, 1 or more threads for checking inner boxes).

There are also several disadvantages of using concurrently processing, specifically multi-threading (assuming we have enough cores for multi-threading). For a small grid size, the overhead of creating, executing, terminating threads may take longer time than single thread. Also each thread has its own memory space, the overall memory usage will be much higher than a single thread application. This is especially serious for this memory-consuming problem. Another problem could be when to create threads. Obviously it is impossible to create a thread for every iteration, if there is only one possible value for a block, multi-threading is not needed. Multi-threading is only needed when a block has multiple possible values. But such block may be found in any time in the process.

### Stochastic Value Selection

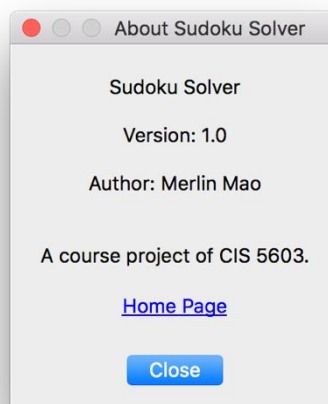
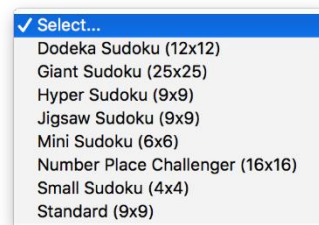
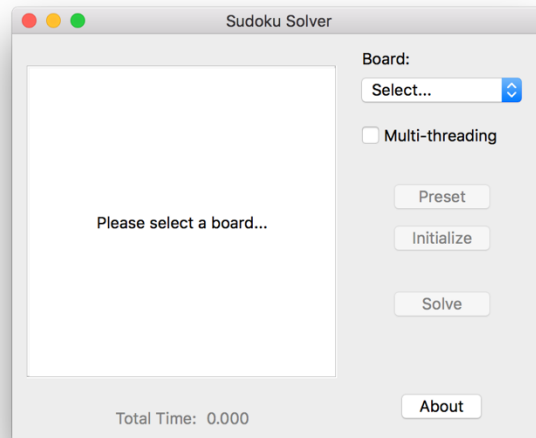
Instead of testing possible values in order, a value can be randomly chosen. Theoretically this is faster and resource efficient. But still, it can also reach the worst scenario, which is the same as testing possible values in order. Also the implementation of this method will be more complicated.

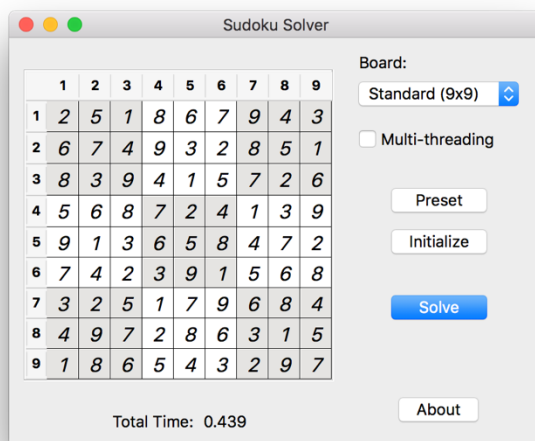
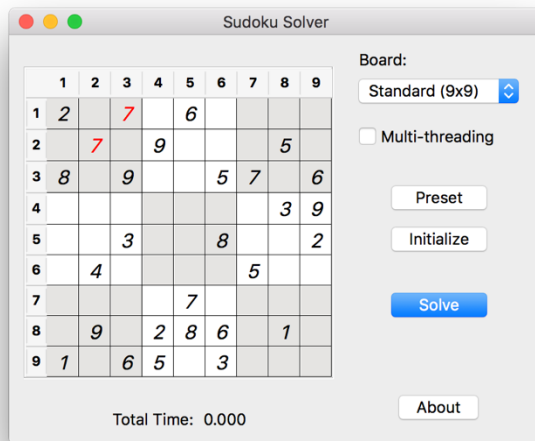
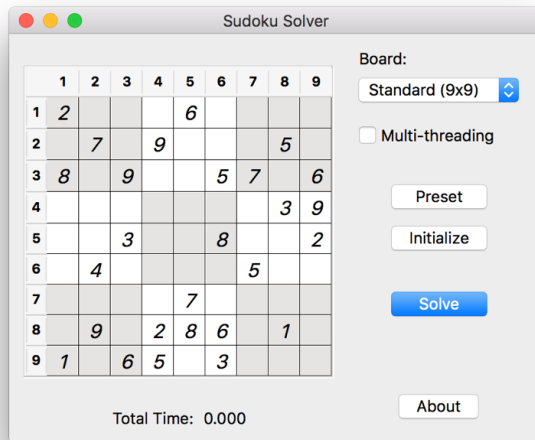
## Application

- **Home Page:** <https://github.com/autopear/Sudoku-Solver> (Source code and executables available)
- **Developer:** Qizhong Mao
- **License:** LGPL 3.0
- **Language:** Qt (C++)
- **Qt Version:** 5.0+
- **Support Platforms:** Windows (x86/x64) \*, Mac OS X (Other major operating systems are also supported)  
\* [Visual C++ Redistributable Packages for Visual Studio 2013](#) may be required
- **Features:**
  - ✓ Include 8 Sudoku variants to play
  - ✓ Heuristic search method to solve the game
  - ✓ Real-time validation is supported if the game is played by human

- ✓ The time to solve the game automatically is provided
- ✓ Easy to load presets
- ✓ Easy to create and edit preset, and save it to file for reuse
- ✓ Flexible to add more Sudoku variants

## Screenshots





## Add a New Board Variant

A board is a *.board* file under the Boards directory. There are several settings in the file, each setting occupies a single line or multiple contiguous lines.

## Settings

- **NAME:** The name of the variant, must be unique
- **ROWS:** The number of rows
- **COLUMN:** The number of columns
- **MINIMUM:** The minimum value allowed, must be at least 1
- **MAXIMUM:** The maximum value allowed, must be greater than the minimum value
- **MINVALUES:** The minimum number of blocks with preset values required to solve the game. Settings this value can avoid some situation such as filling empty grid, which take too long to complete.
- **BLOCKS:** A list of list of blocks that represent inner boxes. Start from top-left to bottom right,  $(x, y)$  represents a block in the grid, where  $x$  is the row number and  $y$  is the column number. Both  $x$  and  $y$  start from 0 to  $N - 1$ . Different blocks are separated by a single comma, and an inner box is enclosed by [ and ] with multiple blocks inside. Different inner boxes are separated by a single comma also, except there is a colon in the end. Blocks and inner boxes may be in one line or across multiple lines. All blocks within an inner block and all inner blocks must be in ascending order.
- **COLORS:** A list of colors for every inner block. The size of this list must be the same as the number of inner blocks. Different colors are separated by a single comma. Valid colors are standard HTML color codes, and strings listed in this page: <http://doc.qt.io/qt-5/qcolor.html#predefined-colors>.

```
NAME: Small Sudoku
ROWS: 4
COLUMNS: 4
MINIMUM: 1
MAXIMUM: 4
MINVALUES: 0
BLOCKS:
[(0, 0), (0, 1), (1, 0), (1, 1)],
[(0, 2), (0, 3), (1, 2), (1, 3)],
[(2, 0), (2, 1), (3, 0), (3, 1)],
[(2, 2), (2, 3), (3, 2), (3, 3)];
COLORS: #E5E4E2, #FFFFFF, #FFFFFF, #E5E4E2
```

### Example of Small Sudoku

Note that **ROWS** and **COLUMNS** are set independently, this makes the application support rectangle grids. Also the **MINIMUM** and **MAXIMUM** can be set to start with 1 or any integer greater than 1.

## Create a Preset File

Preset is a text file with *.sdk* extension. There are two ways to create a preset file.

### 1. Directly edit using text editor


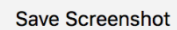
Preset values are separated by commas. Empty block can be represented by 0 or empty string. Each row in the grid requires a line in the file. There is no need of a comma in the end of a line in the file.

```
0, 0, 0, 4
0, 4, 2, 0
0, 1, 3, 0
3, 0, 0, 0
```

### Example of Small Sudoku Preset

## 2. Through the application

User can directly input the preset values to the application, and use the “Save as Preset” function from the context menu in the grid area to save the current preset values into a .sdk file.

A rectangular button with rounded corners and a light gray background, containing the text "Save as Preset".A rectangular button with rounded corners and a light gray background, containing the text "Save Screenshot".

## Summary

In this project, we implemented a playable application to solve a variety of Sudoku games by using heuristic search and multi-threading. The heuristic search is more efficient than brute force search in most cases (though the worst can be the same). Our application is compatible with most Sudoku variants and is flexible to add more variants. A solution is guaranteed (or no solution will be given), though sometimes it is restricted by the hardware power. The application does not only support solving a Sudoku game, users can also play the game by themselves.

We have studied several facts that may affect the heuristic search, and proposed a stochastic heuristic search model as a future work.

## References

- [1] <https://en.wikipedia.org/wiki/Sudoku>
- [2] [https://en.wikipedia.org/wiki/Brute-force\\_search](https://en.wikipedia.org/wiki/Brute-force_search)
- [3] [https://en.wikipedia.org/wiki/Heuristic\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Heuristic_(computer_science))
- [4] [https://en.wikipedia.org/wiki/Sudoku\\_solving\\_algorithms](https://en.wikipedia.org/wiki/Sudoku_solving_algorithms)
- [5] <http://doc.qt.io/qt-5/qcolor.html> - predefined-colors.
- [6] <http://www.lamsade.dauphine.fr/~cazenave/papers/sudoku.pdf>
- [7] <http://publish.wm.edu/cgi/viewcontent.cgi?article=1077&context=caaurj>
- [8] [http://www.sudokuwiki.org/sudoku\\_creation\\_and\\_grading.pdf](http://www.sudokuwiki.org/sudoku_creation_and_grading.pdf)
- [9] <http://www.sudoku-solutions.com>
- [10] <http://angusj.com/sudoku/hints.php>
- [11] <http://www.paulspages.co.uk/sudoku/howtosolve/>
- [12] <http://www.bigfishgames.com/blog/how-to-solve-sudoku-puzzles-quickly-and-reliably/>