Heuristic Solver for Various Sudoku

Bao Yelun, 2014012710
Liang Zhenxiao, 2014012508

January 14, 2018

1 Introduction

Sudoku, originally called Number Place, is a logic-based, combinatorial number-placement puzzle. The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids that compose the grid (also called "boxes", "blocks", or "regions") contains all of the digits from 1 to 9. The Figure 1 gives an example of normal Sudoku Game. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a single solution. Completed games are always a type of Latin square with an additional constraint on the contents of individual regions. For example, the same single integer may not appear twice in the same row, column, or any of the nine 3×3 subregions of the 9×9 playing board.

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

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

Figure 1: Example of a Normal Sudoku Puzzle and its Solution

In this project we want to design an AI engine to play Sudoku game. The Sudoku will not only be the classic 9×9 playing board, but also bigger size Sudoku and Sudoku with irregular cells, by introducing new algorithms. Figure 2 provides an example of an irregular Sudoku Puzzle.

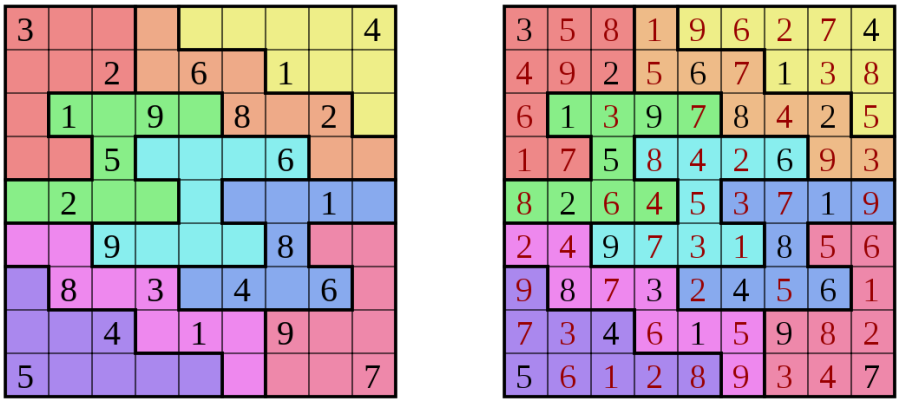

Figure 2: Example of an Irregular Sudoku Puzzle and its Solution

2 Infrastructure

First of all, to test the perfomance of Sudoku algorithms for normal 9×9 Sudoku games, we run our proposed algorithm and the trivial trackback algorithm on the open 9×9 Sudoku dataset [7], which includes one million puzzles and each of them has exactly one solution.

Our algorithm also used for larger or irregular puzzles, but there are few corresponding open large dataset. Hence we wrote our own generater to generate irregular puzzles of arbitrary size and test the algorithms on these generated puzzles (see Fig 3). However, it is hard to gurantee each generated puzzle has only one solutions when generating very large puzzles.

```
partition label map:
 [[0 0 1 1 2 2 2 3 3]
 [4 0 0 1 2 2 2 3 3]
 [4 4 0 1 2 2 2 3 3]
 [4 4 0 1 1 1 1 1 3]
 [4 5 0 0 0 5 5 3 3]
 [4 5 5 5 5 5 5 6 6]
 [4 4 7 7 6 6 6 6 6]
 [7 7 7 7 6 6 8 8 8]
 [7 7 7 8 8 8 8 8 8]]
```

```
one solution:
 [[4 1 6 2 7 3 5 8 0]
 [6 8 2 3 0 4 1 7 5]
 [5 0 7 4 6 8 2 3 1]
 [2 4 3 5 1 7 8 0 6]
 [8 7 0 6 5 1 3 2 4]
 [7 5 8 0 4 2 6 1 3]
 [1 3 5 8 2 0 4 6 7]
 [3 6 1 7 8 5 0 4 2]
 [0 2 4 1 3 6 7 5 8]]
```

Figure 3: Generated Irregular Puzzle

Here are multiple criterions that can be used to measure the performances of Sudoku algorithms, and we choose two simple criterions to compare our proposed algorithm with the already-known one:

1. Average time consumed to solve one puzzle. Obviously this is an intuitive measure for performance and can reflect how well an algorithm works in practice. However,

this criterion also depends on how to implement the algorithm and which machine the program is running on.

2. The number of node explored in the search procedure. This criterion is independent of specific implementation and could reflect goodness of our heuristic function. But this measure ignores the cost of calculating heuristic function, so that an aggressive heuristic that explores less nodes may cost much more time in practice.

3 Literature review

Various algorithms have been implemented to solve the Sudoku problem [1]. A logic-based algorithm, mimicking the way a human would solve the puzzle, is adequate to attain a solution. Harder puzzles, where guessing is required, can be solved using backtracking algorithms [1]. The problem with back tracking is that the efficiency of the algorithm is dependent on the number of guesses required to solve the puzzle [2]. A solution to this problem could lie in using stochastic optimization techniques.

Some work has been done on solving Sudoku using stochastic optimization techniques [2-4]. The primary motivation behind using these techniques is that difficult puzzles can be solved as efficiently as simple puzzles. This is due to the fact that the solution space is searched stochastically until a suitable solution is found [2]. Hence, the puzzle does not have to be logically solvable or easy for a solution to be reached efficiently.

Furthermore, stochastic optimization techniques are used to find the global optimum of a problem which contains many local optima [4]. Due to the constraint nature of Sudoku, it is very likely to find a solution which satisfies some of the constraints but not all of them, hence finding a local optimum. Due to the stochastic nature of these techniques, the solution space is still searched even though a local optimum has been found, allowing for the global optimum to be detected.

Another solution is to improve the exploration process so that we can find a feasible solution before expanding too much nodes. Therefore, a well-designed heuristic function is very important to our exploration process. A brute force algorithm visits the empty cells in some order, filling in digits sequentially, or backtracking when the number is found to be not valid [6]. Minimum Remaining Values ordering is proposed when faced with the decision on what order should its cells be explored [3]. However, when faced with which value should be tried first in a cell, most solvers just use lexicographical order to decide it. Similar to the order of cell exploration, Minimum Remaining Values ordering is also considered to work well in this choice [3], but the definition of this heuristic is not very clear. This part can be improved a lot.

Some other work has been done on reducing Sudoku puzzle to graph coloring problems, or more often an exact cover problem. Then they want to use Knuths elegant Dancing Links algorithm (DLX) [5]. However, this technique is not very direct to the origin puzzle and the efficiency is restricted by the DLX algorithm. Thus, we do not consider this algorithm.

4 Algorithm Description

The input of a Sudoku game is a digit sequence with totally 81 digits, each of which represents the given number in the puzzle, from left to right, top to bottom. We will use 0 to represent the empty grids to be fill out. And the expected solution, the output, is also a 81-digit sequence, which is the right digit assignment satisfying the rules. There is no 0 in the output.

We are going to develop some new algorithms to speed up the solving process. Based on back tracing algorithm, we assign the nonempty position's value to the initial dictionary variable, which is the table of our playing board. When we meet 0, assign the list $[1, 2, 3, 4, 5, 6, 7, 8, 9]$ to the entry. Then eliminate the digits in each grid which violate the constraint rules in current state. If a grid's list remains only one digit, assign it the grid and update the current state. Naked Twin or Triplets trick, which is also called Naked Twin exclusion rule, is used, in order to speed up this process. Even if the pair of grids are not assigned, we can also use these naked twins (or triplets) to eliminate some candidates. Implement recursion techniques to create depth-first-search trees for solving hard sudokus that need guess work in any particular box to proceed. If we can find a feasible solution, just halt and return this solution.

5 Advanced Variation

In the search process we need guess the number in every blank box to proceed. Here the right choice, or reasonable attempt priority is very important to our solver, which can help us find a feasible solution much faster. Therefore, a reasonable and novel heuristic function, instead of simple natural number order, is a very good tool to decide the order of trial. For now we use a specially designed heuristic function to speed up our search. Basically, this heuristic function is calculated by summing over the number of possible values in each cell and then minus the number of cells. It is easy to see that a valid solution is reached if and only if the heuristics function is exactly zero. The intuition behind the definition of heuristics function is that when the function value is a small non-negative integer then the further search procedure from this state would be relatively quick since the branches in search tree from that state would be rather sparse. However, sometimes there may be no valid number that can be put into some cell. In this case we let the heuristics function be infinity. There is a trade-off between the big and small heuristics function value. The bigger function value, the more complicated situation we will face with. The less function value, the more chance that it will end up as an unfeasible case. We want to set a reasonable threshold value, which has much to do with the number of cells not assigned. We set part of the 9 Sudoku dataset as "training set", and use this set to adjust the threshold value manually. Then use another dataset as our test set.

Moreover, we will also try to solve Sudoku with bigger size. The advantage of imple-

menting heuristic function will be more significant with the growth of the size. We will also try to solve for Sudoku with irregular cells. The performance of the new algorithm can be measured by the time it consumes to solve the testing set problems, compared to that consumed by the simple back tracing solver. This evaluation also involves the design of larger Sudoku, since this data set is hard to find on the Internet.

6 Experiment Results

Now we have completed the basic back tracing algorithm (which can be seen in util.py). We can run the python file and type the input (n^2 numbers for normal $n \times n$ Sudoku Puzzle where n is a complete square number), and then we can get one of its solutions. This algorithm will also use some other novel tricks, and will be generalized to a more universal solver so that we can solve the Sudoku Puzzle with irregular blocks.

The following are some comparisons between the known algorithm and ours. We ran our solver on 20 very hard 99 Sudoku puzzles and five 16x16 Sudoku puzzles, which are chosen as test data (differ from the data to adjust the Heuristic.) The basic solver is the Sudoku solver using backtracing algorithm, without minimum remaining values ordering in cells exploration and lexicographical order in digit choice. The simple heuristic solver uses simple minimum constraining ordering.

Solver	9×9	16×16
Basic Solver	2872	2536
Simple Heuristic Solver	2197	1728
Our Heuristic Solver	2164	1872

Table 1: The number of expanded nodes for different solvers

Solver	9×9	16×16
Basic Solver	8428	33039
Simple Heuristic Solver	6173	26651
Our Heuristic Solver	6012	24713

Table 2: The time used to solve hard puzzles for different solvers (in ms)

References

[1] G. Santos-Garcia and M. Palomino. Solving Sudoku Puzzles with Rewriting Rules. Electronic Notes in Theoretical Computer Science, 17(4), 79-93.

[2] T. Mantere and J. Koljonen. Solving and Rating Sudoku Puzzles using Genetic Algorithms. Proceeding of the IEEE Congress on Evolutionary Computation, 1382-1389.

```
❯ python3 main.py
type sudoku as a long string of 81 characters with 0 as unsolved place
004300209005009001070060043006002087190007400050083000600000105003508690042910300
8 6 4 |3 7 1 |2 5 9
3 2 5 |8 4 9 |7 6 1
9 7 1 |2 6 5 |8 4 3
------+------+------
4 3 6 |1 9 2 |5 8 7
1 9 8 |6 5 7 |4 3 2
2 5 7 |4 8 3 |9 1 6
------+------+------
6 8 9 |7 3 4 |1 2 5
7 1 3 |5 2 8 |6 9 4
5 4 2 |9 1 6 |3 7 8
```

Figure 4: A usage example of our program on some normal 9 Sudoku Puzzle

[3] R. Lewis. Metaheuristics can Solve Sudoku Puzzles. Journal of Heuristics Archive, 13(4), 387-401.

[4] A. Moraglio and J. Togelius. Geometric Particle Swarm Optimization for Sudoku Puzzles. Webpage: http://julian.togelius.com/Moraglio2007Geometric.pdf.

[5] Knuth, Donald. "Dancing links". Millennial Perspectives in Computer Science. P159. 187. arXiv:cs/0011047Freely accessible.

[6] Norvig, Peter. "Solving Every Sudoku Puzzle". Peter Norvig (personal website). Archived from the original on 20 Oct 2016.

[7] "1 Million Numpy Array Pairs of Sudoku Games and Solutions"