Sudoku puzzle solver using backtracking approach

2008817

Brief introduction to backtracking

Backtracking method is a type of brute force search. For solving a given Sudoku puzzle, the program firstly finds out all the empty square from 9x9 grid (usually starts with 1st row and 1st column). Then, when reaching a new empty square, the program fills in the possible digit (in the range from 1 to 9) and check whether it fit the rule of sudoku. If the digit is not allowed to be placed in here, the program will backtrack the previous cell and try another digit. If the digit meets Sudoku rules, the program head to the next cell, until the Sudoku puzzle is solved.

A summary of the algorithm

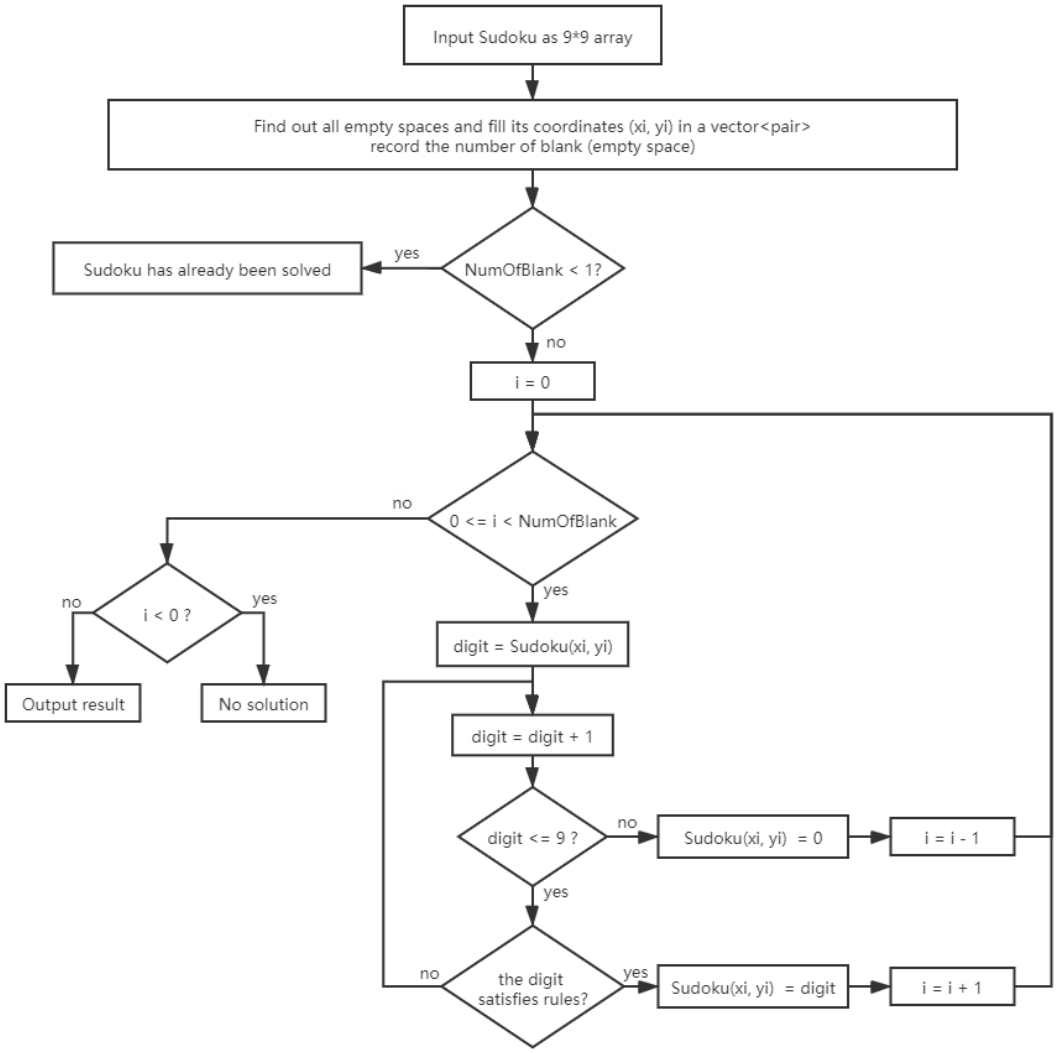

Figure 1. flow diagram of the implementation

Why backtracking

There are several different approaches to solve the Sudoku puzzle by algorithm, including backtracking, stochastic search, constrain programming and exact cover. To be honest,

backtracking might not be the best approach in many aspects, but I decided to use it for the following reasons.

Most importantly, backtracking looks attractive because it is easy to understand and implement. Nevertheless, its performance does not have to be worse than other methods when solving most of Sudoku puzzle.

Time cost is basically unrelated the difficulty of the Sudoku puzzle. For some approach, like constrain programming, the time cost and the difficulty to implement will rise sharply when the difficulty of Sudoku increasing. I have tried to understand the constrain programming. Frankly, although it seems that I can still implement this approach for simple Sudoku, it is difficult to solve the difficult Sudoku.

Some Sudoku could be not solution or more than one solution, which can be identified using backtracking. When I considered whether to use stochastic search, I found it difficult to identify whether there are multiple solutions for Sudoku (should I stop to "shuffle" the digits when I find the first solution? If not, then when solving a “unproper” Sudoku, the program will run continually and output the same solution.) There might be some way to deal with this problem, but at least I have not found an acceptable solution.

Algorithm efficiency

I think that the backtracking has a relatively high efficiency when solving Sudoku with fewer empty spaces. However, it has no advantage when solving the Sudoku with many empty spaces, because any one more empty space will greatly increase the number of backtracking, which means perform an extra large number of loops. In addition, even compared to the stochastic search (similar to brute-force approach), when solving some particular Sudoku, backtracking is not dominant because of its own characteristics. For example, some Sudoku with a lot of empty space in the first few rows could take it more time, because every time the program identifies a violation, it needs to backtrack to check the very previous cell. Therefore, backtracking may require more time to solve these special Sudoku puzzles, especially compared with some logic-based algorithms. Users can enter ***ex1.csv*** (the sample Sudoku) and ***ex2.csv*** (the designed Sudoku for backtracking) and compare the time cost (have to move these to csv files to the debug folder manually as the *cmake-build-debug* will not be submitted.

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

Figure 2. A Sudoku designed to work against the brute force algorithm

Despite these shortcomings, backtracking is still a qualified Sudoku solving approach. It can solve all the “proper” Sudoku puzzle, even some of them cannot be solve logically. Moreover,

backtracking can identify “improper” Sudoku and its time cost is independent to the difficulty.

Efficiency of implementation

Solving a Sudoku by backtracking, the program needs to backtrack and check the rules over and over again, which leads to numerous loops. Hence, how to optimize the “for” loops is the key to improving overall efficiency.

To do this, I tried to write the for loops using an iterator, but I found that it did not perform better than the normal one in my program. It is unreasonable that iterators perform even worse than ordinary loops. A reasonable explanation is that in my program, in order to build an iterator for a “for” loop, a vector needs to be repeatedly passed to the function (as a reference argument) or initialized (being define in the function), and these operations take a lot of time.

Form of for loop	Ex1	Ex2
`for (int i {0}; i < 9; ++i)`	5 ms	4562 ms
`for (auto i {std::begin(vec)}; std::end(vec) != i; ++i)`	39 ms	184718 ms
`for (const int &i : vec)`	36 ms	140374 ms

Ps 1: The time shown in the table is the average time of 50 runs.

Ps 2: Ex1 is the example Sudoku in the assignment, Ex2 is a Sudoku against backtracking

Ps 3: I changed the form of loop in the function “FitOrNot” (line 106 in 2008817.cpp), which is called repeatedly.

Then, I try to combine some loops into one. In the function “FitOrNot”, I combine two “for” loops together, which were used to check the row and column respectively. Surprisingly, this small change greatly improves efficiency.

		Ex2
Before combine	two separate `for (int i {0}; i < 9; ++i)`	4562 ms

Combined
 	
(Line 111 in 2008817.cpp)
	`for (int i {0}; i < 9; ++i) {`
	`  if (digit == sudoku[x][i])`
	`  return 0;`
	`  if (digit == sudoku[i][y])`
	`  return 0;`
	`}`
		4084 ms

In addition, I searched for better containers to store coordinates of empty spaces. Using a vector with its element type is pair<int, int> will perform better than using vector<vector<int>>.

Different element types	Ex2
`vector<vector<int>> CoordOfBlank;`	4084 ms
`vector<pair<int, int>> CoordOfBlank;`	3305 ms