UNIVERSITY OF BURGUNDY

COMPUTER SCIENCE

Sudoku

Author
LASSE MACKEPRANG

May 30, 2019

Contents

1 Introduction

This project report was for the Computer Science class in the last semester of BSCV at the university of Burgundy class 2018-2019. The task was to make a Sudoku solver using C++. Approximately three weeks were provided for the project. This report is best read with the comments of the program on the side.

1 The Sudoku puzzle

A Sudoku is a simple number puzzle consisting of a 9 by 9 grid with 9 sub-grids dividing the big grid into 3 by 3 sections. The rules are that you have to put the numbers 1 to 9 in every row, column and 3 by 3 sub-grid without repeating numbers. At the start some squares are provided with numbers while the rest are left blank, and then you start solving by filling in empty squares. A proper Sudoku puzzle will only have one unique solution, though it happens that multiple very similar solutions can occur.

1 Solving the puzzle

There are several methods of solving a Sudoku puzzle. The most simple way is by finding a square where only one sole number fits according to the rules. Most easy Sudoku's can be solved this way by going through all the blank squares and filling in these sole numbers, this process is repeated until the puzzle is solved. When the difficulty of a Sudoku is increased, the sole number method becomes insufficient, when this happens more advanced methods of solving have to be used, and in some cases a Sudoku cannot be solved without guessing at least one number.
Advanced solving techniques can be seen at:
https://www.kristanix.com/sudokuepic/sudoku-solving-techniques.php

2 C++ solution

The task was to find a way to make a Sudoku solver using C++ with no specification of how. Some form of GUI was also to be provided although i did not make one.

1 Sudoku solver in C++

My initial idea was to implement a solver with logic similar to how I think when i try to solve a Sudoku. Although it is a fairly simple process, this task is easier said than done. And a few different solutions were tried before everything fell in place.
I wanted it to be easy to input a Sudoku to be solved, and decided that a simple .txt text file would suffice. This text file is filled all rows of the Sudoku, that is all numbers and blanks which are set as "0". The text file should be 9 lines of 9 numbers. The path to this file is then provided for the program to work. Displaying the Sudoku is also important, so a function has to be made for that purpose. For storing the Sudoku itself, I initially went for a 2D vector, but since I realized the size of the Sudoku would never change, I changed that to a 9 by 9 array. First when using the vectors, I tried if it would be easier to make three separate vectors, each for the Sudoku's Rows, Columns, and 3 by 3 sub-cells, where the two latter would be filled with pointers to the first Sudoku. This quickly became tedious, and I abandoned the idea. Finally the solving methods where to be decided on, and since a lot of methods exist, some would be very tedious to program. I decided to implement the Sole number and Unique candidate methods for logically solve the Sudoku, and for guesswork after no number could be found logically some brute force was to be used.

1 Implementing the sole number finder

For the sole number finder I had to implement a way to check for specific numbers in the horizontal and vertical directions at specific coordinates of the Sudoku array, as well as checking for these numbers in their respective sub-boxes. The horizontal and vertical checks are very simple as a single for loop can check these, while box checker requires a nested for-loop as demonstrated below:

```
int Sudoku::bcheck(int s[9][9], int row, int col, int num){
 for (int x=0;x<3;x++)
  for (int y=0;y<3;y++)
   if (s[x+(row-row%3)][y+(col-col%3)] == num) return 1;
 return 0;
}
```

These three functions are then used in a solving function to go through the Sudoku's blank squares, and numbers are filled in where only one sole number can be placed.

2 Going further with unique candidate method

The next thing I decided to do was to implement a unique candidate algorithm. This is realized by checking the adjacent squares horizontals and verticals in the same 3 by 3 sub-box for a potential number. If this number can fit in none the adjacent squares, then the number can only be placed at this square. This is partly realized in the next_check() function. This function is run after the check for the sole number check for efficiency. I realized later that a few scenarios are missing. It still solves a lot of medium difficulty Sudoku's as is.

Obviously a lot of Sudoku solving algorithms can be written and implemented in C++, and computationally these algorithms would be very effective. The problem with implementing them is that a brute forcing algorithm can written is a fraction of the time.

3 Brute force using recursion

Far the simplest solution to a Sudoku solver is simple brute force. Even without logically solving a Sudoku, there are only so many possible arrangements of numbers in a 9 by 9 grid. And with computers of today going through all the possibilities is trivial. Implementing a recursive function to solve a Sudoku is not a new idea, and has been done many times before. A very simple backtracking solver from which i got a lot of inspiration can be found at:
https://www.geeksforgeeks.org/sudoku-backtracking-7/

The basic gist of this recursive function is that it finds a free square in the Sudoku array, Goes through all the numbers and checks if that number is not in the same

horizontal and vertical lines of the coordinate and if it is not in the same sub-box. If the number is not found, the square is set to that number. This is recursively done, and once the Sudoku has been solved a "1" is returned by the function. If no solution is found, a "0" is returned and the guessed square reverted to "0".

```
int Sudoku::brute_solve(int s[9][9]){
 int row, col;
 if (!find_zero(s, row, col)){
  return 1;
 }
 for (int num=1;num<10;num++){
  if (!all_check(s, row ,col, num)){
   s[row][col] = num;
   if (brute_solve(s)) return 1;
   else s[row][col] = 0;
  }
 }
 return 0;
}
```

This is effectively the only thing you need to solve a Sudoku.

2 Conclusion

The Sudoku solver works very well, it has shown that it is much easier to use a recursive brute forcing algorithm rather than an advanced logical algorithm. In the final implementation, if only brute force was to be used, the following functions could be left out: "next_check()", "missing_numbers()" and "logic_solve()", leaving only a few functions left. In the end I did not implement a GUI, but solely printed through the console.