Bu-Ali Sina University

Report

Solving Sudoku with Backtracking

Mehran Shahidi

supervised by
Dr. Moharam Mansorizadeh

June 22, 2020

Summary

In this project, we will solve Sudoku using Backtracking and implement it in C++.
To access the code and see more visual documents you can vist the follwing github repository.
https://github.com/m3hransh/sudoku

Contents

1 introduction **1**

2 implementaion of solution with Backtracking **1**

2.1 implementaion in C++ 1

3 Analysis **4**

1 introduction

Sudoku is a logic-based, combinatorial number-placement puzzle. In classic sudoku, 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 contain all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a single solution.

Completed games are always an example of a Latin square, including 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

(a) example of a sudoko 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

(b) corrosponding solution of the puzzle of (a)

2 implementaion of solution with Backtracking

One of the good method for solving Sudoku is using Backtracking. Sudoku can be solved by one by one assigning numbers to empty cells. Before assigning a number, check whether it is safe to assign. Check that the same number is not present in the current row, current column and current $3*3$ subgrid. After checking for safety, assign the number, and recursively check whether this assignment leads to a solution or not. If the assignment doesnt lead to a solution, then try the next number for the current empty cell. And if none of the number (1 to 9) leads to a solution, return false and print no solution exists.

2.1 implementaion in C++

Our main fucntion is shown in the Listing 1. In the for loop all possibele choices for k_{th} element in the array of empty cells(emptycells is global array that is initialize with empty cells index and value of 0) are consirdered and if the choice is promising we continue our process on the next element of empty cells array.

```cpp
void sudoku(int k){
   for(int i=1; i<= N;++i){
       //The value of cell is char so we add '0' to assign ASCII number
       emptycells[k].v = '0' + i;
       if (promising(k)){
            if(k== emptycells.size()-1){
                print_table();
            }
            else
                sudoku(k+1);
```

```
        }
    }
}
```

Listing 1: main recursive function for implementing backtracking

Now lets see how the promising fucntion works. The code is shown in the Listing 2. The function has four things to check the compatiblity.

1. elements in the row
2. elements in the column
3. elements in the $3 * 3$ square
4. previous choices of cells 0,...,k-1 of emptycells array.

```
bool promising( int k){
    if (check_row(k) && check_col(k) && check_sq(k) && check_pre_cells(k))
        return true;
    else
        return false;
}
```

Listing 2: promising function

Implementation of each check functions are shown in Listing 3

```
//check if all elements of the row element k is in compatiblity with the choice of k
bool check_row(int k){
    //take care of constant valued cells
    for(int j=0; j<N; ++j){
        //to make sure is not the same cell
        if(emptycells[k].y != '0' +j){
            if (table[emptycells[k].x-'0'][j] == emptycells[k].v){
                return false;
            }
        }
    }
    return true;
}
bool check_col(int k){
    //take care of constant valued cells
    for(int i=0; i<N; ++i){
        // check it isn't the same element
        if(emptycells[k].x != '0'+i){
            if (table[i][emptycells[k].y-'0'] == emptycells[k].v){
                return false;
            }
        }
    }
    return true;
}
bool check_sq(int k){
    //finding the top left square in 3*3 square k is in
    int sq_row = int(emptycells[k].x /3)*3;
    int sq_col = int(emptycells[k].x /3)*3;
    for(int i=sq_row; i <sq_row+3;++i){
        for(int j=sq_col ; j<sq_col+3;++j){
            if(emptycells[k].x -'0' != i || emptycells[k].y -'0' !=j)
            {
                if(emptycells[k].v == table[i][j])
                    return false;
            }
        }
    }
    return true;

}
//checking if the previous choices are compatible
```

```cpp
bool check_pre_cells(int k){
    //the squre that element k is in
    int sq_row = int(emptycells[k].x /3)*3;
    int sq_col = int(emptycells[k].y /3)*3;

    for(int i=0; i<k; ++i){
        //row_check
        if(emptycells[i].x == emptycells[k].x)
            if (emptycells[i].v == emptycells[k].v)
                return false;
        //col_check
        if(emptycells[i].y == emptycells[k].y)
            if (emptycells[i].v == emptycells[k].v)
                return false;
        //square_check
        if(int(emptycells[i].x/3)*3 == sq_row && int(emptycells[i].y/3)*3 == sq_col)
            if (emptycells[i].v == emptycells[k].v)
                return false;
    }
    return true;
}
```

Listing 3: checking functions

Now to make our life easier for testing our functions on different senarios of the puzzle we write codes to read puzzles from a file that user has specified in the terminal argument and after that store it in a 2-D array. Then we create a vector of data structure cell that that has value x(row number), y(column number) and v(value of the cell) to store the empty cells that the program is meant to fill. You can see all of this in Listing 4

```cpp
struct cell{
    char x;
    char y;
    char v;
};

//global variables
char table[N][N]; // the main sudoku table
std::vector<cell> emptycells;// list of empty cells

int main(int argc, char** argv){
    //getting file name from argv in terminal
    //else default value of game1.txt
    string filename;
    if (argc >1)
        filename = argv[1];
    else
       filename = "game1.txt";
    ifstream myfile;
    myfile.open(filename);
    string line;
    int i=0;
    while (getline(myfile,line)) {
        istringstream iss(line);
        for(int j=0; j<N; ++j){
            iss>>table[i][j];
            if (table[i][j] == '0'){
                cell c{char('0'+i),char('0'+j),'0'};
                emptycells.push_back(c);
            }
        }
        i++;
      }
    sudoku(0);
    return 0;
}
```

Listing 4: other parts of program

Now with the help of g++ we can compile our program.

```
$g++ -std=c++14 main.cpp -o out
```

now we store a puzzle in a file like the one is shown in the Figure 2

Figure 1: input.txt

```
4 0 0 0 0 2 8 3 0
0 8 0 1 0 4 0 0 2
7 0 6 0 8 0 5 0 0
1 0 0 0 0 7 0 5 0
2 7 0 5 0 0 0 1 9
0 3 0 9 4 0 0 0 6
0 0 8 0 9 0 7 0 5
3 0 0 8 0 6 0 9 0
0 4 2 7 0 0 0 0 3
```

And then with following command you can solve the puzzle.

```
$./out input.txt
```

This is what you get in the terminal:

Figure 2: input.txt

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

3 Analysis

In this part we calculate complexity of the code that was written in the previous section. In this Analysis we only consider the code related to solving the sudoku and we ignore the reading from the file and storing them in matrix. Suppose our puzzle is $N * N$ and it has K elements to fill. First thing that we will calculate is emptycells, that it takes $O(N * N)$ cause we need to visit all the cell to see which one has the value of zero. The next part is sudoku function that in the worst case it will run on the all choices so our runtime is the recursive form of the following:

$$T(k) = 9T(k-1) + max(O(N), O(k))$$

By a little bit of sloppiness about k the number of empty cells and by considering that k is a number betwenn $1 \leq K \leq N * N$ the above recurrence give us the time complexity of $O(9^{N*N})$. For the space complexity we will need O(N*N) to just store the puzzle and other varibales don't exceed this bound.