CSCE 2211 Term Sudoku Game Project Report Fall 2024

Yomna Othman, Ayla Saleh, Omar Beheiry,
Saif Hashish & Abdulaziz Alhaidary
Dr. Dina Mahmoud
Section 01

CSCE 2211-01: Applied Data Structures
December 5, 2024

Abstract

Sudoku has remained a game of interest all throughout the years. Thus, our project presents a Qt application that provides a user-friendly gaming experience of Sudoku. This implementation utilizes the well-known backtracking algorithm which has proven itself efficient in handling the constraints of Sudoku while still maintaining an engaging user experience. This document plans to outline the problem, methodology, algorithms, data specifications, experimental results, and analysis of the effectiveness of our implementation and process in making said Qt application.

Keywords: Sudoku, backtracking algorithm, puzzle solving, sudoku gaming

1.1 Introduction

Sudoku is a puzzle game that requires filing a 9x9 grid with digits (1 to 9) such that each row, column, and 3x3 box contain all digits from 1 to 9 exactly and only once. Solving Sudoku requires that you keep in mind all the constraints that you have provided by both the rules of the Sudoku and the given grid, i.e. the numbers provided and/or fixed. In this report, we will go over two possible approaches to create a Sudoku solver, and show our exact implementation and analysis of the process.

1.2 Problem Definition

The Sudoku game is the epitome of a constraint satisfaction problem (CSP), where its solution and decision-making has to be within some constraint. More specifically, in Sudoku, the game requires that each number be placed while simultaneously ensuring that it satisfies the following constraints:

Row, Column, and Subgrid Constraints—

Each row, column, subgrid must contain all the digits from 1 to 9 without repetition in the same row, column, and subgrid.

This is the case for a 9x9 Sudoku; however, it is important to note that different Sudokus also follow the same structure with minor differences, like grid size and digit range.

The main problem is actually navigating all possible solutions for each square in the 9x9 grid while maintaining that the constraints are not violated. Our project discusses the possible algorithms that could be used to solve this problem and tackles the problem by leveraging the backtracking algorithm while implementing an interface to deliver a Sudoku game and solver.

1.3 Methodology

To solve this problem, we could approach it in multiple ways. We will only discuss two methods in this report; however, that is not to say that these two techniques are the only techniques to solve the problem. In fact, a more optimal solution can be found if we were to combine the two methods into one.

A. Approach 1 (Brute Force):

The whole entire idea of this approach is to essentially generate all possible boards that can be generated from the given Sudoku puzzle. After which, the algorithm will go through each of the possible boards to validate them. We know that given a detailed, accurate, and solvable Sudoku, the algorithm would create all possible boards and still have only one valid board, i.e. only one valid solution to the Sudoku puzzle.This would be equivalent to validating an *exponential* amount of full boards only to find that only one is possible with the constraints.

> Within those exponential amounts of boards, some boards can be trivially eliminated as not possible. For example, if we filled all empty spaces with only one digit over and over, we know that somewhere there will inevitably be repetition in either the row, column, and/or 3x3 subgrid, resulting in an invalidate solution provided that we consider the constraints.

Knowing that, we can probably see where the next approach would go. We will need something that constantly checks as it adds digits to the possible solution board.

B. Approach 2 (Backtracking):

We use backtracking and direct our recursion process. Essentially, we choose a number and recurse on the decision. Going row by row, to solve column by column, we make sure that our placements do not break the board, i.e. violate any of our constraints. If we move to the next box, then we know that whatever digit we placed in the last space didn't violate any of the constraints, and we repeat the process of ensuring that the new placement does not break the board accordingly. It is important to note that we have to check if the digit chosen would break the row and/or column and/or subgrid it is sitting in. Therefore, as we follow this, at some point in the puzzle, after some Nth placement, we know that all the placements completed didn't break the board, and we are safe to check if our (N+1)th placement would break the board in only its row, column, and subgrid.

Our recursive function would try digit choices (1 to 9) in each empty cell, and before we place the digit in the cell and do recursion on it accordingly, the function would check that it doesn't break the board with its choice. If it does not break the board, we move on to the next column.

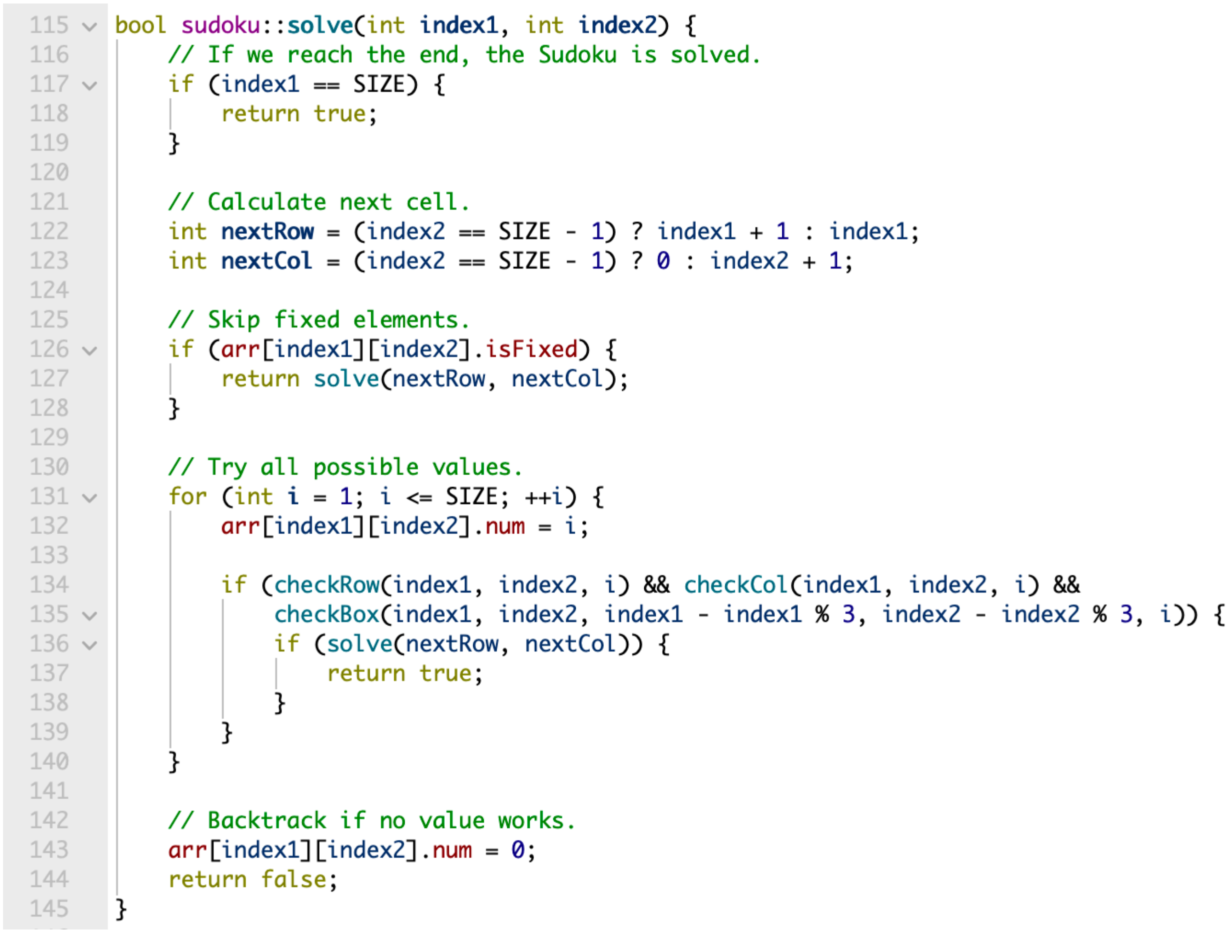
1.4 Algorithms Specifications

This is a recursive algorithm with the 2 indices as the parameters (the current element in terms of its row and column indices), and so we need to start the algorithm with a base case and then a general case.

The base case in this situation is for the puzzle to be fully solved. This will occur when the first index, or the row coordinates to be equal to the size, meaning it has gone out of the bounds of the array of the puzzle. This means the algorithm has filled the array of the puzzle while following the conditions of sudoku, hence has achieved the solution.

The general case (lines 121-144) :

1. Assign a value for the next element (nextRow, nextCol) in the array to use later on. If they have reached the last element in the row (index2 == SIZE -1), we will assign the element in the first column and next row, otherwise we will simply increment the column number keeping the same row number.
2. Check if the current element is fixed in the puzzle or if the algorithm should solve for this element. If it is fixed, it will return a recursive call to solve for the next element assigned in step 1. Otherwise, if it is not fixed, it will continue in the algorithm.
3. Using a for-loop, we will try all possible values and check Sudoku Conditions. If the conditions for all possible numbers are false, it will go out of the for loop and assign a value of 0 (default value of array for empty spaces) to the element and return false to the previous element, meaning it has tried all possible values and none fit the conditions hence it is not solvable with the current placement of numbers. If it's true, it will continue.
4. Check if the next element is solvable, hence recursively calling the function for the next element. If it has reached the base case, it will return true, hence entering the if statement in line 136 and returning true inside. This will lead to a chain of return trues as it recursively goes back hence exiting the recursion with a return true. If the next element's solve call returns false (as shown possible in step 3), it will continue in its for-loop to try all other possible numbers until it finds another one that fits all the conditions. If it still couldn't be found, it will recursively go back trying other possible numbers and returning false until it has reached (0,0), the first call of solve having exited the recursion and return false, meaning the puzzle is not solvable.



```cpp
bool sudoku::solve(int index1, int index2) {
    // If we reach the end, the Sudoku is solved.
    if (index1 == SIZE) {
        return true;
    }

    // Calculate next cell.
    int nextRow = (index2 == SIZE - 1) ? index1 + 1 : index1;
    int nextCol = (index2 == SIZE - 1) ? 0 : index2 + 1;

    // Skip fixed elements.
    if (arr[index1][index2].isFixed) {
        return solve(nextRow, nextCol);
    }

    // Try all possible values.
    for (int i = 1; i <= SIZE; ++i) {
        arr[index1][index2].num = i;

        if (checkRow(index1, index2, i) && checkCol(index1, index2, i) &&
            checkBox(index1, index2, index1 - index1 % 3, index2 - index2 % 3, i)) {
            if (solve(nextRow, nextCol)) {
                return true;
            }
        }
    }

    // Backtrack if no value works.
    arr[index1][index2].num = 0;
    return false;
}
```

1.5 Data Specifications

When creating a sudoku solver, we have two options:

1. Create new sudoku puzzles with different difficulty levels.

   That means that we would have to create a randomly generated puzzle that is solvable. Although this may be slightly challenging, it is doable by inputting a few random numbers and using the solve algorithm from there. However another problem could be that the puzzle could have 2 viable solutions. This can occur since our algorithm will return the first solved puzzle it will happen upon. Therefore, the user can have a valid answer that would not be validated by our solver. Another challenge we would face is how to balance the amount and placement of the numbers we remove such that it would have only 1 solution, and be able to adjust the difficulty level of the puzzle. Difficulty involves many factors and so trying to adjust the difficulty level of the puzzle after creation would have been too complicated.

2. Outsource sudoku puzzles from other sources and read the puzzles from a saved file. This is an easier alternative and so we resorted to it in the end. This method would require more manual labor though; we needed to copy multiple sudoku puzzles from different difficulty levels onto txt files, then create a function to read the puzzles from the txt file correctly, then create a function to randomly choose a puzzle based on the difficulty.

1.6 Experimental Results

To ensure our backtracking algorithm was working correctly, we used it on a number of different sudokus and it gave the correct results. Not only that but we also used the algorithm on an unsolvable sudoku in which the solve function returned false. Throughout this project we experienced many difficulties, but we were able to overcome them in the end. To start with, when we wanted to add a GUI we tried using ‘wxWidgets’ and ‘NAPP GUI’ but we faced problems with it so we resorted to using ‘QT’ instead, since we were familiar with it and have used it before. Another challenge we faced was formatting the sudoku grid. We initially tried using ‘QWidget’ to implement the grid but could not find a way to make the grid lines. As a solution we used background colors to differentiate boxes. We also struggled with the ‘checksolution’ function. We had a missing condition in an if statement when comparing the user’s answers with the correct answer. The condition was if the user left a slot empty, and not having this condition did not give us what was intended when the ‘Check Solution’ button was pressed.

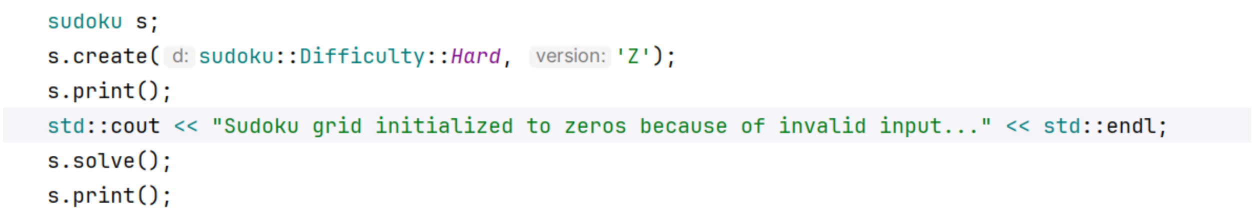
1.7 Analysis and Critique

Analysis of Output

Starting with the output results, they are as expected. Different outputs result from different inputs:

1. The sudoku cannot be accessed, the sudoku grid is by default initialized to zero and an interesting result is saved into the grid when the *solve* function is called: it actually creates its own sudoku grid by the specific order which we expect to occur (see attached).

Invalid input in main function to test case of initialization into zeros:



```
sudoku s;
s.create( d: sudoku::Difficulty::Hard,  version: 'Z');
s.print();
std::cout << "Sudoku grid initialized to zeros because of invalid input..." << std::endl;
s.solve();
s.print();
```

Output (as expected):

```
Wrong Input in generateSoduku function.
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
Sudoku grid initialized to zeros because of invalid input...
1 4 7 2 3 8 5 6 9
2 5 8 1 6 9 3 4 7
3 6 9 4 5 7 1 2 8
4 7 1 3 8 2 6 9 5
5 8 2 6 9 1 4 7 3
6 9 3 5 7 4 2 8 1
7 1 4 8 2 3 9 5 6
8 2 5 9 1 6 7 3 4
9 3 6 7 4 5 8 1 2

Process finished with exit code 0
```

Note how the grid initialized the first column into 1 through 9 and then solved the grid as should.

2. The sudoku presented is unsolvable and thus the algorithm returns false into the *solve* function. Just to test this case, we modified the *solve* function to return some statement of meaning if the sudoku is unsolvable, and here is the output:

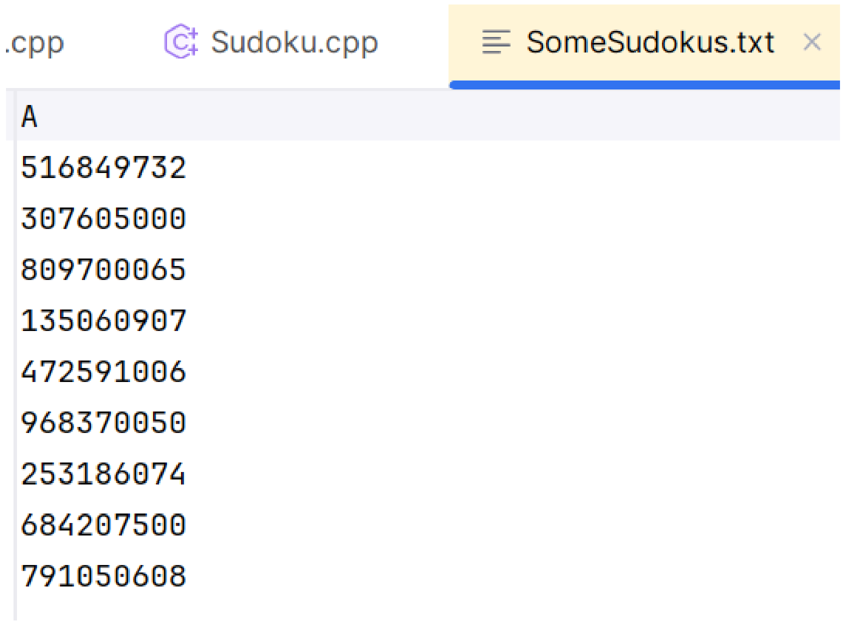
Modified solve function to show a meaningful statement if the sudoku is unsolvable:

```
bool sudoku::solve()
{
    if (solve(0,0))
        return true;

    std::cout<< "Sudoku has no solution..." << std::endl;

    return false;
}
```

The unsolvable sudoku:



```
A
516849732
307605000
809700065
135060907
472591006
968370050
253186074
684207500
791050608
```

The output, as expected:

```
5 1 6 8 4 9 7 3 2
3 0 7 6 0 5 0 0 0
8 0 9 7 0 0 0 6 5
1 3 5 0 6 0 9 0 7
4 7 2 5 9 1 0 0 6
9 6 8 3 7 0 0 5 0
2 5 3 1 8 6 0 7 4
6 8 4 2 0 7 5 0 0
7 9 1 0 5 0 6 0 8
Sudoku grid initialized to zeros because of invalid input...
Sudoku has no solution...
5 1 6 8 4 9 7 3 2
3 0 7 6 0 5 0 0 0
8 0 9 7 0 0 0 6 5
1 3 5 0 6 0 9 0 7
4 7 2 5 9 1 0 0 6
9 6 8 3 7 0 0 5 0
2 5 3 1 8 6 0 7 4
6 8 4 2 0 7 5 0 0
7 9 1 0 5 0 6 0 8
```

Note how the program did not change any of the values despite passing by all of them.

3. The program solves the solvable sudoku using backtracking:

Inserting a valid sudoku and solving it:

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

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

Analysis and Critique of Algorithm

In this course, we have learned how to analyze algorithms and that is to analyze performance in two different domains: size and time. As for the analysis of the size, the algorithm does not involve storing much information. Only temporary values are used for counting and updates are implemented directly on the original “sudoku grid” array. Thus, our algorithm is quite efficient when it comes to the size factor. Now we come to the discussion of the time complexity. The algorithm analyzes the possible values *for each* cell. In most cases, and the average case, would have a normal complexity. In extreme cases, however, the worst case time complexity would depend on the number of empty cells (because the algorithm tries nine different choices for each cell), we would have a complexity of $O(9^n)$, which is a very, very bad time complexity. There is more than one reason we have implemented this algorithm despite the inefficiency it shows: it is easy to implement and easy to understand, it is quite simple in terms of code, and it is utilized by many different programs when finding solutions for sudokus.

1.8 Conclusions

To conclude this report, the Sudoku Game project allowed us to apply our knowledge of data structures that we have learned throughout this course to implement the backtracking algorithm that solves the sudoku. In the backtracking algorithm, recursion was used to go back to previous slots and assign other digits. This only happens only when the algorithm tries to place every digit in a slot and every digit violates the rules of sudoku. To ensure our backtracking algorithm worked perfectly we tried it on multiple sudokus where we knew the answer and on an unsolvable sudoku. In both cases the solve function gave us the expected output. We also applied our knowledge to create a user friendly interface to ensure a smooth experience for the user.

Acknowledgements

A thank you goes out to our families and friends for helping us through writing this report.

Appendix: Implementation Codes

Link to GitHub Repository with Full Implementation:

https://github.com/aylasaleh/CSCE2211-SudokoProject