CS 531 Homework 1 - Report

Ege Alpay
egealpay@sabanciuniv.edu

February 27, 2020

1 The Code

```
bool solve(int **&matrix, std::vector<std::pair<std::pair<int, int>, std::pair<
    int, int> > > &constraints, int size) {
    // Find a empty cell
    std::pair<int, int> emptyCell = findEmptyCell(matrix, size);
    int currentRow = emptyCell.first;
    int currentCol = emptyCell.second;

    // If there is no empty cell, then we are done
    if (currentRow == -1 && currentCol == -1)
        return true;

    for (int value = 1; value <= size; value++) {
        if (doesSatisfyRules(matrix, currentRow, currentCol, value, constraints,
     size)) {
            matrix[currentRow][currentCol] = value;

            if (solve(matrix, constraints, size))
                return true;

            // Invalidate failed
            matrix[currentRow][currentCol] = -1;
        }
    }

    return false;
}
```

Listing 1: Solve method for Futoshiki Puzzle

In Listing 1, you can find the implementation of solve method. First of all, we check the matrix for empty cells by calling $findEmptyCell$ function.

```
std::pair<int, int> findEmptyCell(int **&matrix, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (matrix[i][j] == -1)
                return std::pair<int, int>(i, j);
        }
    }

    return std::pair<int, int>(-1, -1);
```

```
}
```

Listing 2: Method to find empty cell

This method iterates over the matrix, in order to find an empty cell. If the current cell has value of -1, then it is empty and we can return the row index and column index of that cell. If we cannot find any empty cell, we return row index as -1 and column index as -1.

After retrieving the row and column indeces of an empty cell, we check these indeces are valid. If they both equal to -1, then we terminate this function since there are no empty cells in the matrix. If they are not equal to -1, then we have found an empty cell. Next, we have a for loop to try each value to that empty cell. A cell can take values from 1 to the size of the matrix. So first, we start with value 1. We check if we can assign current value to that empty cell by calling $doesSatisfiesRules$ function.

```
bool doesSatisfyRules(int **&matrix, int currentRow, int currentCol, int value,
                      std::vector<std::pair<std::pair<int, int>, std::pair<int,
    int> > > &constraints, int size) {
    return canUseInRow(matrix, currentRow, value, size) &&
           canUseInColumn(matrix, currentCol, value, size) &&
           isSatisfyConstraints(matrix, currentRow, currentCol, value,
    constraints) &&
           isCurrentCellEmpty(matrix, currentRow, currentCol);
}
```

Listing 3: Checking the cell to assign a value

There are 4 rules that we have to control.

1. Check if the value has not used in the current row of the current cell.

```
    bool canUseInRow(int **&matrix, int currentRowIndex, int value, int
    size) {
        int *currentRow = matrix[currentRowIndex];
        for (int i = 0; i < size; i++) {
            if (currentRow[i] == value)
                return false;
        }

        return true;
    }

```

Listing 4: Check the row of the current cell

2. Check if the value has not used in the current column of the current cell.

```
    bool canUseInColumn(int **&matrix, int currentColIndex, int value, int
    size) {
        for (int i = 0; i < size; i++) {
            if (matrix[i][currentColIndex] == value)
                return false;
        }
```

```
6
7           return true;
8       }
9
```

Listing 5: Check the column of the current cell

3. Check if the current cell is empty.

```
1       bool isCurrentCellEmpty(int **&matrix, int currentRow, int currentCol)
        {
2           return matrix[currentRow][currentCol] == -1;
3       }
4
```

Listing 6: Check the current cell is empty

4. Check if assigning the value to current cell does not violate any constraints.

```
1       bool isSatisfyConstraints(int **&matrix, int currentRow, int currentCol
        , int value,
2                                 std::vector<std::pair<std::pair<int, int>, std::
        pair<int, int> > > &constraints) {
3           int numConstraints = constraints.size();
4           for (int i = 0; i < numConstraints; i++) {
5               std::pair<std::pair<int, int>, std::pair<int, int>> constraint
        = constraints[i];
6               std::pair<int, int> start = constraint.first;
7               std::pair<int, int> end = constraint.second;
8
9               if (currentRow == start.first && currentCol == start.second) {
10                  if (value < matrix[end.first][end.second] && matrix[end.
        first][end.second] != -1)
11                      return false;
12              } else if (currentRow == end.first && currentCol == end.second)
         {
13                  if (value > matrix[start.first][start.second] && matrix[
        start.first][start.second] != -1)
14                      return false;
15              }
16          }
17
18          return true;
19      }
20
```

Listing 7: Check constraints

The first three rules are also the main rules of Sudoku. But the major difference between Futoshiki and Suduko, is that Futoshiki puzzle have some constraints between cells. Some cells should have a greater value to its adjacent cell.

If these four rules are satisfied, we assign that value to the current cell. And we call $solve$ function recursively until all cells are filled with a value. When all cells are filled up, $solve$ function will return true, so the if check will also be true, which means that $solve$ function will return true recursively up to the first call of $solve$ function. In some cases, we will not find any value to place to the current cell. In this case, the $solve$ function will return false to the caller. So, if check will fail and we revert the value assignment in the previous call by assignment -1 to that cell. This type of algorithms are called as **Backtracking Algorithms**.

2 Optimization Tryouts

As my first tryout, I have used $short$ variables instead of $integer$ variables. Mainly, I was planning to see some changes in $findEmptyCell$, $canUseInColumn$ and $canUseInRow$ since they have for loops.

Table 1: Execution times with Short

File	Size	Time (Seconds) -O0
input1.txt	4x4	0.00005
input1_2.txt	4x4	0.00003
input2.txt	5x5	0.00045
input2_2.txt	5x5	0.00551
input3.txt	6x6	0.00421
input3_2.txt	6x6	0.06945
input4.txt	7x7	1.45
input4_2.txt	7x7	4.94
input5.txt	8x8	121.62
input5_2.txt	8x8	218.03

I will not go further to measure the run time for -O3 flag since I have observed that by using short, $solve$ function now runs much slower than the integer version.

My second tryout is to change $findEmptyCell$ function from Row Major to Column Major. With this experiment, I am expecting to see some decrease in performance since Row Major is faster than Column Major.

```
std::pair<int, int> findEmptyCell(int **&matrix, int size) {
    for (int i = 0; i < size; i++) { // column
        for (int j = 0; j < size; j++) { // row
            if (matrix[j][i] == -1)
                return std::pair<int, int>(j, i);
        }
    }

```

```
    return std::pair<int, int>(-1, -1);
}
```

Listing 8: Method to find empty cell with Column Major

Table 2: Execution times with Column Major

File	Size	Time (Seconds) -O0
input1.txt	4x4	0.00004
input1_2.txt	4x4	0.00002
input2.txt	5x5	0.00124
input2_2.txt	5x5	0.00032
input3.txt	6x6	0.01456
input3_2.txt	6x6	1.06
input4.txt	7x7	0.26079
input4_2.txt	7x7	0.93599
input5.txt	8x8	2.60
input5_2.txt	8x8	101.65

As a result, I have observed increase in performance, which is the opposite of my expectation. I believe that this is happened since the input sizes are small, cache was able to store all rows, so that cache miss rate has decreased and performance has increased. If we compare Row Major vs Column Major in a very large input, Row Major will have better performance in comparison to Column Major.

3 Execution times

The following execution times are obtained by running the code explained in Section 1.

Table 3: Execution times

File	Size	Time (Seconds) -O0	Time (Seconds) -O3
input1.txt	4x4	0.00005	0.00002
input1_2.txt	4x4	0.00002	0.00001
input2.txt	5x5	0.00034	0.00010
input2_2.txt	5x5	0.00453	0.00129
input3.txt	6x6	0.00344	0.00103
input3_2.txt	6x6	0.05464	0.01661
input4.txt	7x7	1.16	0.33297
input4_2.txt	7x7	3.80	1.10
input5.txt	8x8	89.18	24.59
input5_2.txt	8x8	173.32	50.78

For now, my algorithm works slower in comparison to your implementation, when size becomes larger. (For -O0 flag) One observation is that, with O3 optimization flag, compile time increases but execution time decreases.

4 Compile and Run

For development, I have used CLion IDE to write the code. For compilation and testing the implementation, I have used gcc 9.2.0 on my Linux Mint machine. On Gandalf, gcc 8.2.0 was used.

Following command was used to run the code on Gandalf:

```
1    gcc hw1.cpp -lstdc++
2
```

Listing 9: Run command on Gandalf