Gregory D. Stula
12/7/20
CS325 Fall 2020
HW 8: Portfolio Assignment

Sudoku Ncurses

Sudoku Game Description:
Sudoku is a puzzle game where the goal is to fill a $n^2 * n^2$ grid divided into $n * n$ subgrids with the number between $1 \ldots n^2$ such that each row, columns, and subgrid contains no repeated numbers. The user is presented with a partially filled grid with one unique solution. The standard 9x9 grid typically contains at least 17 filled sections, also known as "hints".

Implementation:
Sudoku Ncurses is implemented in C++ for Linux and has been tested on the flip engr server. The program includes a real-time console-based GUI, a verification algorithm, and a solver algorithm which solves the puzzle.

The user is presented with a hard-coded "easy" level sudoku problem that is represented by a 9x9 two-dimensional array. As the user fills the rest of the board, the game checks to see how many unfilled spaces are left. Once all the spaces are filled, a verification algorithm is run to check to see if the user solved the puzzle. The result of the verification algorithm is outputted to the user and they will see if it is "Solved" or "Invalid". If the grid has yet to be completely filled, they will see the output as "Unsolved". The last 3 pages of this document presents screenshots of the game's UI in action.

GUI and Navigation:
The GUI is implemented via the ncurses library, which was wrapped in a personally written C++ library that makes the relevant portions of the ncurses API easier to use in an idiomatic C++ style. The C++ ncurses wrapper was adopted from a previous ncurses game I created called "Quick Snake" which is available on my github.

The user navigated the board by moving the cursor with the arrow keys. The partially filled hints of the original puzzle are highlighted and cannot be changed. All initially unsolved parts of the board are presented as the '.' character. The user can change any non-highlighted part of the board to a number between 1 and 9 by navigating the cursor to that position and entering a number.

At any time the user can ask the game to solve the puzzle by pressing the 's' key. Similarly, the user can quit the game at any time by pressing the 'q' key.

Verification Algorithm:
The verification algorithm checks if a filled board is valid and solved. While the actual implementation used a hard-coded a 9x9 grid, we will discuss the verification algorithm in terms of variable dimensions for the purposes of analyzing runtime complexity.

The sudoku game is played on a $n^2 * n^2$ board divided into $n * n$ subgrids.

To simplify analysis we will say the board is of $n * n$ dimensions and the board is divided into sub grids of $\sqrt{n} * \sqrt{n}$ dimensions where $\sqrt{n}$ is a whole number.

The board is represented as a two-dimensional array with $n * n$ dimensions.
A *brute force strategy*, where every item in matrix is checked as valid was used.

There are three conditions that need to be checked in the filled $n * n$ grid:

1. That each column has exactly one instance of each of the numbers in the range 1 to n (inclusive).
2. That each row has exactly one instance of each of the numbers in the range 1 to n (inclusive).
3. That each $\sqrt{n} * \sqrt{n}$ subgrid has exactly one instance of each of the numbers in the range 1 to n (inclusive).

The verification algorithm for a generic $n * n$ grid can be represented in pseudo-code as:

```
bool board_is_valid(int matrix[][], int N)
{
    // if row or column is not valid then board is invalid
    for (int i = 0; i < N; i++) {
        if (!row_is_valid(matrix, i, N) && !col_is_valid(matrix, i, N)) {
            return false;
        }
        // check 3x3 grid at multiples of 3
        // if grid is invalid then the board is invalid
        if (i % √N == 0) {
            for (int j = 0; j <= N/2; j += √N) {
                if (!grid_is_valid(matrix, i, j, N)) {
                    return false;
                }
            }
        }
    }
    return true;
}
```

The subroutines called can be written in pseudo-code as:

```
// checks if given column is valid
bool col_is_valid(int matrix[][], int col, int N)
{
    set<int> unique;
    for (int i = 0; i < N; i++) {
        // all the values in a set are unique if insertion fails
        // we have duplicate values in the column
        // and the sudoku solution is invalid
        if (!unique.insert(matrix[i][col]).success) {
            return false;
        }
   }
   return true;
}

// checks if given row is valid
bool sudoku_game::row_is_valid(int matrix[][], int row, int N)
{
    set<int> unique;
    for (int i = 0; i < N; i++) {
        // all the values in a set are unique, so if insetion fails
        // we have duplicate values in the row
        // and the sudoku solution is invalid
        if (!unique.insert(matrix[row][i]).success) {
            return false;
        }
   }
   return true;
}
```

```
// checks if 3x3 grid is valid
bool grid_is_valid(int matrix[][], int row, int col, int N)
{
    set<int> unique;
    for (int i = 0; i < √N; i++) {
        for (int j = 0; j < √N; j++) {
            // all the values in a set are unique, so if insetion fails
            // we have duplicate values in the column
            // and the sudoku solution is grid
            if (!unique.insert(matrix[row + i][col + j]).success) {
                return false;
            }

        }
    }
    return true;
}
```

Each of the subroutines must iterate n^2 times.
We know this following way:

- row_is_valid must check each element of each row of size n and there are n rows
    - Hence, the tight bound of row_is_valid is $O(n^2)$
- col_is_valid must check each element of each column of size n and there are n columns
    - Hence, the tight bound of grid_is_valid is $O(n^2)$
- grid_is_valid must check each element of each subgrid of size $\sqrt{n} * \sqrt{n}$ and there are n subgrids
    - Hence, the tight bound of grid_is_valid is $O(n^2)$

Therefore, the overall tight bound is $O(n^2)$.
So, we can say that we have a polynomial time verification algorithm.

Solving the Sudoku Problem:

Since we have shown we can verify the solution in polynomial time we can say that the Sudoku solving problem is in NP. The problem has been shown to be NP-Complete by Yato and Seta. The paper can be accessed here.

The solver algorithm in the implementation operated on a hard-coded 9x9 grid using a recursive backtracking solution.

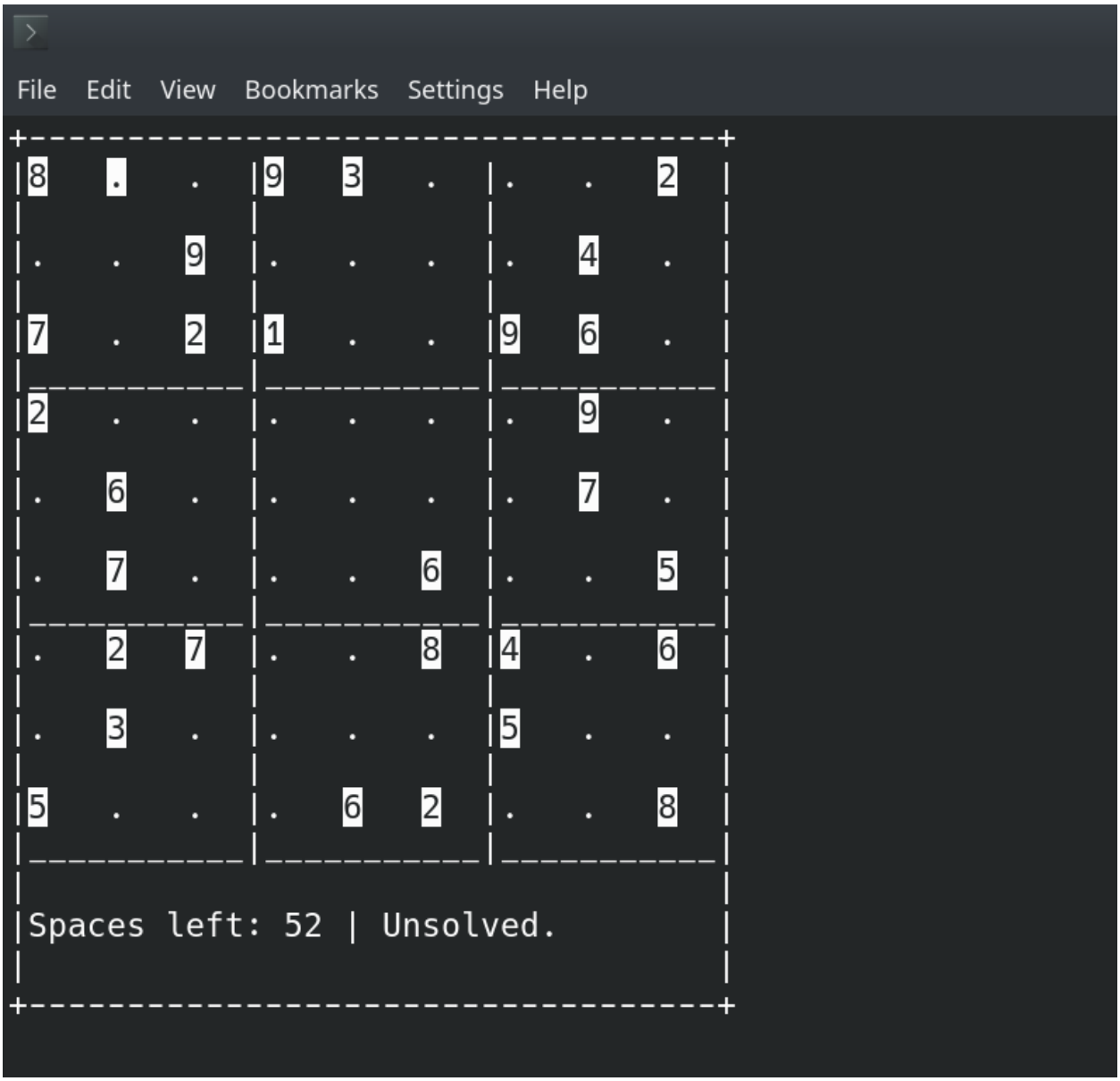
The algorithm for a generic grid with $n * n$ dimensions can be described with the following steps:

1. Check if there are empty spaces on the grid.
2. If there are no empty spaces left the puzzle is solved.
3. Otherwise, get the coordinates of the next empty space.
4. For all the numbers $1...n$, see if the number can be legally inserted at the empty space.
5. If the number can be legally inserted, recursively check if this solves the puzzle.
6. If it does not solve the puzzle remove the number that was inserted and try again.

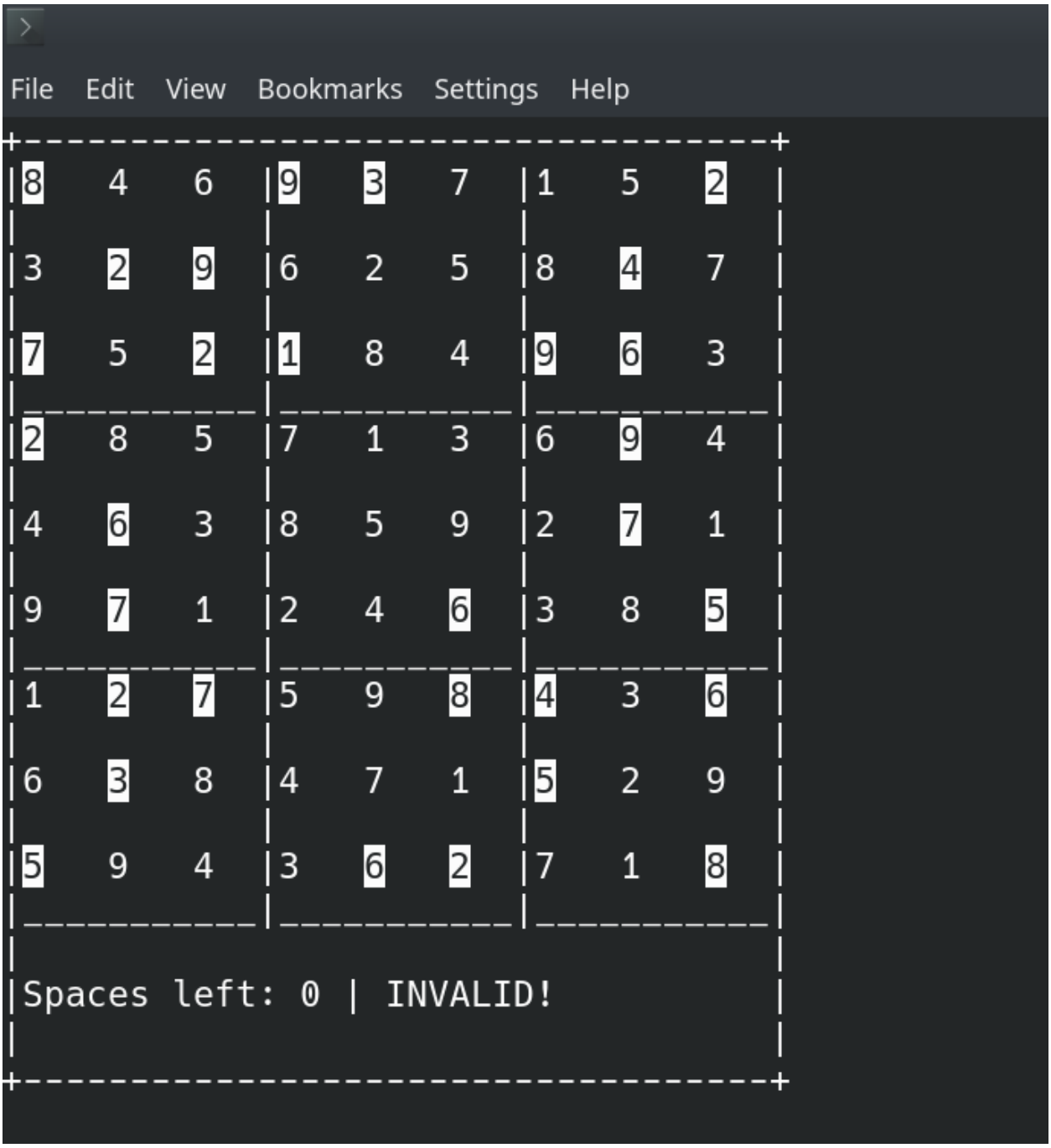
The resulting pseudo-code looks like this:

```
// recursive backtracking algorithm that brute forces the solution
#define EMPTY 0
bool generate_solution(int matrix[][], int N) {

    // game is solved if all positions are filled
    if (!editable_location_exists()) return true;

    // find editable location by reference
    int row, col;
    get_editable_location(row,col);

    // generate chars 1-9 to test
    for (int val = 1; int <= N; val++) {
        // try num in position if move is legal
        if (is_legal(matrix, row, col, val, N)) {
            matrix[row][col] = val;
            // check if solved
            if(generate_solution(matrix, N)) {
                return true;
            }
            // reset to retry if move didn't work
            matrix[row][col] = EMPTY;
        }
    }
    return false;
}
```

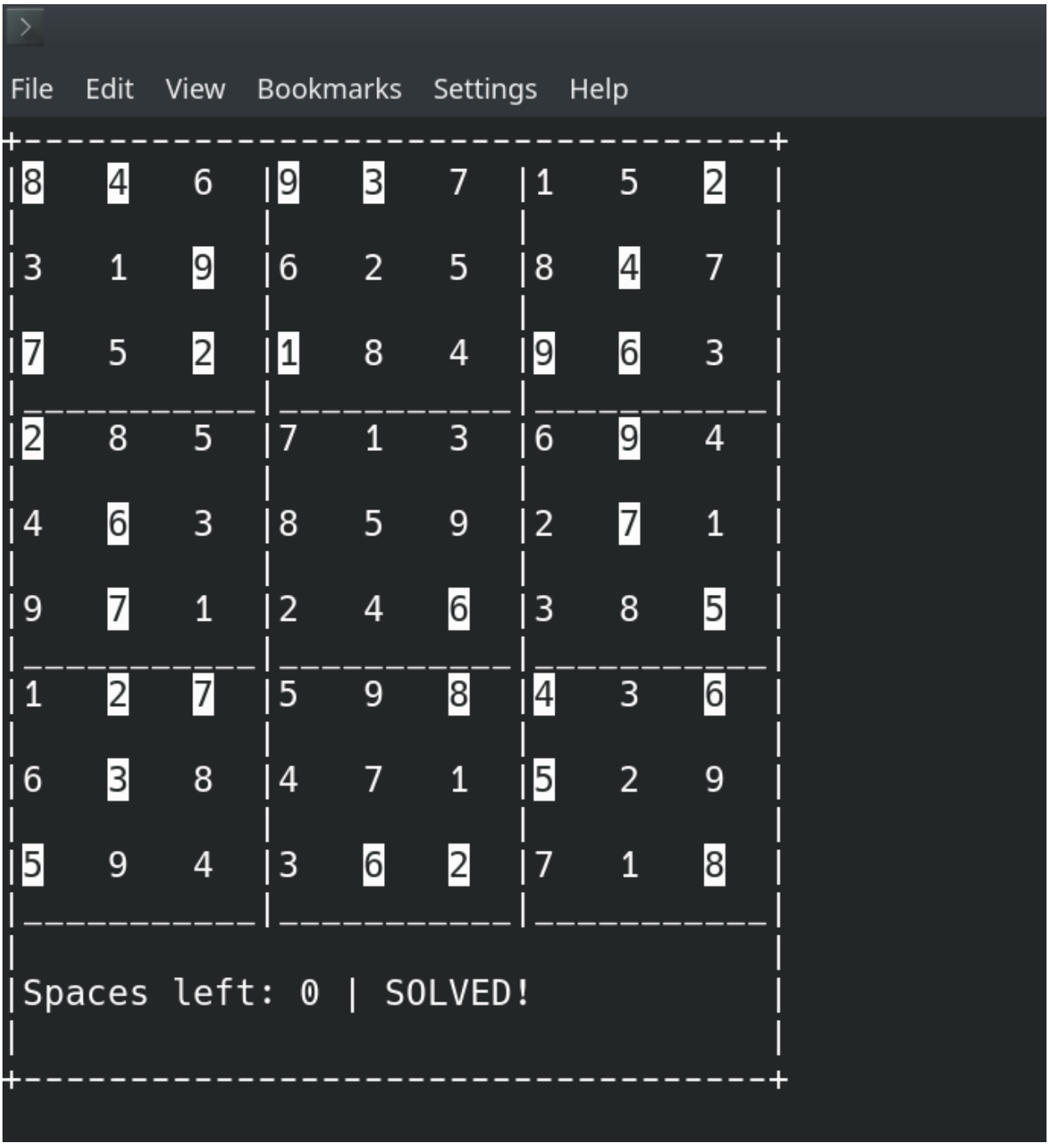
Screenshots



Incomplete and unsolved board.



Complete but invalid board.



Solved and valid board.