EECE 2560 Project 4

Casey Goyette & Eric Concannon

For this project, we first decided the conflict vector approach we were going to take. We decided to take a modified conflict counts approach as it was determined to be generally faster and easier to work with than a conflict list approach (vector for each cell to stating what digits would cause conflicts if placed in the cell) or a standard conflict count approach (vector for each cell stating how many conflicts there are for each digit if placed in the cell). The method we decided to use involved keeping a single Boolean conflict vector for each row, column, and square. This greatly reduced the number of conflict vectors needed. The code for initializing the conflict vectors can be seen in Figure 1 below.

```
void board::initializeConflicts()
{
    for (int i = 0; i < BoardSize; i++)
    {
        for (int j = 0; j < MaxValue; j++)
        {
            row_conflicts[i][j] = checkRow(i,k: j+1);
            column_conflicts[i][j] = checkColumn(column: i, k: j+1);
            square_conflicts[i][j] = checkSquare(square_num: i, k: j+1);
        }
    }
}
```

Figure 1. Conflict vector initialization function

As seen in the code in Figure 1, to initialize the conflict vectors, every row, column, and square were checked to see which of the possible digits they contained. For example, if the first row (Row 0) contained the digit 4, then the 4th index of the conflict vector corresponding to Row 0 would be set to 1, indicating that the digit 4 is in the row. For example, indicating that the digit 4 is present in Row 0, the assignment would look like this: *row_conflicts*[0][3] = 1. Matrices were used to store the conflict vectors so that each row and digit could be easily iterated through. For example, in the statement *row_conflicts*[*i*][*j*], *i* indicates the row being referenced and *j* indicates the digit being referenced (the nth index of the specified row vector where n is the digit being checked).

This method of determining conflict vectors allowed the conflict vectors to be easily updated whenever a value of added or removed from the sudoku puzzle. When a value *x* was added to the puzzle at a position (*i, j*), the x^{th} index of the conflict vectors corresponding to the i^{th} row, the j^{th} column, and the current square were updated to 1 to indicate the digit specified was now in the current row, column, and square. When an element was removed from the puzzle, the same steps were taken, but the conflict vectors were updated to 0 instead of 1. When output, the conflict vectors corresponding to a sudoku board looked like the output shown in Figure 2 on the next page.

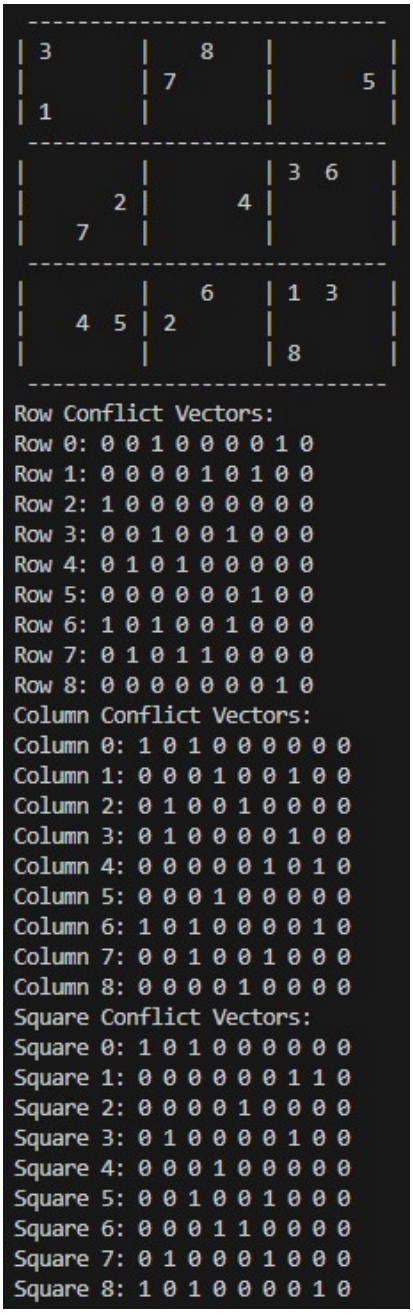

Figure 2. Conflict vector outputs for a corresponding board

With the conflict vectors determined able to be updated, the recursion algorithm for solving the board was then developed. The code for the solving algorithm can be seen in Figure 3 on the next page.

```
bool board::solveBoard(int row, int col)
{
    if (row == 0 && col == 0){iterations = 1;}
    else {iterations += 1;}

    if (row == BoardSize) {return true;}
    else if (col == BoardSize) {return solveBoard(row: row+1, col: 0);}
    else if (value[row][col] != Blank) {return solveBoard(row, col: col+1);}
    else
    {
        for (int k = 1; k <= MaxValue; k++)
        {
            if (checkValid(i: row, j: col, k))
            {
                setCell(i: row, j: col, set_val: k);
                if (solveBoard(row, col: col+1))
                {
                    return true;
                }
                eraseCell(i: row, j: col);
            }
        }
        return false;
    }
}
```

Figure 3. Recursive solving algorithm code

As seen above, the algorithm first increments the variable storing the number of recursive calls. The algorithm then checks to see if it has reached the row after the last row of the board, in which case it returns true to indicate that the algorithm has finished. Next, the algorithm checks to see if it has reached the end of a board column in which case it calls the solve function again on the first column of the next row. If that condition doesn't trigger, the algorithm then checks to see if the current cell is blank. If the current cell is not blank, the algorithm skips this cell and calls the solve function on the next cell in the row. Finally, if all of the previous conditions are not met, it indicates that the current cell is blank and a digit can be placed. The algorithm starts with the first possible digit, 1, and checks to see if there are any conflicts with the digit 1 in the current row, column, or square. If there are conflicts, the algorithm moves to the next possible digit and repeats the process until a valid digit is found. If no valid digit is found, the function returns false to indicate that there is no solution to the board. Once a valid digit is found, the cell is set to the valid digit and the program calls the solve function on the next cell in the column. If this call of the solve function returns true, that indicates that the digit selected was correct and no conflicts were found further on down the line. However, if this call of the solve function returns false, this means that conflicts were found later on down the line and the selected digit is not in the right place. At this point, the algorithm erases the selected digit from the cell and moves on to try the next possible digit. Some input boards with solved outputs can be seen in Figure 4, 5, and 6 on the next few pages.

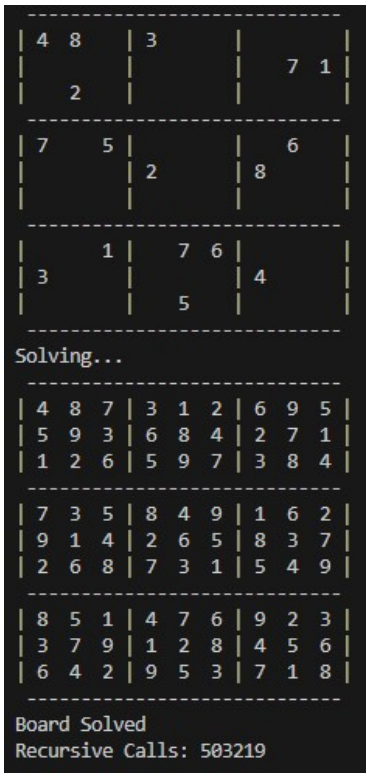

Figure 4. Input/solved board 1

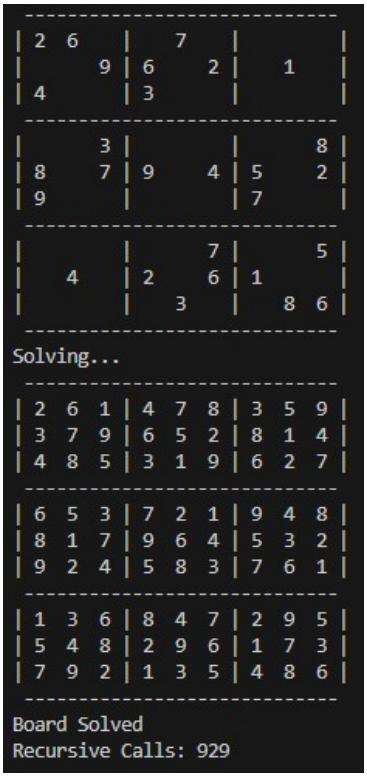

Figure 5. Input/solved board 2

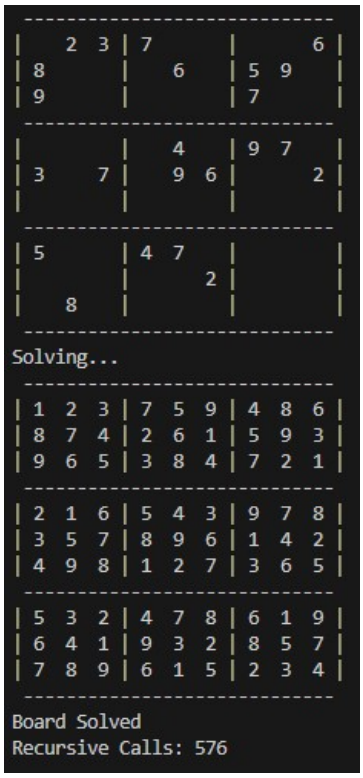

Figure 6. Input/solved board 3