Report: Sudoku Solver with GUI in C

Hardik Gera, McMaster University

4ex

1 Introduction

This report describes a Sudoku solver program implemented in C with a graphical user interface (GUI) using the GTK+ library. The program allows users to input a Sudoku puzzle, solve it automatically, and display the solution.

2 Sudoku Solving Algorithm

The program employs a backtracking algorithm to solve Sudoku puzzles. Backtracking is a recursive technique that systematically explores all possible solutions and backtracks when an invalid configuration is encountered.

3 Code Functionality

3.1 Core Sudoku Logic

Listing 1: Core Sudoku Logic

```
#include statements: Include necessary header files (\texttt{stdio.h, stdbool.h}) for standard input/output and boolean operations.
BOARD_SIZE definition: Define a constant for the Sudoku board size (\texttt{9}).
board array: Declare a 2D integer array \texttt{board} to represent the Sudoku puzzle, initialized with zeros.
display_board function: Prints the current state of the Sudoku board to the console in a formatted way.
locate_unfilled_spot function: Finds the first empty cell (represented by a zero) in the \texttt{board} array and stores its row and column indices in the \texttt{empty\_spot} array.
check_row, check_column, check_subboard functions: These functions check if a given number (\texttt{num}) can be placed at a specific position (\texttt{row, col}) in the Sudoku board without violating the Sudoku rules (no row, column, or sub-block can contain duplicate numbers).
is_valid_placement function: Combines the previous three functions to determine if placing \texttt{num} at (\texttt{row, col}) is a valid move.
solve_puzzle function (recursive): This core function implements the backtracking algorithm. It:
    Counts the number of attempts (commented out in the provided code).
    Finds an empty cell using \texttt{locate\_unfilled\_spot}.
    Base case: If no empty cell is found, it means the solution has been ↩
        reached, so it returns true.
    Recursive case:
        Iterates through possible values (\texttt{1 to 9}) for the empty ↩
            cell.
        For each value, it checks if its a valid placement using \texttt{↩
            is\_valid\_placement}.
        If valid, it places the value in the \texttt{board} and ↩
            recursively calls \texttt{solve\_puzzle} to see if it leads to↩
             a solution.
        If the recursive call returns true, it means a solution is found, ↩
            so it returns true.
        If none of the values lead to a solution, it backtracks by setting↩
             the empty cell back to zero.
        If no solution is found for any valid placement, it returns false.
```

3.2 GUI Implementation with GTK

Listing 2: GUI Implementation with GTK+ (continued)

```
GTK+ initialization: The code initializes the GTK+ library using gtk_init↩
    (&argc, &argv).
Window creation: A new window is created with the title "Sudoku Solver" ↩
    using gtk_window_new.
Window closing: A signal handler g_signal_connect is attached to the ↩
    window to close the application when the user clicks the close button.
Board layout: A gtk_grid_new widget is created to arrange the Sudoku board↩
     elements in a grid layout.
board initialization: The board array is initialized with zeros to ↩
    represent an empty Sudoku puzzle.
Entry widgets: A nested loop iterates to create 81 gtk_entry_new widgets, ↩
    representing the Sudoku cells where users can input numbers. These ↩
    entries are set to have a maximum length of one character to enforce ↩
    single-digit input. They are attached to the grid layout using ↩
    gtk_grid_attach.
Solving the puzzle: The function calls solve_puzzle to solve the Sudoku ↩
    puzzle using the user-provided input.
Displaying the solution: If a solution is found (solve_puzzle returns true↩
    ), the function iterates through the board array and updates the ↩
    content of the entry widgets with the solved values using ↩
    gtk_entry_set_text. It then displays the solved board to the console (↩
    commented out in the provided code).
Handling no solution: If solve_puzzle returns false (no solution found), a↩
```

message indicating "No solution found" is printed to the console (you can modify this to display a message within the GUI).

10 onresetButtonClicked function: This function is called when the "Reset" button is clicked. It iterates through the entry widgets, clearing their text using gtk_entry_set_text and sets the corresponding values in the board array to zero. This effectively resets the Sudoku board to an empty state.

11 Showing widgets and main loop: Finally, the code shows all widgets in the window using gtk_widget_show_all and starts the main GTK+ event loop with gtk_main to handle user interactions.

4 Conclusion

The provided C program demonstrates a functional Sudoku solver with a user-friendly GUI using GTK+. The backtracking algorithm effectively solves Sudoku puzzles, and the GUI allows users to input, solve, and view the results visually. By incorporating input validation and potentially enhancing the GUI with features like highlighting solved cells or difficulty levels, the program can be further improved.

5 Code

The complete C code for the Sudoku solver with GUI is provided in the `Sudoku_Solver_GUI.c` file. Here's a text form code of the file.

```c
#include <stdio.h>
#include <stdbool.h>
#include <gtk/gtk.h>

// Global variables:
#define BOARD_SIZE 9

int board[BOARD_SIZE][BOARD_SIZE];

GtkWidget *entry[BOARD_SIZE][BOARD_SIZE];

void display_board(int board[BOARD_SIZE][BOARD_SIZE])
{
    for (int row = 1; row <= BOARD_SIZE; row++)
    {
        for (int col = 1; col <= BOARD_SIZE; col++)
        {
            if (board[row - 1][col - 1] == 0)
            {
                printf(". ");
            }
            else
            {
                printf("%d ", board[row - 1][col - 1]);
            }
            if (col % 3 == 0)
            {
                printf("| ");
            }
        }
        printf("\n");
        if (row % 3 == 0)
        {
            printf("------+-------+--------\n");
        }
    }
}

void locate_unfilled_spot(int board[BOARD_SIZE][BOARD_SIZE], int empty_spot[2])
{
    for (int row = 0; row < BOARD_SIZE; row++)
    {
        for (int col = 0; col < BOARD_SIZE; col++)
        {
            if (board[row][col] == 0)
            {
                empty_spot[0] = row;
                empty_spot[1] = col;
                return; // Return as soon as an empty spot is found
            }
        }
    }
    // If no empty cell is found, set emptyspot to [-1, -1]
    empty_spot[0] = -1;
    empty_spot[1] = -1;
}

bool check_row(int board[BOARD_SIZE][BOARD_SIZE], int row, int num)
{
    for (int col = 0; col < BOARD_SIZE; col++)
    {
        if (board[row][col] == num)
        {
            return false;
        }
    }
    return true;
}

bool check_column(int board[BOARD_SIZE][BOARD_SIZE], int col, int num)
{
    for (int row = 0; row < BOARD_SIZE; row++)
    {
        if (board[row][col] == num)
        {
            return false;
        }
    }
    return true;
}

bool check_subboard(int board[BOARD_SIZE][BOARD_SIZE], int row, int col, ↩
   int num)
{
    int row_start = (row / 3) * 3;
    int col_start = (col / 3) * 3;
    for (int r = row_start; r < row_start + 3; r++)
    {
        for (int c = col_start; c < col_start + 3; c++)
        {
            if (board[r][c] == num)
            {
                return false;
            }
        }
    }
    return true;
}

bool is_valid_placement(int board[BOARD_SIZE][BOARD_SIZE], int row, int ↩
   col, int num)
{
    return check_row(board, row, num) && check_column(board, col, num) && ↩
       check_subboard(board, row, col, num);
}

bool solve_puzzle(int board[BOARD_SIZE][BOARD_SIZE])
{
    // Count variable renamed for clarity


    int attempts = attempts + 1;
    int empty_spot[2];
    locate_unfilled_spot(board, empty_spot);
    int empty_row = empty_spot[0];
    int empty_col = empty_spot[1];
    if (empty_row == -1)
    {
        printf("Solution found after %d iterations: \n\n", attempts);
        return true;
    }
    else
    {
        for (int guess = 1; guess < 10; guess++)
        {
            if (is_valid_placement(board, empty_row, empty_col, guess))
            {
                board[empty_row][empty_col] = guess;
                if (solve_puzzle(board))
                {
                    return true;
                }

                board[empty_row][empty_col] = 0;
            }
        }
    }
    return false;
}
// called when solve button is clicked
void onsolveButtonClicked(GtkWidget *widget, gpointer data)
{
    for (int row = 0; row < BOARD_SIZE; row++)
    {
        for (int col = 0; col < BOARD_SIZE; col++)
        {
            const char *entry_text = gtk_entry_get_text(GTK_ENTRY(entry[row][col]));
            if (entry_text[0] != '\0')
            {
                board[row][col] = atoi(entry_text);
            }
            else
            {
                board[row][col] = 0;
            }
        }
    }
    printf("Input board:\n");
    display_board(board);

    // Call the solve_puzzle  function to get the solution
    if (solve_puzzle(board))
    {
        for (int row = 0; row < BOARD_SIZE; row++)
        {
            for (int col = 0; col < BOARD_SIZE; col++)
            {
                char buffer[2];
                sprintf(buffer, "%d", board[row][col]);
                gtk_entry_set_text(GTK_ENTRY(entry[row][col]), buffer);
            }
        }
        display_board(board);
    }
    else
    {
        printf("No solution found.\n");
    }
}

// called when Reset button is clicked
void onresetButtonClicked(GtkWidget *widget, gpointer data)
{
    for (int row = 0; row < BOARD_SIZE; row++)
    {
        for (int col = 0; col < BOARD_SIZE; col++)
        {
            gtk_entry_set_text(GTK_ENTRY(entry[row][col]), "");
            board[row][col] = 0;
        }
    }
}

int main(int argc, char *argv[])

{
    GtkWidget *window;
    GtkWidget *board_layout;
    GtkWidget *solve_button;
    GtkWidget *reset_button;

    // Initialize GTK
    gtk_init(&argc, &argv);

    // Create a new window
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "Sudoku Solver");

    //  The following line of code, closes the app (not just the window opened) from terminal, if you click close
    g_signal_connect(window, "destroy", G_CALLBACK(gtk_main_quit), NULL);

    // 20 size distance from the border of outer window.
    gtk_container_set_border_width(GTK_CONTAINER(window), 20);

    // Create a new gridls. We will later put buttons in it.
    board_layout = gtk_grid_new();
    gtk_container_add(GTK_CONTAINER(window), board_layout);

    // Initialize the Sudoku grid with 0 values
    for (int i = 0; i < BOARD_SIZE; i++)
    {
        for (int j = 0; j < BOARD_SIZE; j++)
        {
            board[i][j] = 0;
        }
    }

    for (int i = 0; i < 9; i++)
    {
        for (int j = 0; j < 9; j++)
        {
            // Create an entry
            entry[i][j] = gtk_entry_new();
            gtk_entry_set_max_length(GTK_ENTRY(entry[i][j]), 1);
            gtk_grid_attach(GTK_GRID(board_layout), entry[i][j], j, i, 1, 1);
        }
    }

    /*// Create a button with name "Submit"
    solve_button = gtk_button_new_with_label("Solve");
    reset_button = gtk_button_new_with_label("Reset");
    // if the button is click execute the function button_clicked
    g_signal_connect(solve_button, "clicked", G_CALLBACK(button_clicked), entry);
    g_signal_connect(reset_button, "clicked", G_CALLBACK(button_clicked), entry);
    // attach the button to the grid, and specify the location
    gtk_grid_attach(GTK_GRID(board_layout), solve_button, 3, 9, 1, 1);
    gtk_grid_attach(GTK_GRID(board_layout), reset_button, 4, 9, 1, 1);*/

    solve_button = gtk_button_new_with_label("Solve");
    g_signal_connect(solve_button, "clicked", G_CALLBACK(onsolveButtonClicked), NULL);
    gtk_grid_attach(GTK_GRID(board_layout), solve_button, 0, BOARD_SIZE, BOARD_SIZE, 1);

    reset_button = gtk_button_new_with_label("Reset");
    g_signal_connect(reset_button, "clicked", G_CALLBACK(onresetButtonClicked), NULL);
    gtk_grid_attach(GTK_GRID(board_layout), reset_button, 0, BOARD_SIZE + 1, BOARD_SIZE, 1);

    // It is a loop because the window will be shown and it will stay opened.
    // Show all widgets
    gtk_widget_show_all(window);
    // Start the GTK main loop.
    gtk_main();
    // This is hard coding to receive the "grid"

    return 0;
}
```

6 Sources

1. https://chat.openai.com/ (Used ChatGPT for reference, script writing, understanding GUI, and some functions.)

2. https://www.geeksforgeeks.org/sudoku-backtracking-7/ (For reference on Sudoku backtracking.)

3. https://github.com/coder-zs-cse/Sudoku-Mini-Project (Referenced for understanding Sudoku implementation.)

4. https://codereview.stackexchange.com/questions/243087/made-a-sudoku-solver-with-basic-gui-in-c (GUI reference.)