cs280s21-b.sg

Grade

Reviewed on Tuesday, March 9, 2021, 1:21 AM by Automatic grade
grade: 84.62 / 100.00

Assessment report[-]
[+]**Failed tests**
[+]**Test 10: board4-3**
[+]**Test 13: board5-2**
[+]**Summary of tests**

Submitted on Tuesday, March 9, 2021, 1:18 AM (Download)

Sudoku.h

```
1  //---------------------------------------------------------------------------
2  #ifndef SUDOKUH
3  #define SUDOKUH
4  //---------------------------------------------------------------------------
5  #include <cstddef> /* size_t */
6
7  //! The Sudoku class
8  class Sudoku
9  {
10   public:
11     //! Used by the callback function
12     enum MessageType
13     {
14       MSG_STARTING,      //!< the board is setup, ready to go
15       MSG_FINISHED_OK,   //!< finished and found a solution
16       MSG_FINISHED_FAIL, //!< finished but no solution found
17       MSG_ABORT_CHECK,   //!< checking to see if algorithm should continue
18       MSG_PLACING,       //!< placing a symbol on the board
19       MSG_REMOVING       //!< removing a symbol (back-tracking)
20     };
21
22     //! 1-9 for 9x9, A-P for 16x16, A-Y for 25x25
23     enum SymbolType {SYM_NUMBER, SYM_LETTER};
24
25     //! Represents an empty cell (the driver will use a . instead)
26     const static char EMPTY_CHAR = ' ';
27
28     //! Implemented in the client and called during the search for a solution
29     typedef bool (*SUDOKU_CALLBACK)
30       (const Sudoku& sudoku, // the gameboard object itself
31        const char *board,    // one-dimensional array of symbols
32        MessageType message,  // type of message
33        size_t move,          // the move number
34        unsigned basesize,    // 3, 4, 5, etc. (for 9x9, 16x16, 25x25, etc.)
35        unsigned index,       // index (0-based) of current cell
36        char value            // symbol (value) in current cell
37       );
38
39     //! Statistics as the algorithm works
40     struct SudokuStats
41     {
42       int basesize;      //!< 3, 4, 5, etc.
43       int placed;        //!< number of valid values the algorithm has placed
44       size_t moves;      //!< total number of values that have been tried
45       size_t backtracks; //!< total number of times the algorithm backtracked
46
47       //!< Default constructor
48       SudokuStats() : basesize(0), placed(0), moves(0), backtracks(0) {}
49     };
50
51     // Constructor
52     Sudoku(int basesize, SymbolType stype = SYM_NUMBER,
53         SUDOKU_CALLBACK callback = 0);
54
55     // Destructor
56     ~Sudoku();
57
58     // The client (driver) passed the board in the values parameter
59     void SetupBoard(const char *values, int size);
60
61     // Once the board is setup, this will start the search for the solution
62     void Solve();
63
64     // For debugging with the driver
65     const char *GetBoard() const;
66     SudokuStats GetStats() const;
67
68   private:
69   // Other private data members or methods...
70
71     int board_width;
72     int board_size;
73
74     char* board_;
75     SudokuStats stats_;
76
77     SymbolType stype_;
78     SUDOKU_CALLBACK callback_;
79
80     bool place_value(int value);
81     bool Conflict(int index, char value);
82 };
83
84 #endif  // SUDOKUH
85
```

Sudoku.cpp

```cpp
/******************************************************************************/
/*!
\file:      Sudoku.cpp
\author:    Goh Wei Zhe, weizhe.goh, 440000119
\par email: weizhe.goh\@digipen.edu
\date:      March 8, 2021
\brief      To implement a simple recursive algorithm to solve a sudoku puzzle.

Copyright (C) 2021 DigiPen Institute of Technology.
Reproduction or disclosure of this file or its contents without the
prior written consent of DigiPen Institute of Technology is prohibited.
*/
/******************************************************************************/
#include "Sudoku.h"

/******************************************************************************/
/*!
\fn     Sudoku::Sudoku(int basesize, SymbolType stype, SUDOKU_CALLBACK callback)
        :stype_{stype}, callback_{callback}

\brief  Constructor for sudoku
*/
/******************************************************************************/
Sudoku::Sudoku(int basesize, SymbolType stype, SUDOKU_CALLBACK callback)
:stype_{stype}, callback_{callback}
{
    stats_.basesize = basesize;

    board_width = basesize * basesize;
    board_size = board_width * board_width;
}

/******************************************************************************/
/*!
\fn     Sudoku::~Sudoku()

\brief  Destructor for sudoku, delete board
*/
/******************************************************************************/
Sudoku::~Sudoku()
{
    delete[] board_;
}

/******************************************************************************/
/*!
\fn     const char* Sudoku::GetBoard() const

\brief  Gettor for sudoku board

\return Returns char array of values within the board
*/
/******************************************************************************/
const char* Sudoku::GetBoard() const
{
    return board_;
}

/******************************************************************************/
/*!
\fn     Sudoku::SudokuStats Sudoku::GetStats() const

\brief  Gettor for sudoku statistics

\return Returns data structure SudokuStats for sudoku board
*/
/******************************************************************************/
Sudoku::SudokuStats Sudoku::GetStats() const
{
    return stats_;
}

/******************************************************************************/
/*!
\fn     void Sudoku::SetupBoard(const char* values, int size)

\brief   Set up sodoku board with specific board values and board size
*/
/******************************************************************************/
void Sudoku::SetupBoard(const char* values, int size)
{
    board_ = new char[size];

    //Set board values to be empty or filled with values;
    for(int i = 0; i < size; ++i)
        board_[i] = (values[i] == '.') ? EMPTY_CHAR : values[i];
}

/******************************************************************************/
/*!
\fn     void Sudoku::Solve()

\brief  Attempts to solve the sudoku board
*/
/******************************************************************************/
void Sudoku::Solve()
{
    //When you start the algorithm (the client calls Solve),
    //you will send MSG_STARTING.
    callback_(*this, board_, MSG_STARTING, stats_.moves,
    stats_.basesize, 0, stype_);

    int value = 0;

    if(place_value(value))
    //If, after placing a value you have filled the board, you will send
    //MSG_FINISHED_OK and terminate the search.
```

```
        callback_(*this, board_, MSG_FINISHED_OK, stats_.moves,
            stats_.basesize, 0, stype_);
    else
    //If you do not find a solution after exhaustively checking, you will send
    //MSG_FINISHED_FAIL.
        callback_(*this, board_, MSG_FINISHED_FAIL, stats_.moves,
            stats_.basesize, 0, stype_);
}

/*****************************************************************************/
/*!
\fn     bool Sudoku::place_value(int index)

\brief  Recursive function that place values cell by cell till board is
        completed or deem unsolvable.

\return Returns true if able to place a value without conflicts. Else, return
        false.
*/
/*****************************************************************************/
bool Sudoku::place_value(int index)
{
    //return if index is at end of sudoku board
    if(index == board_size)
        return true;

    //if board position is filled with other values, move to next index
    if(board_[index] != EMPTY_CHAR)
        return place_value(index + 1);

    char value;

    //set value type if its number or letter
    if(stype_ == SymbolType::SYM_NUMBER)
        value = '1';
    else
        value = 'A';

    for(int i = 0; i < board_width; ++i)
    {
        //You will send MSG_ABORT_CHECK immediately before you place a value or
        //remove a value. If this call returns true, you will terminate
        //the search.
        if(callback_(*this, board_, MSG_ABORT_CHECK, stats_.moves,
            stats_.basesize, index, value))
            return false;

        //Place value onto board
        board_[index] = value;

        //Increment moves and place count
        stats_.moves++;
        stats_.placed++;

        //After you place a value on the board, you will send MSG_PLACING.
        callback_(*this, board_, MSG_PLACING, stats_.moves,
            stats_.basesize, index, value);

        //If if there is conflict
        if(!Conflict(index, value))
        {
            //Go to next index if there is no conflict
            if(place_value(index + 1))
                return true;

            //if fail to place value, need to increment backtrack count
            stats_.backtracks++;
        }

        //if conflict, remove value by setting board index back to empty
        //decrement number of place count
        board_[index] = EMPTY_CHAR;
        stats_.placed--;

        //After removing a value from the board, you will send MSG_REMOVING.
        callback_(*this, board_, MSG_REMOVING, stats_.moves,
            stats_.basesize, index, value);

        //increment to next number or letter
        value++;
    }

    return false;
}

/*****************************************************************************/
/*!
\fn     bool Sudoku::Conflict(int index, char value)

\brief  Function to check if value in cell has conflicts in row, in column or
        within box.

\return Returns true there is same value within row, column or in box. Else,
        return false.
*/
/*****************************************************************************/
bool Sudoku::Conflict(int index, char value)
{
    //get row as x-axis and column as y-axis
    int x = index % board_width;
    int y = index / board_width;

    int row_start = y * board_width;

    //check row
    for(int i = row_start; i < row_start + board_width; ++i)
    {
```

```
215         if(i == index)
216             continue;
217
218         if(board_[i] == value)
219             return true;
220     }
221
222     //check column
223     for(int i = 0; i < board_width; ++i)
224     {
225         int curr_pos = i * board_width + x;
226
227         if(index == curr_pos)
228             continue;
229
230         if(board_[curr_pos] == value)
231             return true;
232     }
233
234     //check box
235     int startX = x - x % stats_.basesize;
236     int startY = y - y % stats_.basesize;
237
238     for(int i = 0; i < stats_.basesize; ++i)
239     {
240         for(int j = 0; j < stats_.basesize; ++j)
241         {
242             int curr_pos = ((startY + i) * board_width) + startX + j;
243
244             if(index == curr_pos)
245                 continue;
246
247             if(board_[curr_pos] == value)
248                 return true;
249         }
250     }
251
```

VPL

◄ Assignment 2: B List

Jump to...

Assignment 3: AVL Trees ►