Backtracking | Set 7 (Sudoku)

Given a partially filled 9×9 2D array 'grid[9][9]', the goal is to assign digits (from 1 to 9) to the empty cells so that every row, column, and subgrid of size 3×3 contains exactly one instance of the digits from 1 to 9.

3		6	5		8	4		
5	2							
	8	7					3	1
		3		1			8	
9			8	6	3			5
	5			9		6		
1	3					2	5	
							7	4
		5	2		6	3		

Naive Algorithm

The Naive Algorithm is to generate all possible configurations of numbers from 1 to 9 to fill the empty cells. Try every configuration one by one until the correct configuration is found.

Backtracking Algorithm

Like all other Backtracking problems, we can solve Sudoku by one by one assigning numbers to empty cells. Before assigning a number, we check whether it is safe to assign. We basically check that the same number is not present in current row, current column and current 3X3 subgrid. After checking for safety, we assign the number, and recursively check whether this assignment leads to a solution or not. If the assignment doesn't lead to a solution, then we try next number for current empty cell. And if none of number (1 to 9) lead to solution, we return false.

```
Find row, col of an unassigned cell
If there is none, return true
For digits from 1 to 9
  a) If there is no conflict for digit at row,col
      assign digit to row,col and recursively try fill in rest of grid
  b) If recursion successful, return true
  c) Else, remove digit and try another
If all digits have been tried and nothing worked, return false
```

Following is C++ implementation for Sudoku problem. It prints the completely filled grid as output.

```
// A Backtracking program  in C++ to solve Sudoku problem
#include <stdio.h>

// UNASSIGNED is used for empty cells in sudoku grid
```

```
#define UNASSIGNED 0

// N is used for size of Sudoku grid. Size will be NxN
#define N 9

// This function finds an entry in grid that is still unassigned
bool FindUnassignedLocation(int grid[N][N], int &row, int &col);

// Checks whether it will be legal to assign num to the given row,col
bool isSafe(int grid[N][N], int row, int col, int num);

/* Takes a partially filled-in grid and attempts to assign values to
  all unassigned locations in such a way to meet the requirements
  for Sudoku solution (non-duplication across rows, columns, and boxes) */
bool SolveSudoku(int grid[N][N])
{
    int row, col;

    // If there is no unassigned location, we are done
    if (!FindUnassignedLocation(grid, row, col))
       return true; // success!

    // consider digits 1 to 9
    for (int num = 1; num <= 9; num++)
    {
        // if looks promising
        if (isSafe(grid, row, col, num))
        {
            // make tentative assignment
            grid[row][col] = num;

            // return, if success, yay!
            if (SolveSudoku(grid))
                return true;

            // failure, unmake & try again
            grid[row][col] = UNASSIGNED;
        }
    }
    return false; // this triggers backtracking
}

/* Searches the grid to find an entry that is still unassigned. If
   found, the reference parameters row, col will be set the location
   that is unassigned, and true is returned. If no unassigned entries
   remain, false is returned. */
bool FindUnassignedLocation(int grid[N][N], int &row, int &col)
{
    for (row = 0; row < N; row++)
        for (col = 0; col < N; col++)
            if (grid[row][col] == UNASSIGNED)
                return true;
    return false;
}

/* Returns a boolean which indicates whether any assigned entry
   in the specified row matches the given number. */
bool UsedInRow(int grid[N][N], int row, int num)
{
    for (int col = 0; col < N; col++)
        if (grid[row][col] == num)
            return true;
    return false;
}

/* Returns a boolean which indicates whether any assigned entry
   in the specified column matches the given number. */
bool UsedInCol(int grid[N][N], int col, int num)
{
    for (int row = 0; row < N; row++)
        if (grid[row][col] == num)
            return true;
    return false;
}
```

```
/* Returns a boolean which indicates whether any assigned entry
   within the specified 3x3 box matches the given number. */
bool UsedInBox(int grid[N][N], int boxStartRow, int boxStartCol, int num)
{
    for (int row = 0; row < 3; row++)
        for (int col = 0; col < 3; col++)
            if (grid[row+boxStartRow][col+boxStartCol] == num)
                return true;
    return false;
}

/* Returns a boolean which indicates whether it will be legal to assign
   num to the given row,col location. */
bool isSafe(int grid[N][N], int row, int col, int num)
{
    /* Check if 'num' is not already placed in current row,
       current column and current 3x3 box */
    return !UsedInRow(grid, row, num) &&
           !UsedInCol(grid, col, num) &&
           !UsedInBox(grid, row - row%3 , col - col%3, num);
}

/* A utility function to print grid  */
void printGrid(int grid[N][N])
{
    for (int row = 0; row < N; row++)
    {
       for (int col = 0; col < N; col++)
             printf("%2d", grid[row][col]);
        printf("\n");
    }
}

/* Driver Program to test above functions */
int main()
{
    // 0 means unassigned cells
    int grid[N][N] = {{3, 0, 6, 5, 0, 8, 4, 0, 0},
                      {5, 2, 0, 0, 0, 0, 0, 0, 0},
                      {0, 8, 7, 0, 0, 0, 0, 3, 1},
                      {0, 0, 3, 0, 1, 0, 0, 8, 0},
                      {9, 0, 0, 8, 6, 3, 0, 0, 5},
                      {0, 5, 0, 0, 9, 0, 6, 0, 0},
                      {1, 3, 0, 0, 0, 0, 2, 5, 0},
                      {0, 0, 0, 0, 0, 0, 0, 7, 4},
                      {0, 0, 5, 2, 0, 6, 3, 0, 0}};
    if (SolveSudoku(grid) == true)
          printGrid(grid);
    else
         printf("No solution exists");

    return 0;
}
```

Run on IDE

Output:

```
3 1 6 5 7 8 4 9 2
5 2 9 1 3 4 7 6 8
4 8 7 6 2 9 5 3 1
2 6 3 4 1 5 9 8 7
9 7 4 8 6 3 1 2 5
8 5 1 7 9 2 6 4 3
1 3 8 9 4 7 2 5 6
6 9 2 3 5 1 8 7 4
7 4 5 2 8 6 3 1 9
```

References:
http://see.stanford.edu/materials/icspacs106b/H19-RecBacktrackExamples.pdf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Company Wise Coding Practice Topic Wise Coding Practice

79 Comments Category: Backtracking Tags: Backtracking

Related Posts:

- Find shortest safe route in a path with landmines
- Longest Possible Route in a Matrix with Hurdles
- Match a pattern and String without using regular expressions
- Find Maximum number possible by doing at-most K swaps
- Find paths from corner cell to middle cell in maze
- Find if there is a path of more than k length from a source
- Fill two instances of all numbers from 1 to n in a specific way
- Backtracking | Set 8 (Solving Cryptarithmetic Puzzles)

(Login to Rate and Mark)

3.5 Average Difficulty : **3.5/5.0**
Based on **23** vote(s)

Add to TODO List
Mark as DONE

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

79 Comments GeeksforGeeks 1 Login

Recommend 2 Share Sort by Newest