18CSC204J- Design and Analysis of Algorithms
A Project Report on
<<SUDOKU SOLVER>>

Submitted in partial fulfillment of the requirement for the IV semester

of

BACHELOR OF TECHNOLOGY

IN

DESIGN AND ANALYSIS OF ALGORITHM

Submitted By:

<<Diviij Verma>> (RA2011029010048)
<<Yashu Youwaraj>> (RA2011029010062)

Under the supervision of

Dr. Balasaraswathi R

(Asst. Professor)

DEPARTMENT OF COMPUTING AND TECHNOLOGY,

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

SESSION – 2022

SRMI INSTITUTE OF SCIENCE AND TECHNOLOGY

S.R.M. NAGAR, KATTANKULATHUR – 603203

BONAFIDE CERTIFICATE

Register No. RA2011029010048

Certified to be bonafide record of the work done by <<Diviij Verma>> of Computer Science and Engineering with specialization in Computer Networking, B.Tech degree course in the practical 18CSC204J – Design and Analysis of Algorithm in SRM Institute of Science and Technology, Kattankulathur during the academic year 2021-22.

Date: **Lab Incharge**

TABLE OF CONTENTS

CONTRIBUTION TABLE

NAME	REG. NO.	CONTRIBUTION
Diviij verma	RA2011029010048	Documentaion and Coding
Yashu Youwaraj	RA2011029010062	Algorithm Desing and Coding

<<SUDOKU SOLVER>>

<<Diviij verma>> & << Yashu Youwaraj >>
Department of Engineering
SRM Institute of Science and Technology
Chennai-603203

DECLARATION

I, hereby declare that the work presented in this dissertation entitled "**Sudoku solver**" has been done by me and my team, and this dissertation embodies my own work.

PREFACE

This project report gives us the brief working of a Sudoku Solver using C language. Sudoku Solver is a real life application/scenario based project . This project report gives us the detailed understanding of the working of the code as well as which kind of algorithm along with why is it that we are going only with that approach. A basic leyman can also understand this report as it is very simple and user friendly ,the code itself is very simple to understand.

Abstract

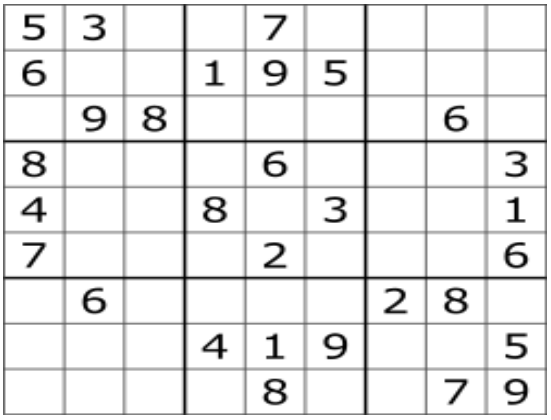
Sudoku is a puzzle in which missing numbers are to be filled into a 9 by 9 grid of squares which are subdivided into 3 by 3 boxes so that every row, every column, and every box contains the numbers 1 through 9.In classic sudoku, the objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids that compose the grid (also called "boxes", "blocks", or "regions") contain all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a single solution

WHAT IS SUDOKU?

Sudoku is a logic-based, combinatorial number-placement puzzle. In classic Sudoku, the objective is to fill a 9 × 9 grid with digits so that each column, each row, and each of the nine 3 × 3 subgrids that compose the grid (also called "boxes", "blocks", or "regions") contain all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a single solution.

French newspapers featured variations of the Sudoku puzzles in the 19th century, and the puzzle has appeared since 1979 in puzzle books under the name Number Place. However, the modern Sudoku only began to gain widespread popularity in 1986 when it was published by the Japanese puzzle company Nikoli under the name Sudoku,

meaning "single number". It first appeared in a U.S. newspaper, and then The Times (London), in 2004, thanks to the efforts of Wayne Gould, who devised a computer program to rapidly produce unique puzzles.



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

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

WHAT IS A SUDOKU SOLVER?

Sudoku Solver program or application is a simple program in c which uses any language (preferred language is c in this project) to predict /in fact getting the most feasible output for the incomplete grid which was given as the input.The input given in the sudoku solver is a partially complete sudoku question which the user can give as an input either on his own or from a previous question and get the desired output for it .The output is an n*n matrix which is a proper complete sudoku .The user can use the sudoku solver to basically cross check whether their answer is matching or not.
Keeping in mind sudoku solver isn't a game ,it's a real time problem solver which brings the final output .The user cant and doesn't have the option to simultaneously update the sudoku .

STRATEGY USED

Backtracking

Backtracking is an algorithmic technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point in time (by time, here, is referred to the time elapsed till reaching any level of the search tree). Backtracking can also be said as an

improvement to the brute force approach. So basically, the idea behind the backtracking technique is that it searches for a solution to a problem among all the available options. Initially, we start the backtracking from one possible option and if the problem is solved with that selected option then we return the solution else we backtrack and select another option from the remaining available options. There also might be a case where none of the options will give you the solution and hence we understand that backtracking won't give any solution to that particular problem. We can also say that backtracking is a form of recursion. This is because the process of finding the solution from the various option available is repeated recursively until we don't find the solution or e reach the final state. So we can conclude that backtracking at every step eliminates those choices that cannot give us the solution and proceeds to those choices that have the potential of taking us to the solution.

How to determine if a problem can be solved using Backtracking?

Generally, every constraint satisfaction problem which has clear and well-defined constraints on any objective solution, that incrementally builds candidate to the solution and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution, can be solved by Backtracking. However, most of the problems that are discussed, can be solved using other known algorithms like *Dynamic Programming* or *Greedy Algorithms* in logarithmic, linear, linear-logarithmic time complexity in order of input size, and therefore, outshine the backtracking algorithm in every respect (since backtracking algorithms are generally exponential in both and space). However, a few problems still remain, that only have backtracking algorithms to solve them until now.
Consider a situation that you have three boxes in front of you and only one of them has a gold coin in it but you do not know which one. So, in order to get the coin, you will have to open all of the boxes one by one. You will first check the first box, if it does not contain the coin, you will have to close it and check the second box and so

on until you find the coin. This is what backtracking is, that is solving all sub- problems one by one in order to reach the best possible solution.

Pseudo Code for Backtracking :

Recursive backtracking solution.

```
void findSolutions(n, other params) : if
    (found a solution) :
        solutionsFound = solutionsFound + 1;
        displaySolution();
        if (solutionsFound >= solutionTarget) :
            System.exit(0);
        return

    for (val = first to last) : if
        (isValid(val, n)) :
            applyValue(val, n); findSolutions(n+1,
            other params); removeValue(val, n);
```

Backtracking in Sudoku Solver

Like all other Backtracking problems, Sudoku can be solved by one by one assigning numbers to empty cells. Before assigning a number, check whether it is safe to assign. Check that the same number is not present in the current row, current column and current 3X3 subgrid. After checking for safety, 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 try the next number for the current empty cell. And if none of the number (1 to 9) leads to a solution, return false and print no solution exists.

Algorithm:

1. Create a function that checks after assigning the current index the grid becomes unsafe or not. Keep Hashmap for a row, column and boxes. If any number has a frequency greater than 1 in the hashMap return false else return true; hashMap can be avoided by using loops.

2. Create a recursive function that takes a grid.

3. Check for any unassigned location. If present then assign a number from 1 to 9, check if assigning the number to current index makes the grid unsafe or not, if safe then recursively call the function for all safe cases from 0 to 9. if any recursive call returns true, end the loop and return true. If no recursive call returns true then return false.

4. If there is no unassigned location then return true.

Analysis – Time and Space Complexity

1. Time Complexity

O(n ^ m) where n is the number of possibilities for each square (i.e., 9 in classic Sudoku) and m is the number of spaces that are blank.

The problem can be designed for a grid size of N*N where N is a perfect square. For such an N, let M = N*N, the recurrence equation can be written as

T(M) = 9*T(M-1) + O(1)

where T(N) is the running time of the solution for a problem size of N. Solving this recurrence will yield, O(9^M).

This can be seen by working backwards from only a single empty spot. If there is only one empty spot, then you have n possibilities and you must work through all of them in the worst case. If there are two empty spots, then you must work through n possibilities for the first spot and n possibilities for the second spot for each of the possibilities for the first spot. If there are three spots, then you must work through n possibilities for the first spot. Each of those possibilities will yield a puzzle with two empty spots that now have n^2 possibilities.

You may also say that this algorithm performs a depth-first search through the possible solutions. Each level of the graph represents the choices for a single square. The depth of the graph is the number of squares that need to be filled. With a branching factor of n and a depth of m, finding a solution in the graph has a worst- case performance of O(n ^ m).

2. Space Complexity

it’s the recursion stack that is used as an auxiliary space which is N*N step deep. Remember we need to fill in 81 cells in a 9*9 sudoku and at each level, only one cell is filled. So, space complexity would be **O(M)**.

Pseudo Code for Sudoku Solver

```
void solveSudoku(char[][] board) {
    solve(board)
}
// Utility function
bool solve(char[][] board) {
    for (int r = 0 to r < 9) {
        for (int c = 0 to c < 9) { if
            (board[r][c] == '.') {
                for (char d = '1'; d <= '9'; d++) {
                    if (isValid(board, r, c, d)) {
                        board[r][c] = d
                         if (solve(board))
                             return true
                         board[r][c] = '.'
                    }
                }
                return false
            }
        }
    }
    return true
}
bool isValid(char[][] board, int r, int c, char d) { for
    (int row = 0 to row < 9)
        if (board[row][c] == d)
            return false
    for (int col = 0 to col < 9) if
        (board[r][col] == d)
            return false;
```

```
    for (int row = (r / 3) * 3 to row < (r / 3 + 1) * 3) for
        (int col = (c / 3) * 3 to col < (c / 3 + 1) * 3)
            if (board[row][col] == d) return
                false
    return true
}
```

Program for Sudoku solver

/*The following program is an implementation of a Sudoku Solver in C.
Sudoku is a 9*9 grid in which each row,each column and each 3*3 grid contains all numbers from 1 to 9 only once.
The program uses backtracking approach to solve the sudoku. There is a recursive function to solve the sudoku.
*/

#include <stdio.h> #include
<conio.h>

int sudoku[9][9]; // The array which stores entries for the sudoku void
solvesudoku(int, int);

int checkrow(int row, int num)
{
	// This function checks whether we can put the number(num) in the row(row) of the Sudoku or not
	int column;
	for (column = 0; column < 9; column++)
	{
		if (sudoku[row][column] == num)
		{
			return 0; // If the number is found already present at certain location we return zero
		}
	}
	return 1; // If the number is not found anywhere we return 1
}

int checkcolumn(int column, int num)
{
	// This function checks whether we can put the number(num) in the column(column) of the Sudoku or not
	int row;
	for (row = 0; row < 9; row++)
	{
		if (sudoku[row][column] == num)

```
                {
                        return 0; // If the number is found already present at certain
location we return zero
                }
        }
        return 1; // If the number is not found anywhere we return 1
}

int checkgrid(int row, int column, int num)
{
        // This function checks whether we can put the number(num) in the 3*3 grid or
not
        // We get the starting row and column for the 3*3 grid row =
        (row / 3) * 3;
        column = (column / 3) * 3; int
        r, c;

        for (r = 0; r < 3; r++)
        {
                for (c = 0; c < 3; c++)
                {
                        if (sudoku[row + r][column + c] == num)
                        {
                                // If the number is found already present at certain location return
we return zero
                                0;
                        }
                }
        }




        // If the number is not found anywhere we return 1 return 1;
}

void navigate(int row, int column)
{
        // Function to move to the next cell in case we have filled one cell

        if (column < 8)
```

```
	{
		solvesudoku(row, column + 1);
	}
	else
	{
		solvesudoku(row + 1, 0);
	}
}


void display()
{
	// The function to display the solved Sudoku

	int row, column;
	printf("\n		THE SOLVED SUDOKU \n\n");
	printf("_____\n");
	for (row = 0; row < 9; row++)
	{
		if (row == 3 || row == 6)
		{
			printf(" |-_____|-_____|-_____-|\n");
		}
		for (column = 0; column < 9; column++)
		{
			if (column == 3 || column == 6 || column == 0)
			{
				printf(" | %d ", sudoku[row][column]);
			}
			else if (column == 8)
			{
				printf(" %d | ", sudoku[row][column]);
			}
			else if (row == 3 || row == 6)
			{
				printf(" %d ", sudoku[row][column]);
			}
			else
			{
				printf(" %d ", sudoku[row][column]);
```

```
                }
            }

            printf("\n");
        }
        printf("_____\n");
        getch();
}

void solvesudoku(int row, int column)
{
        if (row > 8)
        {
            // If the row number is greater than 8 than we have filled all cells hence we
have solved the sudoku
            display();
        }
        if (sudoku[row][column] != 0)
        {
            // If the value filled at a cell is not zero than it is filled with some value from 0
to 9 hence we move further
            navigate(row, column);
        }
        else
        {
            int ctr;
            /* This is a counter to check numbers from 1 to 9 whether the number
can be filled in the cell or not */

            for (ctr = 1; ctr <= 9; ctr++)
            {
                // We check row,column and the grid
                if ((checkrow(row, ctr) == 1) && (checkcolumn(column, ctr) ==
1) && (checkgrid(row, column, ctr) == 1))
                {
                    sudoku[row][column] = ctr;
                    navigate(row, column);
                }
            }
```

```
                sudoku[row][column] = 0; // No valid number was found so we clean up and
return to the caller.
        }
}

int main()
{
        int row, column;
        printf("Enter the desired sudoku and enter 0 for unknown entries\n"); for
        (row = 0; row < 9; row++)
        {
                for (column = 0; column < 9; column++)
                {
                        scanf("%d", &sudoku[row][column]);
                }
        }
        solvesudoku(0, 0); // We start solving the sudoku.
}
```

Conclusion

Finally we would like to conclude that this project of "Making a Sudoku Solver using C" can be best implemented by using backtracking approach as it is having the best time complexity and is more efficient compared to naive and brute force approach.Also we would like to tell that the sudoku solver has many real life uses as well, there are many people in the community who are serious hustlers of the game and for these percentage of people "The Sudoku Solver" can be a very useful and quick toolfor cross verification.

REFERENCES

https://afteracademy.com/blog/sudoku-solver

https://www.geeksforgeeks.org/backtracking-introduction/

https://en.wikipedia.org/wiki/Backtracking

https://www.youtube.com/watch?v=DKCbsiDBN6c

Let us C - By yashwant Kanetkar

SRM Course Coordinator PPT

Introduction To Algorithms – By Thomas H. Cornwell

CONCLUSION

REFERENCES