SUDOKU SOLVER USING BACKTRACKING

A COURSE MINI-PROJECT REPORT

By

JEPHRIN ESTHER(RA2111026010215)

ARYAMA AGRAWAL(RA2111026010242)

UNDER THE GUIDANCE OF

Dr. Shiva Shankar

In partial fulfillment for the Course of

18CSC204J - DESIGN AND ANALYSIS OF ALGORITHM

FACULTY OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chenpalpattu District

FACULTY OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

BONAFIDE CERTIFICATE

Certified that this Mini project Report " Psedo code solver using Backtracking programming" is the bonafide work of

V.JEPHRIN ESTHER(RA2111026010215)

ARYAMA AGRAWAL(RA2111026010242)

Who carried out the project work under my supervision.

SIGNATURE

Dr.Shiva Shankar

Assistant Professor

CINTEL

SRM Institute of Science **and** Technology

TABLE OF CONTENTS

1.CONTRIBUTION TABLE

Members:

V.JEPHRIN ESTHER

ARYAMA AGRAWAL

CONTRIBUTOR	CONTRIBUTIONS
JEPHRIN	**Came up** with **the concept** and **problem explanation**
ARYAMA	**Gave the algorithm for the problem**
JEPHRIN&ARYAMA	**Found complexity and** gave **an example for Problem**

1. PROBLEM DEFINITION

Pseudo-**code** is a **high**-level description of an algorithm or a computer program that uses informal language but **has** a structure **similar** to a programming language.

Problem
Definition**:**

You are given a list of integers, and you need to find the maximum **value** in **the list**.

Pseudo-code solution:

*****Set **max** value to **the** first **element in the list**.

*For each element in the **list** starting from the second element: **If** the current element **is** greater than max_value, set **max_value** to the current element.

This pseudo-code solution uses a loop to iterate through each element **in the** list and compare it to **the** current maximum value. If the current element is greater than the maximum value, it becomes the new maximum value. Finally, the maximum **value is** returned as **the** output of the **function**.

2.PROBLEM EXPLANATION

Pseudo-code **is** a **high-**level description of an algorithm or a computer program that uses informal language but has a structure similar to a

programming language. **It** is often used as a tool for planning and organizing code before writing actual code. **Example**:

Problem: Write a program to calculate the average of a **list** of **numbers**.

Solution using Pseudo-code:

1.**)**Initialize a variable called "sum" to zero.

2.) Initialize a variable called "count" to zero.

For each number in the list:

Add the number to **the** sum.

Increment the count **by** one.

3.Divide the sum by the count to get the
4.Return the average.

Explanation:

Average:

In this **example,** we **want** to write a program that calculates the average of a **list** of numbers. We can use pseudo-code to break down the problem into a series of **steps that** can be easily translated into actual code**.** Return the average.

3.) DESIGN TECHNIQUE USED

Backtracking: This is a technique used to explore all possible solutions by Incrementally building a candidate solution and undoing some choices when they do not lead to a valid solution. **This** technique can be used to solve problems like Sudoku puzzles or the n-Queens problem.

4.ALGORITHM FOR THE PROBLEM

```
def
solveSudoku(grid):
 row, col =
 findEmptyCell(grid)
 if
 row
          == -1 and col == -1:
    return True

 for num in range(1,
 10):
    if is Valid(grid, row, col,
    num):
grid[row][col] = num if
    solveSudoku(grid):
            return True
        grid[row][col]
        = 0

 return
 False

def is Valid(grid, row, col,
num):
 # Check
 row
```

```
for i in
range(9): if
grid[row][i]

    return
    False


# Check column

for j in
range(9):
  if
  grid[j][col
  ]
    return
    False
                          num
                          :




                          num
                          :


# Check 3x3 sub-
grid

sub_row = (row // 3) *
3

sub_col = (col // 3) *
3

for i in range(sub_row, sub_row+3):
  for j in range(sub_col, sub_col+3):
    if
    grid[i][j
    ]
      return False


return True
```

num
:

```
def
findEmptyCell(grid):
    for i in
    range(9):
        for j in
        range(9):
            if grid[i][j]=0:
              return (i, j)
return (-1, -1)
```

5.EXPLANATION OF ALGORITHM

The goal of the algorithm is to solve a given Sudoku puzzle, which **is a 9x9 grid** of cells that must be **filled** with numbers from 1 to **9,** such that each row, column, and **3x3 sub**-grid **contains** all the numbers from 1 to 9 exactly once.

The algorithm uses a backtracking approach, which means it tries out different possibilities for each empty **cell** in the grid until it finds a combination **that satisfies** all **the** Sudoku constraints.

Here's a step-**by-**step explanation of the algorithm:

The algorithm starts by defining **a** function "solveSudoku(grid)" **that** takes in a 2D array representing the Sudoku puzzle, with empty cells represented **as Os.**

The function looks for an empty cell in the grid by looping through each row and column and checking if the cell **value** is 0. If it finds **an** empty cell, it proceeds to step 4. If no empty cells are found, the function returns True to indicate that the puzzle has been solved.

If all the cells in the grid **are** filled with numbers, **the** puzzle **is** solved and **the** function **returns** True.

If the function finds an empty cell, it loops through the numbers 1-9 and checks if each number **is valid** in the current cell **by calling** a helper function "isValid(grid, row, col, num)". The helper function checks if the given number **is** already in **the** same row, column, or **3x3** sub-grid as the current **cell**.

If a number **is** valid, the function sets the current cell to that number and recursively calls "solveSudoku(grid)" to try to solve the rest of the puzzle.

If the recursive call returns True, **then** the puzzle has been solved and the **function** returns True **from the** current **call**.

Otherwise, **the** function resets the current **cell** to **0 and** tries the next number **in** the loop.

If no numbers are **valid in** the current cell, **the** function returns **False** to backtrack to the previous call and try a different number in the previous **cell**.

The algorithm continues to loop **through all the** empty cells **in** the grid until it **finds** a **combination of** numbers **that satisfies all the** Sudoku **constraints**.

Overall, the algorithm uses a recursive approach **that** systematically tries out different **possibilities** for each empty cell until it finds a **valid** solution **or** exhausts all possibilities. The "**isValid**" helper function is used to check if a given number **is** valid **in** a given row, column, and **sub**-grid of the Sudoku puzzle.

6.)COMPLEXITY ANALYSIS

The time complexity of the Sudoku solver using backtracking is dependent on the number of empty cells in the initial board and the number of possible candidates for each empty cell.

In the worst case scenario, the algorithm will have to backtrack all the way to the beginning of the puzzle, trying out every possible combination of numbers for each empty cell. This leads to a time complexity of O(9^(n*n)), where n is the number of rows (or columns) in the puzzle. In practice, however, the solver often finds a solution well before it has to try out all possible combinations.

Thus, the time complexity of the Sudoku solver using backtracking is exponential, but the actual running time will depend on the specific instance of the puzzle being solved.

7.CODE FOR THE PROBLEM

```
def find_empty_cell(board): #
Find an empty cell in the
board
for row in
range(9):
range(
9):
for col
in
if
board[row][col]
=
0
:
return (row,
col)
```

return
None

def is_valid_move(board, num,
row, col): # Check if a move is valid in a
given cell

Check the
row

for i in
range(9):

if board[row][i] =num and i !=
col:

return False # Check the

column for i in range(9):

i
f

board[i][col] - num and i !=
row:

return False # Check the 3x3 box

box row = row // 3 * 3 box col = col // 3 * 3

for i in range(box_row, box_row + 3):

range(box_col,
box_col+3):

num and (i, j) != (row,
col):

return
False

return
True

def solve_sudoku(board): # Find an

empty cell to fill empty_cell =
find_empty_cell(boa
rd)

empty_cell is
None:

el
se
:
return True

row, col =
empty_cell
i
f

for j
in
if
board[i][j]
 # Try
each
number
in the
empty cell
num in
range(1,
10):
i
f

is_valid_move(board, num, row,
col):

board[row][col] =
num

#Recursively try to fill the
rest of the board

if solve_sudoku(board):

return True

# If we can't find a solution, backtrack and **try** the next number
board[row][col] = **0**

If we've tried **all** numbers and none **work**, the board is unsolvable return False

8.CONCLUSION

In conclusion, solving problems using pseudo code is an essential skill in computer science and programming. There are various techniques that can be used to solve these problems, including brute force, greedy algorithms, dynamic programming, backtracking, and branch and bound. The choice of technique depends on the specific problem requirements and constraints. When solving problems **using** pseudo code**, it is** important to develop a clear **and** systematic approach, starting with a careful analysis of the problem statement and input/output requirements. From there**,** a plan can be formulated, pseudo code can **be** written**,** and the code can be tested and **refined**. By using the appropriate techniques and following a well-structured approach, **it is** possible to efficiently and effectively **solve** complex **problems** using pseudo code. This **can** lead to more efficient and effective programming**, as** well as a deeper understanding of algorithmic principles and computational **thinking**

REFERENCES:

https://www.geeksforgeeks.org/understanding-the-coin-change-problemwithdynamic-programming/