Sudoku Solver

A COURSE PROJECT REPORT

By
Yash Kumar(RA2111031010056) Yashi Jain(RA2111031010058)

Under the guidance of

DR. M .Shobana

In partial fulfilment for the Course

of

18CSC204J-DESIGN AND ANALYSIS OF ALGORITHMS

DEPARTMENT OF NETWORKING AND COMMUNICATIONS

FACULTY OF ENGINEERING AND TECHNOLOGY SRM

INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chengalpattu District - 603203

APRIL 2023

SRM

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR — 603 203
BONAFIDE CERTIFICATE

Certified that this project report "SUDUKO SOLVER using backtracking and branch and bounds" is the bonafide work of Yash Kumar(RA2111031010056) and Yashi Jain(RA2111031010058) who carried out the project work under my supervision.

SIGNATURE

Dr. M. Sobana
Assistant Professor
Networking and Communications
SRM Institute of Science and Technology
Kattankulathur

SIGNATURE

Dr. Annapurani Panaiyappan K
Head of the Department
Networking and Communications
SRM Institute of Science and Technology
Kattankulathur

School of Computing

SRM IST, Kattankulathur – 603 203 Course

Code: 18CSC204J

Course Name: DESIGN AND ANALYSIS OF ALGORITHMS

Problem Statement	Suduko solver with the help of backtracking and branch and bounds techniques
Name of the candidate	Yash Kumar(RA2111031010056)
Team Members	Yashi Jain(RA2111031010058)
Date of submission	29-04-2023

Mark Split Up

S. No	Description	Maximum Mark	Mark Obtained
1	Exercise	5	
2	Viva	5	
	Total	**10**	

Staff Signature with date

What is Back-Tracking approach??

Backtracking is a recursive algorithmic technique that is used to solve problems by trying out all possible solutions to find the correct one. It works by incrementally building a solution and backtracking as soon as it determines that the current solution cannot be completed or leads to a wrong result.

Examples- backtracking can be used to solve puzzles such as 8 Queens puzzle, crosswords etc….

The backtracking algorithm starts with an empty solution and incrementally builds it by selecting a candidate solution for each decision. The algorithm proceeds by choosing a candidate solution and testing it to see if it satisfies the constraints of the problem. If the candidate solution is valid, the algorithm moves on to the next decision, and if it is not valid, the algorithm backtracks and tries a different candidate solution.

What is Branch and Bounds approach??

The branch and bound approach is an algorithmic technique that is used to solve optimization problems. It is a more efficient version of the brute-force approach, which involves checking all possible solutions to find the optimal one. The branch and bound approach involves breaking down the problem into smaller subproblems, evaluating the subproblems and pruning subproblems that cannot lead to the optimal solution.

The branch and bound approach starts by formulating the problem as a tree. Each node of the tree represents a subproblem, and the edges represent possible decisions that can be made to reach a solution. The algorithm explores the tree by examining each node and then creating branches for each decision that can be made. As each subproblem is evaluated, the algorithm keeps track of the

current best solution and prunes any branches that cannot lead to a better solution than the current best.

How Back-tracking and Branch and Bounds approach helpful in solving Sudoku ??

Both backtracking and branch and bound algorithms can be helpful in solving a Sudoku puzzle:

Backtracking can be used to generate all possible solutions to the Sudoku puzzle by recursively trying out all possible values for each cell in the grid until a valid solution is found. If a candidate value leads to a contradiction, the algorithm backtracks and tries a different value for the cell. Backtracking can be implemented using recursion and can be optimized using techniques such as constraint propagation, which can help to eliminate values that are unlikely to lead to a valid solution, and variable ordering heuristics, which can help to choose the most promising candidate values first.

Branch and bound can also be used to solve Sudoku puzzles. In this approach, the problem is broken down into subproblems, and a lower bound is used to eliminate subproblems that cannot lead to the optimal solution. The lower bound is calculated by analyzing the current partial solution and identifying the minimum possible value for the remaining cells. The algorithm explores the tree of subproblems by selecting a candidate value for each cell and evaluating its lower bound. If the lower bound of a subproblem is greater than the current best solution, then that subproblem can be pruned. The algorithm continues until it finds the optimal solution or until all subproblems have been evaluated.

C++ code for suduko solver using Back-Tracking Approach:

```
#include<bits/stdc++.h>
using namespace std; class
Solution { public:
```

```
    vector<pair<int, int>> emptyCells;     int rows[9]
= {}, cols[9] = {}, boxes[9] = {};     void
solveSudoku(vector<vector<char>>& board) {

        // First, we loop through the entire Sudoku board and record the empty cells
        // in the `emptyCells` vector and record the used numbers in each row, column, and box
for (int r = 0; r < 9; r++) {             for (int c = 0; c < 9; c++) {                 if (board[r][c] == '.') {
                    emptyCells.emplace_back(r, c);
                } else {
                    int val = board[r][c] - '0';
int boxPos = (r / 3) * 3 + (c / 3);
rows[r] |= 1 << val;                     cols[c] |= 1 <<
val;                     boxes[boxPos] |= 1 << val;
                }
            }
        }

        // Once we have recorded the empty cells and used numbers in each row, column, and
box,
        // we call the `backtracking` function to solve the Sudoku puzzle
backtracking(board, 0);
    }

    // The `backtracking` function uses recursive backtracking to fill in the empty cells of     //
the Sudoku board until a solution is found or all possible solutions have been tried     bool
backtracking(vector<vector<char>>& board, int i) {
        // If we have filled in all of the empty cells, then we have found a solution!
if (i == emptyCells.size()) return true;

        // Otherwise, we get the row, column, and box position of the next empty cell to fill
int r = emptyCells[i].first, c = emptyCells[i].second, boxPos = (r / 3) * 3 + c / 3;
```

```
        // We then loop through all possible values (1-9) to fill the current empty cell          for
(int val = 1; val <= 9; ++val) {
            // We check if the current value is already used in the current row, column, or box
            // using bit manipulation to check the corresponding integer variable (rows, cols,
boxes)
            if (getBit(rows[r], val) || getBit(cols[c], val) || getBit(boxes[boxPos], val)) continue; //
skip if that value is existed!


            // If the current value is not already used, we fill in the empty cell with the value
board[r][c] = ('0' + val);


            // We then update the corresponding integer variables (rows, cols, boxes) to mark
            // the current value as used in the current row, column, and box
int oldRow = rows[r], oldCol = cols[c], oldBox = boxes[boxPos];
rows[r] |= 1 << val;              cols[c] |= 1 << val;              boxes[boxPos] |= 1
<< val;


            // We then recursively call the `backtracking` function to fill in the next empty cell
// If this recursive call returns true, then we have found a solution and can return true
if (backtracking(board, i + 1)) return true;


            // If the recursive call did not return true, we need to backtrack by restoring the
// original values of the corresponding integer variables (rows, cols, boxes)
rows[r] = oldRow; // backtrack              cols[c] = oldCol; // backtrack
boxes[boxPos] = oldBox; // backtrack
        }
return false;
    }
    int getBit(int x, int k) {          return (x >>
k) & 1;     }

};
int main(){
```

```
    vector<vector<char>> board = {
    {'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'}
};
 Solution solution;
solution.solveSudoku(board);     //
print the solved Sudoku board     for
(auto row : board) {         for (auto
cell : row) {             cout << cell << "
";
        }
        cout << endl;
    }
return 0;
}
```

C++ code for suduko solver using Branch and bounds Approach:

```
#include <iostream>
#include <vector> #include
<cmath> using namespace
std;
```

```
const int N = 9;

// Utility function to print the Sudoku grid void
printSudoku(vector<vector<int>> grid) {
   for (int i = 0; i < N; i++) {        for
(int j = 0; j < N; j++) {            cout
<< grid[i][j] << " ";
       }
       cout << endl;
   }
}

// Utility function to check if a number is valid in a cell bool isValid(vector<vector<int>>&
grid, int row, int col, int num) {
   // Check row and column constraints
   for (int i = 0; i < N; i++) {        if (grid[row][i] ==
num || grid[i][col] == num) {
          return false;
       }
   }
   // Check box constraint     int boxSize =
sqrt(N);     int boxRow = (row / boxSize) *
boxSize;     int boxCol = (col / boxSize) * boxSize;
for (int i = boxRow; i < boxRow +
boxSize; i++) {        for (int j = boxCol; j < boxCol
+ boxSize; j++) {            if (grid[i][j]
== num) {               return false;
          }
       }
```

```
    }
    return true;
}

// Function to find the next empty cell pair<int, int>
getNextEmptyCell(vector<vector<int>>& grid) {
    for (int i = 0; i < N; i++) {
for (int j = 0; j < N; j++) {            if
(grid[i][j] == 0) {                return
{i, j};
          }
       }
    }
    return {-1, -1}; // No empty cell found
}

// Branch and bound algorithm for solving Sudoku
bool solveSudoku(vector<vector<int>>& grid) {
// Find the next empty cell     pair<int, int> cell =
getNextEmptyCell(grid);
    if (cell.first == -1) {         return true;
// Puzzle solved
    }
    // Try each possible number in the cell  for
    (int num = 1; num <= N; num++) {
       if (isValid(grid, cell.first, cell.second, num)) {
// Make the assignment
grid[cell.first][cell.second] = num;            //
Recursively try to solve the puzzle            if
(solveSudoku(grid)) {                return true;
```

```
// Puzzle solved
        }
        // Backtrack          grid[cell.first][cell.second]
= 0;
      }
   }
   return false; // No valid number found for the cell
}
int main() {
cout<<"Enter the Sudoku To be Solved !"<<endl;
vector<vector<int>> grid; for(int i=0;i<9;i++){
vector<int> temp;    for(int j=0;j<9;j++){
      int t;       cin>>t;
temp.push_back(t);
   }
   grid.push_back(temp);
}
if (solveSudoku(grid)) {    cout << "Solution
          found:"          <<       endl;
printSudoku(grid);
} else {   cout << "No solution exists."
   << endl;
}
return 0;
}
```

Explanation for above code:

Firstly, the program defines a constant N as the size of the Sudoku grid, which is 9 in this case. Then, the program defines several utility functions such as printSudoku, isValid, and getNextEmptyCell.

printSudoku function is used to print the Sudoku grid on the console. isValid function is used to check if a number is valid in a cell. It takes the Sudoku grid, row, column, and number as arguments and checks if the number violates the row, column, or box constraints. getNextEmptyCell function is used to find the next empty cell in the grid. It returns the row and column indices of the empty cell.

The solveSudoku function takes the Sudoku grid as an argument and recursively tries to solve the puzzle. It first finds the next empty cell using getNextEmptyCell. If there is no empty cell left, it means that the puzzle is solved, and it returns true. Otherwise, it tries each possible number in the cell using isValid function. If a number is valid, it makes the assignment and recursively tries to solve the puzzle. If the puzzle is solved, it returns true. If the puzzle is not solved, it backtracks and tries the next possible number.

In the main function, the program prompts the user to enter the Sudoku grid to be solved. It reads the input grid using cin and stores it in a vector of vectors called grid. Then, it calls the solveSudoku function to solve the puzzle. If the puzzle is solved, it prints the solution using the printSudoku function. Otherwise, it prints a message saying that no solution exists.

Output

```
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
```