# Solving Sudoku with Constraint Satisfaction Problem

Ali Haider Zaveri - 108661497
Jagat Sastry - 108721027

## *Sudoku explained*

Sudoku is a number placement puzzle. In this puzzle you are given an N x N grid of cells. The grid itself is composed of M x K sub-grids. You can place a single digit, drawn from 1 to N, in any cell. Initially the grid will have some of its cells partially filled. The objective of the puzzle is to complete the grid so that:

1. Every cell contains a digit.
2. No digit appears twice in any row, column of the N x N grid or in any row, column of any of the M x K sub-grid.

## *Sudoku as a Constraint Satisfaction Problem*

Each cell on the N x N grid can be thought of as a variable. Constraint on each variable is that no two variables in the same row, column or M x k sub-grid can have the same number([1-N]) assigned to it. In this problem each constraint is a binary constraint as only two variables are involved in each constraint.

We have solved Sudoku by modeling it as a Constraint Satisfaction Problem by using the four methods which are explained in detail below.

### *Pseudo Code:*

```
SudokuBT(board){
      inputs: board, the Sudoku board-each cell of the board is a variable
      local variable: val-domain values of variables
                 cell-board cell
     If AC-3 fails, return false //For CSP arc consistency
      select cell //in order of minimum remaining values except for basic backtracking
      for each val in Domain[cell] do
            if(valid assignment)
                  assign val to cell
                  If fwd checking fails, continue //Only for fwd checking case
                  if(call SudokuBT(board))
                        return true
                  else
                        reset cell to 0
      end for

      return false
}
```

## *Backtracking*

**`SudokuGridDFS.java`**

Backtracking is a recursive algorithm. Using it without any heuristic is a naïve approach for solving any constraint satisfaction problem.

From the above pseudo code it can be seen that the backtracking approach makes recursive calls until it finds the board inconsistent. When it finds the board inconsistent it backtracks to find another assignment of variables for which the boards will be consistent. Pseudo code for this has been given above.

## *Backtracking + MRV heuristic*

In the naïve backtracking approach there was no logic behind choosing next variable for assignment.
The MRV (Minimum Remaining Values or Most Constrained Variable) heuristic is used with backtracking to help choose the next variable for assignment. The most restricted variable is the one which has the least number of domain values remaining to which it can be assigned to.
The pseudo code is similar to normal backtracking with the only difference being in selecting the next variable for assignment.

### ***Implementation specifics and optimizations:***

**`SudokuGridMRV.java`**

We have created a class called Cell.java the objects of which signify the variables of the problem
Cell.java has the following attributes

- m_val: stores the value of the variable
- m_constraints: A list of values that the variable 'cannot' take.
- m_row,m_col: Position of cell on the grid

Instead of storing all the values that a cell can take, we store only those that it cannot take, thus minimizing the memory required and constraint checking time.

## *Backtracking + MRV + Forward Checking*

We use the pseudo code we had used for backtracking but with the following modifications.

As seen in the results section, backtracking used with the MRV heuristic does help in considerable reduction in the number of expanded states (cells). Forward checking helps to determine inconsistent states earlier. After assigning a value to the most restricted variable, if any of its neighbours (that have not been assigned a value as yet), i.e. the variables with which it is involved in a constraint cannot be assigned a value, we backtrack. This helps in determining inconsistencies earlier as we do not have to wait for the assignment to actually fail.

### *Implementation specifics:*

**`SudokuGridFwdChecking.java`**

After assigning a value to the most restricted cell we check the constraint list of each of its neighbours that have not been assigned a value as yet. If the constraint list of any of these neighbours is equal to N i.e. if the neighbouring cell cannot take any more values then we backtrack.

## *Backtracking + MRV + Constraint Propagation*

We use the pseudo code we had used for backtracking but with the following modifications.

We have used constrain propagation as a predecessor to every assignment. In constraint propagation we are seeing whether the arc between two neighbours is consistent or not. The arc between two neighbours is consistent if for every value in the domain of variable X there is a value in the domain of variable Y that it can take without compromising the constraint test.

### *Implementation Specifics:*

**`SudokuGridMRVCP.java`**

We use the AC-3 algorithm to check arc consistency. This helps to detect failures early even before doing an assignment.

Contd in the next page...

## ***Results:***

For Test 1

```
6 2 3
_,_,6,_,5,_
_,1,_,2,_,_
_,_,1,_,_,_
_,_,_,3,_,_
_,_,4,_,1,_
_,2,_,4,_,_
```

| Method | Consistency Checks | Time(ms) | Memory(MB) |
|---|---|---|---|
| Backtracking | 474 | ~0 | 1.23 |
| Backtracking + MRV | 31 | ~0 | 0.62 |
| Backtracking + MRV + Fwd check | 30 | ~0 | 0.62 |
| Backtracking + MRV + CP | 27 | ~0 | 1.25 |

For test case 2

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

| Method | Consistency Checks | Time(ms) | Memory(MB) |
|---|---|---|---|
| Backtracking | 61386 | ~0 | 3.7 |
| Backtracking + MRV | 50 | ~0 | 0.62 |
| Backtracking + MRV + Fwd Check | 51 | ~0 | 0.62 |
| Backtracking + MRV + CP | 51 | ~0 | 3.7 |

For test case 3

```
12 3 4
_,_,_,_,8,3,_,_,6,_,7,_
_,4,_,7,_,_,_,_,5,2,_,12
9,_,_,11,_,5,7,_,_,_,10,4
11,7,_,2,_,10,_,_,_,_,_,5
_,_,12,_,_,_,_,_,_,7,_,_
_,_,_,_,12,_,_,5,_,11,_,_
_,_,3,_,9,_,_,1,_,_,_,_
_,_,4,_,_,_,_,_,_,12,_,_
6,_,_,_,_,_,5,_,3,_,8,2
12,2,_,_,_,9,6,_,4,_,_,3
1,_,8,4,_,_,_,_,2,_,12,_
_,9,_,3,_,_,12,2,_,_,_,_
```

| Method | Consistency Checks | Time(ms) | Memory(MB) |
|---|---|---|---|
| Backtracking | 163104392 | ~54596 | 0.62 |
| Backtracking + MRV | 11576 | ~0 | 4,32 |
| Backtracking + MRV + Fwd Check | 12990 | ~0 | 6.16 |
| Backtracking + MRV + CP | 6278 | 5041 | 21.5 |

**Running Tests**

Run the code in the following way

Change the method as required in *TestSudoku.METHOD* java code and call the java code in the following manner

```
java ai.TestSudoku test_case_file
```

where test_case_file has entry like

```
6 2 3
_,_,6,_,5,_
_,1,_,2,_,_
_,_,1,_,_,_
_,_,_,3,_,_
_,_,4,_,1,_
_,2,_,4,_,_

0 0 0
```