## Solving Sudoku with Constraint Satisfaction Problem

Ali Haider Zaveri - 108661497

Jagat Sastry -

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 using Constraint Satisfaction Problem by using the four methods which are explained in detail below.

Backtracking

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

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

select cell

for each val in Domain[cell] do

if(valid assignment)

assign val to cell

if(call SudokuBT(board))

return true

else

reset cell to 0

end for

return false

}

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.

Backtracking + MRV heuristic

In the naïve backtracking approach there was no logic behind choosing next variable for assignment.

The MRV (**M**ost **R**estricted **V**ariable) 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:

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

Maintaining a list of values that the cell cannot take helps reduce storage space.

We maintain a TreeSet of all variables (cell) which is ordered by the size of the list that maintain values in the domain the cell cannot be assigned to. The TreeSet implementation provides guaranteed log(n) time cost for the basic operations (add, remove and contains).

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 consistency checks performed. 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 then we backtrack. This helps in determining inconsistent earlier as we do not have to wait for the assignment to actually fail.

Implementation specifics:

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:

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

Results:

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Consistency Checks | Time(ms) | Memory |
| Backtracking |  |  |  |
| Backtracking + MRV |  |  |  |
| Backtracking + MRV + Forward checking |  |  |  |
| Backtracking + MRV + Constraint Propagation |  |  |  |