SUBJECT NAME: ARTIFICIAL INTELLIGENCE:

KNOWLEDGE REPRESENTATION AND PLANNING

ASSIGNMENT - 1
SUDOKU PUZZLE SOLVER

SUBMITTED BY:

PATIL KHUSHBU MAHENDRA

Matriculation No: - 882697

KHALID VAFA

Matriculation No: - 882677

SUBMITTED TO:

PROF. ANDREA TORSELLO

Abstract

We are given a starting board, and we have to find a valid solution to the Sudoku problem. The board can have random values with empty blanks space to be filled with correct numbers.

Attempting to solve this problem using the backtracking algorithms. We solve problem using AC-3, Forward checking and Backtracking search algorithms.

Contents

1 Introduction

A Sudoku puzzle (to fill a 9*9 grid with digits so that each column, each row, and each 3*3 box contains all of the digits from 1 to 9) can be considered as a CSP: there are 81 variables, one for each node, among which those empty nodes have the domain {1,2,3,4,5,6,7,8,9}, while those prefilled nodes contain only one value in their domains. Moreover, the Sudoku puzzle is a 9-consistency CSP in that there are 27 (9 rows, 9 columns and 9 boxes) $Alldiff$ constraints and each involves 9 nodes.

In this assignment, we solve the Sudoku puzzle by using AC-3, forward checking and backtracking search algorithms.

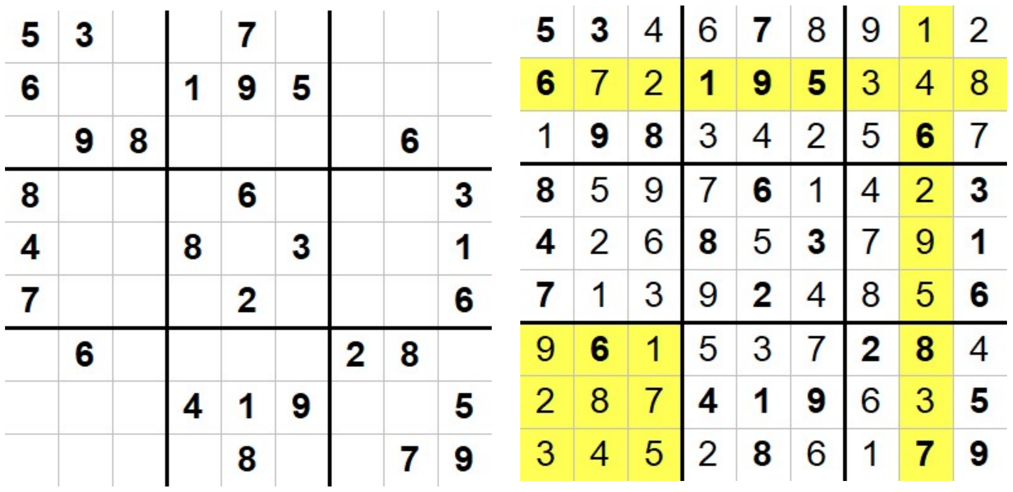
2. Goal

The goal is to understand constraint propagation and backtracking algorithm by solving Sudoku puzzle. Solving Sudoku can be done in many ways as it is a game for some the analysis of Sudoku falls into two main areas:

1. Analysing the properties of completed grids.
2. Analysing the properties of puzzles

In our case we have a classic Sudoku, i.e. N = 9 (a 9x9 grid and 3x3 regions) . A rectangular Sudoku that uses rectangular regions of row column dimension Row by Col. Other variants include those with irregularly-shaped regions or with additional constraints *hypercube* or different constraint types. Typically some of the cells in a Sudoku grid will have been pre-filled by the puzzle master and our goal is to fill the board with digits from 1 to 9 such that

1.each number appears only once for each row column and 3 X 3box. 2. each row, column, and 3 x 3 box should containing all 9 digits as this in shown in[1] Figure 1.



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**

Figure 1: Sudoku Puzzle Board Unsolved

3. Constraint Propagation

AC-3 is used to make the whole CSP arc-consistent on the basis that the CSP only contains binary constraints or unary constraints. To solve this puzzle in AC3, we need to transform all the higher-order constraints $ Alldiff$ to binary constraints.

The $ Alldiff$ constraint tells that each of 9 variables can choose one integer from 1 to 9 plus it is different from others, which is equal to the statement that each of 9 variables can choose one integer from 1 to 9 while every 2 variables in 9 are different.

$$ Alldiff(A,B,C,D,E,F,G,H,I)$$
$$=Diff(A,B)+Diff(A,C)+Diff(A,D)+Diff(A,E)+Diff(A,F)+Diff(A,G)+Diff(A,H)+Diff(A,I)$$
$$+Diff(B,C)+Diff(B,D)+Diff(B,E)+Diff(B,F)+Diff(B,G)+Diff(B,H)+Diff(B,I)$$
$$+Diff(C,D)+Diff(C,E)+Diff(C,F)+Diff(C,G)+Diff(C,H)+Diff(C,I)$$
$$+Diff(D,E)+Diff(D,F)+Diff(D,G)+Diff(D,H)+Diff(D,I)$$ $$+Diff(E,F)+Diff(E,G)+Diff(E,H)+Diff(E,I)$$
$$+Diff(F,G)+Diff(F,H)+Diff(F,I)$$ $$+Diff(G,H)+Diff(G,I)$$ $$+Diff(H,I)$$

The formula shows that one $Alldiff$ constraint can be transformed to 36 Diff(A,B) constraints, thus following codes add 1944 (27*36*2) arcs into Array List<Arc> arcs(actually acts like a set) before AC-3 starts. Notice that Arch(node1,node2) is different with `Arch(node2,node1)`.

`// parts of the code in AC3.java`

```java
for (int i = 0; i < 9; i++) {
    addRowArcs(i);
    addColArcs(i);
    addBoxArcs(i);
}

void addRowArcs(int row) {
    for (int i = 0; i < 8; i++) {
        for (int j = i + 1; j < 9; j++) {
            arcs.add(new Arc(nodes[row][i], nodes[row][j]));
            arcs.add(new Arc(nodes[row][j], nodes[row][i]));
        }
    }
}

void addColArcs(int col) {
    for (int i = 0; i < 8; i++) {
        for (int j = i + 1; j < 9; j++) {
            arcs.add(new Arc(nodes[i][col], nodes[j][col]));
            arcs.add(new Arc(nodes[j][col], nodes[i][col]));
        }
    }
}

void addBoxArcs(int box) {
    // add nodes which are in the same box together
    ArrayList<Node> nodesBox = new ArrayList<>();
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            if (nodes[i][j].box == box) {
                nodesBox.add(nodes[i][j]);
            }
        }
    }
    for (int i = 0; i < 8; i++) {
        for (int j = i + 1; j < 9; j++) {
            arcs.add(new Arc(nodesBox.get(i), nodesBox.get(j)));
            arcs.add(new Arc(nodesBox.get(j), nodesBox.get(i)));
        }
    }
}
```

AC-3

AC-3 is a way of reduce domains of variables by making them arc-consistent before searching.

In the previous section, we have put all arcs into the set arcs. In AC-3, every time we remove one arc from arcs and then check node1's arc-consistency with respect to node2. Values will be removed from node1.domain if they are also in node2.domain. If node1.domain contains no value after this process, we can assume that this puzzle has no solution. Otherwise we add all Arch(node3,node1) into set arcs where node3 has constraint with node1 and node3 is not node2(implemented by function addNeighborArcsExceptB). We continue the whole step until the set is empty (AC-3 is done) or one node's domain is empty (there is no solution).

```
// parts of the code in AC3.java
```

```
while (!arcs.isEmpty()) {
    Arc arch = arcs.remove(0);
    if (!establishAC(arch)) {
        // no solution
    }
}

boolean establishAC(Arc arc) {
    Node node1 = arc.node1;
    Node node2 = arc.node2;
    for (Integer i : node1.domain) {
        boolean isDiff = false;
        // if node2 contains one value same with that in node1, then isDiff==true
        for (Integer j : node2.domain) {
            if (i != j) {
                isDiff = true;
                break;
            }
        }

        if (!isDiff) {
            node1.domain.remove(i);
            if (node1.domain.isEmpty()) {
                return false;
            }
            addNeighborArcsExceptB(node1, node2);
        }
    }
    return true;
}
```

In some situations, we can find a solution when AC-3 is finished. Here is one example: after 320 steps of shrinking domains we find a solution.

```
// output when running
// new AC3(Node.getNodes("data0"));
// Test.VERBOSE = true;

loading puzzle from data0...

[0][0][0][0][0][0][6][0][0]
[8][0][4][0][0][0][0][0][7]
[0][0][5][0][0][3][0][9][0]
[0][0][1][0][9][8][0][5][2]
```

```
[6][9][2][0][0][7][3][0][0]
[4][5][0][1][0][0][0][0][6]
[5][0][0][3][6][0][2][4][9]
[2][4][0][8][7][0][1][0][3]
[3][0][6][2][4][9][7][8][5]

AC3...
step 1  set.size():1933  AC:(0,0)(0,6)  (0,0).remove(6)  domain:[12345789]
step 2  set.size():1938  AC:(0,1)(0,6)  (0,1).remove(6)  domain:[12345789]
...
step 319  set.size():770  AC:(0,1)(1,1)  (0,1).remove(2)  domain:[7]
step 320  set.size():392  AC:(0,7)(0,2)  (0,7).remove(3)  domain:[2]

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

total running time:21ms
```

4. Direct Checking Method

Like AC-3, forward checking is also a way of inferencing reductions in variables' domains before searching.
Whenever a variable X is assigned, the forward checking process establishes arc consistency for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y.domain any value that is inconsistent with the value chosen for X (Y is one neighbor of X). If Y.domain is empty, one inference can be made that there is no solution.
Codes below show how forward checking works; however with a little difference: Instead of checking Ys' arc consistencies with newly assigned X, we replace X with all variables containing only one value. Yet it may take more steps for executing, there is no need to check whether a variable with one value is given by the puzzle or assigned by our assumption.

```
// parts of the code in Backtracking.java
```

```java
boolean forwardCheck() {
        boolean isContinue = false;
        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                if (nodes[i][j].domain.size() == 1) {
                    ArrayList<Node> neighbors = getNeighbors(nodes[i][j]);
                    for (Node n : neighbors) {
                        int value = nodes[i][j].domain.first();
                        if (!removeNodeValue(n, value)) {
                            return false;
                        } else if (n.domain.size() == 1) {
                            isContinue = true;
                        }
                    }
                }
            }
        }
        if (isContinue) {
            return forwardCheck();
        }
        return true;
    }
```

When forward checking is applied after one puzzle is given, the result is the same with that of AC-3 because they both make all variables arc-consistent. However, since the former one has recursion and pruning, we choose to use it during backtracking search step.

```
// output when running
// new AC3(Node.getNodes("data6"));
// new Backtracking(Node.getNodes("data6"));
// Test.VERBOSE = false;

loading puzzle from data6...

[0][7][0][8][0][0][0][0][0]
[2][0][0][3][0][6][0][7][0]
[0][9][8][0][2][0][1][3][0]
[0][5][0][0][0][0][9][8][7]
[8][0][0][0][7][0][0][4][1]
[9][0][7][0][0][0][0][5][2]
[0][0][9][0][6][0][5][0][3]
[7][0][0][1][0][4][0][6][0]
[0][0][0][0][0][3][7][0][4]
```

```
AC3...

[13456][7][13456][8][1459][159][246][29][569]
[2][14][145][3][1459][6][48][7][589]
[456][9][8][457][2][57][1][3][56]
[1346][5][12346][246][134][12][9][8][7]
[8][236][236][2569][7][259][36][4][1]
[9][1346][7][46][1348][18][36][5][2]
[14][1248][9][27][6][278][5][12][3]
[7][238][235][1][589][4][28][6][89]
[156][1268][1256][259][589][3][7][129][4]

first time of forward checking...

[13456][7][13456][8][1459][159][246][29][569]
[2][14][145][3][1459][6][48][7][589]
[456][9][8][457][2][57][1][3][56]
[1346][5][12346][246][134][12][9][8][7]
[8][236][236][2569][7][259][36][4][1]
[9][1346][7][46][1348][18][36][5][2]
[14][1248][9][27][6][278][5][12][3]
[7][238][235][1][589][4][28][6][89]
[156][1268][1256][259][589][3][7][129][4]
```

5. Backtracking

Backtracking search uses a depth-first search to choose values for one variable at a time and backtracks when a variable has no legal values left to assign.

Here we can apply heuristics when selecting variables and values. In codes, the get HeuristicNode() function uses the minimum-remaining-values (MRV) heuristic, getting one unassigned node (node.domain>1) with the minimum domain size. There is no need to apply degree heuristic for choosing the first node to expand because in the sudoku puzzle degree heuristic and MRV heuristic are the same --- one node with the most constraints with other unassigned nodes is always the one having the minimum remaining values.

The getHeuristicValue() function uses the least-constraining-value (LCV) heuristic, selecting one value for current variable that can provide the maximum choices for all neighboring variables.

Moreover, every time when we remove one value from one variable's domain, we put this variable and the removed value as a record into a stack. Then we use the size of the stack to represent current state. Next time when we want to backtrack to former state, we use backtrackState(state) function to remove those records exceeding the recorded size of the stack and then put values back into their corresponding variables' domains.

Backtracking search starts from one variable, assigning one value and do forward check each time. When forward checking find no error, use recursion to search on another node. Otherwise remove this value from current variable's domain and do forward check on another value.

```
// parts of the code in Backtracking.java
```

```java
// use recursion to search value for all the nodes
//
// error, return -1
// not end, return 1
// find a solution, return 0
int search(Node node) {
    int state = stack.size();

    int result = Node.judgeState(nodes);
    // return 0, find a solution; return -1, error
    if (result != 1) {
        return result;
    }
    // not end while cannot get one more node, return error
    if (node == null) {
        return -1;
    }

    // not end, iterate each value of the node
    int value = getHeuristicValue(node);
    while (value != 0) {
        setNodeValue(node, value);
        if (!forwardCheck()) {
            // if this value is invalid
            backtrackState(state);
            if (!removeNodeValue(node, value)) {
                return -1;
            }
            state = stack.size();
            value = getHeuristicValue(node);
        } else {
            // if this value is valid
            // use recursion to search the next node
            result = search(getHeuristicNode());
            if (result == 0) {
                return 0;
            } else if (result == -1) {
                backtrackState(state);
                if (!removeNodeValue(node, value)) {
                    return -1;
                }
                state = stack.size();
            }
            value = getHeuristicValue(node);
        }
    }
    // tried all value while not finding a solution, then is a error
    return -1;
}
```

6. Conclusion

We can see that the result after AC-3 and that after first time of forward checking are the same. Two executions are the same during the later backtracking search step while the total running times are different --- this is because forward checking uses recursion and pruning thus performs better than AC-3.

You can also set static boolean VERBOSE=true in Test.java to show detailed records of steps in AC-3 and backtracking search, getting output like this:

```
...
AC3...
step 1  set.size():1941  AC:(0,0)(0,2)  (0,0).remove(5)  domain:[12346789]
step 2  set.size():1958  AC:(0,0)(0,3)  (0,0).remove(3)  domain:[1246789]
step 3  set.size():1965  AC:(0,1)(0,2)  (0,1).remove(5)  domain:[12346789]
...
step 302  set.size():5766  AC:(6,7)(8,6)  (6,7).remove(7)  domain:[14]
step 303  set.size():5778  AC:(7,6)(6,8)  (7,6).remove(9)  domain:[1268]
step 304  set.size():5769  AC:(8,8)(7,7)  (8,8).remove(3)  domain:[12458]
step 305  set.size():5787  AC:(7,8)(8,6)  (7,8).remove(7)  domain:[1258]

backtracking...
step 1  try 5 for (5,1)
step 2  try 2 for (4,0)
step 3  try 8 for (3,1)
...
step 81  try 6 for (0,4)
step 82  try 1 for (0,0)
```