ISP Project 3
Spring 2018

Thor Olesen (tvao@itu.dk)
Dennis Nguyen (dttn@itu.dk)
Daniel Hansen (daro@itu.dk)

April 28, 2018

Contents

1 Sudoku Constraint Satisfaction Problem

The sudoku puzzle represents a constraint satisfaction problem (i.e. CSP) defined by a state and a goal test. Namely, it is defined by a set of variables, Xi, with values from a domain D. By default, the sudoku board is 9x9, meaning its state is defined by 81 variables that may be assigned to values from a domain of possible values ranging between 1 to 9. In this regard, the goal test is a set of constraints specifying the legal combinations of values for variables. Specifically, the places placed on the board has to be between 1 to 9 (i.e. the domain), and the values should not be duplicated across any rows, columns or squares of the sudoku puzzle.

Finally, it should be noted that the CSP is an NP-complete problem, since it is a decision problem that currently admits no polynomial-time solution. By example, consider the problem depicted in the constraint satisfaction graph below with nodes representing variables and edges representing constraints:

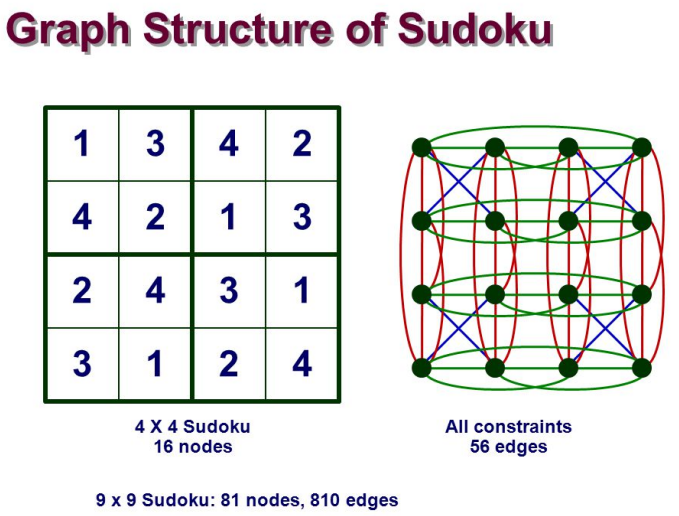

Figure 1: Sudoku Constraint Graph

The Sudoku problem is reducible to the NP-hard graph coloring problem where one has to assign "colors" to vertices of the graph such that no two adjacent vertices share the same color. Namely, the Sudoku puzzle can be thought of as a graph with 81 vertices, one for each cell, and two vertices are connected by an edge if they cannot be assigned the same value from the domain, D1,2,3,4,5,6,7,8,9. Thus, given a Sudoku puzzle one may build a graph like the above and use graph coloring to find a 9-coloring of the graph (colors 1 to 9), indicating the problem is not polynomially solvable.

2 Runtime Complexity

As shown previously, solving the Sudoku puzzle is computationally hard and using a traditional backtracking algorithm will have the disadvantage that the algorithm is slow. Specifically, he backtracking algorithm is a brute-force algorithm that uses depth-first search to solve the Sudoku CSP. However, this means it potentially goes through all N possible variable assignments to all domain values in D, ultimately resulting in a total of d^n possible assignments or 9^{81} assignments. In other words, given a Sudoku puzzle with 81 variables that each have 9 possible states from the domain, the exponential runtime complexity can be roughly estimated to be bound by $O(d^n)$.

3 Forward Checking and Arc-Consistency

Instead of using the backtracking algorithm, one may use the forward checking algorithm to reduce the game state exploration. Forward checking addresses the problem that it may sometimes not be possible to assign legal domain values to given variables in the Sudoku puzzle. Thus, forward checking is used to evaluate the effect of a specific variable assignment. It does so by making a tentative variable assignment and then checking the remaining unassigned variables recursively to see if there exists a consistent solution.

As improvement, one may use the **arc-consistency** look-ahead technique to ensure that all the remaining unassigned variables belong so a solution. In other words, one may enforce that the remaining variables are arc-consistent with another variable, meaning all of its admissible (i.e. legal) values are consistent with some admissible value of another variable. As a result, the AC (arc-consistency) algorithm will propagate the constraints and reduce the domain size of variables by ensuring all possible (future) assignments are consistent.

Finally, **forward checking** is applied to further reduce the domain of variables by identifying possible variables that do not admit any legal value assignments from the domain. For further improvements, one might optimize the algorithm using heuristics such as **Minimum Remaining Values** (MRV assigns variables with fewest legal conflicts first), textbfDegree heuristic (picks variable with most constraints to reduce domain of remaining variables) or **Least constraining value** (LCV picks variables with the most slack or flexibility for the future).

NB: for some reason, our SudokuSolver implementation is correctly able to detect unsolvable Sudoku puzzle configurations but fails to solve valid Sudoku puzzles due to a memory stack overflow caused in the provided AC method (i.e. arcConsistencyFC that uses revise).

4 Implementation

We have implemented the forward checking algorithm to solve the constraint satisfaction problem defined by the Sudoku puzzle. The implementation uses the provided arc-consistency algorithm to check that all future variable assignments are consistent. In other words, it checks the 4 constraints of Sudoku, meaning all variables are assigned to a value between 1 to 9 in the domain and no variables are duplicated across columns, rows or squares in the puzzle. In order to implement **SudokuSolver.java**, one should initialize an empty puzzle in the **setup(size)** method and implement the **solve()** method to check if the puzzle can be solved. For the sake of brevity, only the FC algorithm is provided:

Source Code 1: Forward Checking Algorithm (FC)

```
public ArrayList<Integer> forwardChecking(ArrayList<Integer> assignment) {
        // Check if board contains solution (all values are 1 to 9)
        if (!assignment.contains(0)) return assignment;

        // Find first unassigned variable (X)
        int tentativeAssignmentX = assignment.indexOf(0);

        // Maintain separate copy of domains for roll-back/backtracking
        ArrayList<ArrayList<Integer>> oldDomains = deepCopy(domains); // D_old
        ArrayList<Integer> temporaryAssignmentDomainDX =
                new ArrayList<>(domains.get(tentativeAssignmentX)); // DX

        // Assign each value in domain to variable and for consistency
        for(int valueV : temporaryAssignmentDomainDX) {

                // Check if variable assignment is consistent (no violations)
                if(arcConsistencyFC(tentativeAssignmentX, valueV)) {

                // Assign domain value V to variable X
                assignment.set(tentativeAssignmentX, valueV);

                // Do forward checking on remaining variables to reduce domain
                ArrayList<Integer> forwardCheckingResult = forwardChecking(assignment);

                // Check for valid solution in look ahead (backtracking)
                if (forwardCheckingResult != null) return forwardCheckingResult;

                // No valid variable assignment so undo (0)
                assignment.set(tentativeAssignmentX, 0);
                domains = deepCopy(oldDomains); // Rollback
                } // No consistent variable assignment
                else domains = deepCopy(oldDomains);
        } return null; } // Failure
```