ISP Mandatory Assignment 3 - Sudoku Solver Based on Forward Checking

Overview

This week the task is to implement a forward checking algorithm to solve a *Sudoku* puzzle. You should form groups of two to three students. You will be provided with Java code support but implementations in other languages are also acceptable.

Background Information

The Forward Checking (FC) algorithm is described in RN10. It is a backtracking algorithm that after assigning a value V to a variable X, enforces consistency between all unassigned variables Y and the variable X. If any domain D_Y becomes empty, assignment $X = V$ is inconsistent, and another value should be tried for X. More details on Forward Checking will be provided in the follow-up sections.

The Sudoku puzzle is defined by 9×9 board. To successfully solve the puzzle, one should fill all the empty squares so that the numbers 1 to 9 appear once in each row, column and 3×3 box. An example of solved Sudoku is shown in Figure 1.

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

Figure 1: A solved Sudoku puzzle. All the empty squares are filled so that the numbers 1 to 9 appear once in each row, column and 3x3 box.

A GUI displaying a puzzle and accepting a user input is provided (see Figure 2). Your task is to implement an FC algorithm, that accepts a partial

assignment representing initially filled numbers in the Sudoku puzzle. The algorithm either returns an assignment that represents a successful filling of all the empty squares, or it returns a failure, indicating that the Sudoku cannot be solved. The puzzle-board is then updated based on this result.

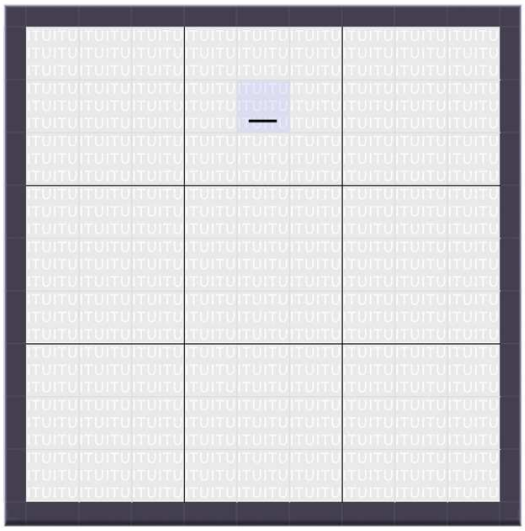

Figure 2: The empty Sudoku puzzle GUI. Position (4,1) is highlighted for insertion of a value.

File Overview

A GUI-class (`SudokuGUI`) has been implemented in Java for you to use, so you can spend more time on implementing the forward checking (FC) algorithm. The GUI-class calls functions defined in the interface `ISudokuSolver` and your task is to implement this interface. You do not have to change anything in the GUI class if you just implement the interface according to the comments above the methods in the interface. If you program in C# you have to make a GUI yourself. You are provided with 4 files, `SudokuGUI`, `ISudokuSolver`, `SudokuSolver` and `ShowSudoku`. The `SudokuSolver` is a very small implementation of the interface, so you are able to see an empty puzzle board and to insert numbers on the board. This file should be extended with an implementation of FC algorithm. The last file contains the main-method to view the puzzle.

GUI and Interface

When running the main method of the class `ShowSudoku`, a frame opens that displays an initially empty sudoku puzzle. When a particular square in the puzzle is clicked, it is highlighted, and a value between 1 and 9 can be inserted by typing the number on the keyboard (see Figure 2). An assigned value can be changed by inserting another number or by deleting the value. To delete a value, you should press space after clicking on the particular place. To see the solution of the sudoku (if one exists), you should press the solve-button. It will be displayed, if the puzzle cannot be solved. Of course the above only works

after you implemented the `ISudokuSolver` interface described below. The interface has the following 5 methods: `getPuzzle()`, `setValue(col, row, value)`, `setup(size)`, `readInPuzzle(puzzle-array)` and `solve()`; the last method is the one that involves calling a foward checking (FC) algorithm.

`getPuzzle()` returns the content of the sudoku puzzle as a 2 dimensional array of integers. If there are no inserted values in the puzzle the array should contain zeros in all entries. If a value has been inserted at a specific place, then the corresponding entry in the array should have that particular value. The puzzle has entry (0,0) in the upper left corner (see Figure 2).

`setValue(col, row, value)` inserts the specified value in the specified position in the puzzle given by a column number and a row number. Again, the numbering starts with the 0th column and the 0th row in the upper left corner of the puzzle. Inserted values should be in the legal range (1-9 for a normal sudoku), and these should only be inserted in places that correspond to places in the sudoku; the method should therefore check this before insertion. In order to be able to erase already inserted values, it should also be possible to insert the value 0.

The `setup()` function initializes a sudoku puzzle of the specified size, such that all entries are empty (the array contains 0s). The size is given in number of blocks in a row, ie. the size is 3 for a normal sized sudoku. In this function you can initialize auxiliary datastructures for FC algorithm. The `readInPuzzle(puzzle-array)` reads in an 2D-array and stores it as the current content of the puzzle. This is done to make a more efficient way of reading in a predefined puzzle, if you want to avoid initializing a puzzle by clicking and using `setValue` function. Before reading in the puzzle, the method should check if the specified array has the right dimensions and legal values according to the size of the puzzle.

The most interesting (and hard) method to implement is the function `solve()`. This method should check if the puzzle with the current content can be solved or not and return true, if it is solvable and false otherwise. If the puzzle is solvable, the puzzle should furthermore be solved and the result saved, such that when `getPuzzle()` is called subsequently, it will return the content of the full solution. To solve the puzzle, the FC algorithm should be used.

Forward Checking

We suggest an implementation of forward checking (FC) with as flat code structure as possible. We do not introduce special classes for assignment, constraint, domain etc. We rather use already available datastuctures. All procedures can be written within `SudokuSolver.java` file. We introduce 9*9 variables x_{ij} , one for each square. Domain of each variable x_{ij} is $\{0, \ldots, 9\}$, where $x_{ij} = 0$ indicates that the square (i, j) is empty. Otherwise, it indicates the value that has been entered in the square. Every variable is identified with an integer $\{0, \ldots, 80\}$. A function `GetVariable(i,j)` returns this variable id for each square (i, j). Therefore, an assignment *asn* is a list of integers of length 81, where each integer is a number from $\{0, \ldots, 9\}$. An assignment is complete if it contains no zeros, i.e. if $asn[X] \neq 0$ for all $X \in \{0, \ldots, 80\}$. We say that if $asn[X] = 0$ the variable X is unlabelled, and if $asn[X] \neq 0$ variable X is labelled. When selecting the first unlabelled X for assignment in backtracking algorithms, we are actually taking the first variable X from *asn* for which

$asn[X] = 0$.

You are provided with an arc-consistency for forward-checking procedure (`AC-FC(X,V)`), which given a tentative assignment $X = V$ enforces consistency over all the unassigned remaining variables. Differently than the `AC-3` algorithm, `AC-FC` enforces consistency only over the ordered variables that come after X. Consider for example the constraint $x_1 > x_2 = x_3$ with the variable ordering $x_1 \succ x_2 \succ x_3$, where all the variables share the same domain $\mathcal{D} = \{1, 2, 3\}$; calling `AC-FC(`$x_2$`,1)` would result in a reduction of the domain for x_2 and x_3 to $\{1\}$, while nothing would be changed in the domain of x_1. This is a basic building block you can use for implementing the forward checking algorithm `FC(asn)`.

In addition, you are also provided with utility functions that translate between an assignment and a current state of the puzzle: `GetAssignment(puzzle)` and `GetPuzzle(assignment)`. Since RN03 does not discuss the FC algorithm in detail, we provide a pseudocode in Algorithm 1.

Algorithm 1 `FC`(*asn*)

```
if asn contains no 0 then
  return asn
X ← index of first 0 in asn
D_old ← D
for all V ∈ D_X do
  if AC-FC(X, V) then
    asn[X] ← V
    R ←FC(asn)
    if R ≠ fail then
      return R
    asn[X] ← 0
    D ← D_old
  else
    D ← D_old
return fail
```

Implementation Instructions

There is an issue when loading an initial assignment (for example by calling `GetAssignment(puzzle)`). Since forward checking algorithm is incremental (it enforces consistency only wrt. the last assigned variable), the result would not be correct if you start `FC-search` immediately. You should first enforce consistency on all unassigned variables wrt. ALL initially assigned variables. This can be done in a separate function `INITIAL-FC`, that is called immediately before starting the FC search. There is a global variable representing current domains D. Therefore, a function that you need to implement, `FC(asn)`, has a direct access to domains D, even though it is not explicitly passed as the function argument. Therefore, every change to domain D must be explicitly rolled-back when backtracking. This is done by maintaining a separate copy of initial domains D_{old}. Be careful when copying domains $D_{old} \leftarrow D$ or $D \leftarrow D_{old}$: make sure you do create separate objects, such that operations over D will not affect D_{old} and vice versa.

Hand-in

Hand-in the compilable and executable Java code and a short report (max 3 pages) that explains the most important aspects of your implementation. The code should be well-commented and easy to follow. For those not using Java: in addition to providing the code, provide a more detailed report where you explain the overall structure of the code, point to files where you have implemented the required functions. The functions should be well documented (commented).