211hw11_S11

Homework 11: Sudoku Solver

Milestone 1 Due: Mon. 18 April 2011 at 9:59:59 am

Milestone 2 Due: Friday 22 April 2011 at 9:59:59 am

Tip: Out of Memory Error

Try the following: Go to `Edit/Preferences/Micellaneous/JVMs` and add the `-Xss64M` option to the "`JVM Args for Interactions JVM`" line to increase th amount of memory used.

The Sudoku Game

Sudoku is a 9x9 grid-based puzzle in which the goal is to place numbers from 1 to 9 in the grid squares taking into account specific constraints. The 9 x 9 puzzle grid can be seen as divided into 9 sub-grids of size 3x3. The 3 main conditions in the classic version of Sudoku state are that each square in the grid contains a number from 1 to 9 and a number cannot be repeated:

- In any line
- In any column
- In any on the 3x3 sub-puzzles

Sudoku is not limited to 9x9 boards. An "order" for a Sudoku puzzle can be defined, and thus a Sudoku puzzle is an $\texttt{order}^2$ x $\texttt{order}^2$ matrix. A regular Sudoku puzzle is this `order = 3`. The above rules can be generalized to any order. This is Rice, so our system should be able to handle any order of Sudoku puzzle.

For more information of the game of Sudoku, visit: http://en.wikipedia.org/wiki/Sudoku

Need to practice your Sudoku-playing skills? There are lots of on-line games. Here's one: http://www.websudoku.com/

A Sudoku puzzle as needed for this assignment may not be well formed and in general may have no solutions or multiple solutions. Your solution must be able to handle these situations! (You only have to find one solution, if one exists however, not all solutions.)

Supporting Code Download

The support code provided contains a DrJava project, some sample puzzle boards and a simple Junit test.

Download the code here: HW11.zip

Full Javadocs are available in the supplied code in the `docs` folder. Simply open the `index.html` file in your browser.

The assignment

The purpose of this assignment is to build a Java solver that finds one of any possible solutions of the game using a constraint-based approach.

In a constraint-based approach, each square is viewed as a variable that can take on multiple values from a given set. A partial (intermediate) solution is one in which at least one variable has more than one possible value. A final solution is one which each variable has exactly one value (all the sets are singletons).

The required solution (built on the basis of the support code provided) should work as follows. At each step, the program will evaluate a partial solution. Initially, this partial solution corresponds to the input Sudoku puzzle (a 9*9 matrix that is partially filled) in which the values of some squares are known and others are unknown.

Modeling Sudoku

To find all solutions for a partial solution, we have to keep track of all the possible values for each square in the grid. Each square is modeled by an interface, `ICell`.

In general, there are many ways to model a system. Naively, we could simply model the Sudoku board as a NxN square matrix of cells. We could certainly build our system this way and there might even be some advantages to doing so, e.g all out speed if one does a lot of optimization since the underlying processor architecture is optimized for arrays.

But one of the main tenets of modeling is that one's model of a system should should capture and clearly express the salient characteristics of the system. Sudoku is more than a dumb square grid of numbers. Here are some Sudoku-specific features:

- An empty board is fully determined by a single value, it's "`order`", which for a 9x9 board is 3.
- The board is viewable in terms of rows, columns and blocks.
    - Each row, column and block has exactly $\texttt{order}^2$ cells, e.g. for `order` = 3, each row, column and block has 9 cells.
    - Each board has exactly $\texttt{order}^2$ rows, columns and blocks.
- Each cell is a member of exactly one row, one column and one block.

    - No rows share the same cell. Likewise, no columns or blocks share cells. This forces the square matrix layout of rows, columns, and o `rder^2` sub-matrix of blocks if one were to display the rows, columns and blocks where each cell is only shown once.
- The rules of Sudoku stipluate that the value in any given cell has very specific constraints:
    - The valid values are `1...order`^{2} (inclusive), e.g. for order=2, the allowed values are [1, 2, 3, 4]
    - Any valid value in a cell cannot appear in another cell in the row, column or block to which that cell belongs. Conversely, each row, column or block must contain every valid value exactly once.

Notice how rows, columns and blocks are "homomorphic", that is they are exactly the same abstractly and structurally in the above feature descriptions. Nowhere does it say that rows are horizontal, or columns are vertical or that the blocks are arranged in a square sub-matrix -- that's strictly a result of trying to display each cells only once in our visual representation of the board.

What does this all mean? ==> *Rows, columns and blocks can all be represented with the same data structure!* We will use the interface `ICellSet` to represent the set of cells corresponding to a particular row, column or block. There is also the implication that each cell "knows" which specific row, column and block to which it belongs, so each `ICell` has a set (vector) of `ICellSets` that correspond to the row, column and block to which that cell belongs (in that order).

A "board" is therefore a set of three `Vector<ICellSet>`s representing the rows, columns and blocks of the board. Note that any given cell will appear once in the rows, once in the columns and once in the blocks. The data structures that we have created enable us to access the cells in multiple methods that clearly represent the cell relationships in the manner that best models the problem at hand, namely the Sudoku board. We can easily and consistently see the cells in terms of rows or columns or blocks in a manner that is abstractly equivalent, enabling us to easily reuse the same code to process the cells in any direction we choose. Compare this to a standard NxN matrix representation (a collection of rows) which represents only a single cell relationship feature, namely that they are in rows. Luckily, accessing the cells in terms of columns is not too difficult, but accessing them in terms of blocks is a bear!

Solving Sudoku

The fundamental principle of Sudoku is that any given row, column or block must contain all integers from 1 to `order*order` inclusive, exactly once.

In the beginning, a Sudoku board is defined by particular cells already being solved (i.e. have a single value in them) and all the rest being unsolved (i. e. have all possible values in them). We can think of an unsolved cell as containing all possible guesses for the value at that moment, which at the beginning, is all possible values.

Thus the simplest reduction process is to look at the set of all values from the solved cells in the row, column and block to which a cell belongs and eliminating those values from the current set of value choices for that cell. (*Hint*: `HashSet<T>.removeAll()` does exactly what you want!) To reduce a cell set is to reduce every cell in that cell set. To reduce a board is to reduce every cell in *any* collection of cell sets -- do you need to reduce *every* collection of cell sets?

Requirement: the reduction process for a cell set and for a board should be able to detect if the cell set or board is

- Irreducible - the reduction pass did not change any values but at least one cell is unsolved.
- Solved - all the cells were solved already or became solved
- Unsolvable - a cell somewhere has or ends up having no value choices left, i.e. and empty `HashSet` of values.
- Reducible - a change to the number of values for a cell somewhere occurred.

Irreducible Sudoku boards ARE potentially solvable

You must **first** get a **simple** reduction pass working! *Do NOT attempt to solve irreducible boards until you are confident that your simple reductions are working properly!*

But once your simple reduction operational, you can tackle how to deal with irreducible boards. An irreducible board means that a reduction pass on the board causes no change even though the board contains at least one cell that is neither solved nor empty. To solve an irreducible board, you need to pick a particular unsolved cell from the board using some sort of heuristic to choose what you think is the best cell to work with, and test each of its values one by one to see if they lead to a solved board. To test a value, pick on of the values, set the cell to have that one value, i.e. to be "solved" and then try to solve the entire board. If the board is solvable, you're done, otherwise, take the next value choice and repeat.

'But this trial process requires that you be able to return the board back to its original state before you tried the previous choice! Luckily, a `Board` object is capable of making a "deep" copy (`clone`) of itself which you can save away. Unluckily, the method to find a cell candidate (`findMinChoiceCell`) only returns the cell, not its location in the puzzle, so you don't know which cell you actually found (does it matter?). See below for a tip on one way to get around this problem.

Note that the process of solving an irreducible board may itself encounter an irreducible board which you must then solve, so the overall solving process must fundamentally be a recursive process!

See the code details wiki page for explanations of the return types of the reduction and solving methods of `GameModel`.

Supporting Code Details (click to go to next wiki page)

Running the Application

Starting the program:

Start the program by running the* `Controller` class. The supplied DrJava project should already be set to run this class when you click "Run Project". The program will come up with an order 3 board with **empty** cells. You can change the order by changing the number in the "Game Order" text box and press the `Enter` key.

*Entering values:*You can enter values into the cells by clicking on a cell and typing in a number and then either pressing the `Enter` key or by clicking on a different cell.To enter multiple values in a cell, separate the values by spaces and/or commas. You can pre-load guesses this way.To enter all possible numbers into a cell, as the start of a game would require for cells that are not initially supplied, either enter all possible values or enter a dash, -
. Remember that an empty cell means that the puzzle is not solvable! The start of a game should have some cells with single values and the rest with dashes.

Loading values from files:

Clicking the `Load` button will open up a standard file dialog window where you can browse to and select a values file to load.A values file is a normal text file that can be created on any text editor and has the following format:

- Each line of text represents a single row of cells.
- Each cell's values are separated from the other cell's values by a space or tab (tab is recommended because it will keep your values nicely lined up and easy to read).
- Multiple values for a single cell are separated by commas with NO spaces.
- A dash means "all possible values" for the cell, e.g. for an order 2 board, "-" is a shortcut for "1,2,3,4"
- A zero in a cell means that the cell is empty

A standard valid puzzle starting file should therefore have only single values and dashes.

Test files may have empty cells (zero) or cells with specific value choices in them to test specific features of the program.

Available test data:

A number of test data files have been supplied but are in no way to be considered comprehensive or exhaustive!

Saving cell values from the program:

Clicking the `save` button will bring up a standard file dialog window where you can browse to and save the current cell values as a text file.

The program does NOT warn you about overwriting existing files!!

The file format that the program saves is identical to the format used for loading, so cell values can be saved and re-loaded.

Validate the Board:

Clicking the `Validate Button` button will pop up a message dialog box with information on the state of the board. This button, which is fully operational in the supplied code, is a very useful tool to check your code's performance as you work. The information shown includes

- If the board is solved.
- If the board is valid but not solved. The board could be irreducible however.
- The board is invalid
    - The same value appears more than once -- the offending row, column or block is shown
    - An invalid value is present, e.g. a negative number -- the offending row, column or block is shown.
    - A cell is empty - Only one empty cell per cell set is detected. There may be more empty cells present.

All recution and solving operations should always leave the board in a valid state, unless the board is unsolvable, wherein a cell may become empty.

See the note about the `GameModel.validateBoard()` method in the Code Details wiki page!!

Perform a Single Reduction Pass on the Board.

When your `reduceBoard()` method is working, clicking the `Reduce` button will perform a single pass reduction on every cell on the board. You should thus be able to progress towards a solution by repeatedly clicking the `Reduce` button, though you might end up with either an unsolvable or irreducible board. Attempting to reduce an unsolvable or irreducible board will have no effect on the board.

Find a Minimum Choice Cell

When your `findMinChoiceCell()` method is working, clicking the `Find Min Choice Cell` button will pop up a message dialog box displaying the contents of the chosen cell. It does not show the location of the cell because that information is not available. Which cell is chosen depends on the heuristic used. One can manually attempt to solve an irreducible board by getting the values of the cell to be replaced and removing all values except one from the selected cell and then trying to further reduce the cell.

Solve the Board

Once your `solve()` method is working, clicking the `Solve` button will attempt to solve the board from its current configuration. The result will be a pop-up saying that the board is either solved or unsolvable.

During the middle of your implementation, *after* you have implemented the `reduceBoard` but *before* you have implemented the `solveIrreducibleBoar` method, you should do a partial implementation of solve that can solve an easy-level puzzle that never encounters an irreducible board. At that point, the `Solve` button may return a message saying that the board is irreducible.

Generating Solvable Puzzles

A separate, stand-alone, fully-functional utility has been provided that can generate 9x9 Sudoku puzzles from a store of over 50,000 puzzles. The utility consists of a singleton utility class `GeneratePuzzle` and a companion GUI application to run it, `GeneratePuzzleApp}.` `The utilities utilize the files in the {{HW11\data\puzzle_src` folder.

1. .In DrJava, right-click the `GeneratePuzzleApp` class and select "Run File" to start the utility.
2. Select a file from the drop-list (yes, `4,txt` is missing--broken link on the download page). The higher the number, the harder the puzzles.
3. Each file contains about 10,000 puzzles. Use the number spinner to choose or type a puzzle number from 1-10,000 (approx.). Specifying "0" as the index will cause the utility to randomly select a puzzle.
4. Click the `Generate` button and the puzzle will appear in the main display area of the utility.
5. Click the `Save` button to save the generated puzzle in a format that the Sudoku solver application can now read.

This should keep you busy for a while, though your Sudoku solver should be able to easily solve any puzzle in the collection.

Note: Your Sudoku solver should be able to solve a completely blank puzzle (all dashes, not empty)!

Programming Tips

Updating the View

At the end of the `reduceBoard` and `solve` methods, be sure to add the line

```
view.setCellViews(currentBoard.blks);
```

This will update the view with the mutated board. (Note that the `IViewAdapter.setCellViews()` wants the cells in the format of the blocks, not the rows, because that's how the screen is laid out.)

Your board on the screen will NOT reflect the changes of your algorithms unless you add this line!!

Mutating a value outside of an anonymous inner class:

The quandry is that a variable that is declared outside of an anoymous inner class, e.g. visitor implementation, that is accessed by the anonymous inner class must be defined as `final` (*Thanks Sun! Argh...*). There a classic "hack" to get around this:

Define your variable as a one-element array.

For instance:

```
 final int[] x = new int[]{0};

final ICell[] aCell = new ICell[]{ new Cell() };
```

An anonymous inner class can thus access these variables by using the array syntax: `x[0]` and `aCell[0]`

You may find that in order to keep track of several properties as you loop through cells and cell sets, that you may need to create multiple one-element array values.

Controlling a loop from inside of an anonymous inner class

Unfortunately, you cannot `break` or `continue` a loop while inside of an anonymous inner class that the loop completely encloses (i.e. the anonymous inner class is defined in the body of the loop). There are two ways to accomplish this though:

1. Return a `boolean` or other primitive value from the anonymous inner class (which is probably a visitor being accepted). Based on the returned value, break or continue with the loop. Note that returning a value to which you delegate to, e.g. `IUtilHostAB/C/D`, will not work because you will just find yourself inside another anonymous inner class.
2. Use the one-element array trick described above. Set its value from inside the anonymous inner class. After the anonymous inner class returns, break or continue the loop based on that value.

How to make copies of a board when you don't know which cell is the one you want to guess from

The problem is that if you lose the reference to the cell that you chose to test its values, you can never find that cell again without doing the whole search over again. This is because a cell doesn't know where it is in the board (it never needs to know!).

The trick is to keep references to everything you need and to make copies at just the right time. So here's what to do:

1. Keep references to BOTH the original board and the cell you found.
2. Get an array copy of the values in the cell. `Cell.getValueArray()` makes a copy. You can use this array to loop over.
3. In your loop:
    a. Clear the contents of the cell. This will mutate your original board!
    b. Set the value of the cell to the desired test value.
    c. Make a copy of the orignal board.

d. Solve the copy of the board -- you will need to set the current board to the copy. Don't mess up the original board!
e. Repeat with the next test value if no solution or quit if you find a solution.

Requirements

Your assignment is to implement the following methods of `GameModel`:

- `Milestone 1`
  - `reduceCellSet` (15 pts)
  - `reduceBoard` (15 pts)
  - `solve` -- partical solution for easy puzzles that have no irredicible states.
- `Milestone 2`
  - `findMinChoiceCell` (15 pts) -- *Be sure to describe what your heuristic is trying to do in some comments accompanying your code! Simply picking the first or a random cell will NOT garner full credit!*
  - `solveIrreducibleBoard` (20 pts)
  - `solve` - full solution (20 pts)

You are also required to have complete unit tests for the above methods. (15 pts)

Javadocs are already complete in the supplied code. If you add any fields, methods or classes, complete Javadocs will be expected for them. It is suggested that you do some level of Javadocs for your anonymous inner classes, if only to help keep things straight in your own head.

Your code must work for ANY order puzzle! (To within machine limits, of course.)

Testing Procedures

The supplied code contains a simple test routine that gives you examples on how to instantiate a `GameModel`, {[Board]} and test them.

It is reasonable to test your code by the following process:

1. Instantiate a `GameModel` using the supplied dummy `IViewAdapter` object.
2. Load a test output puzzle into the model and then save a reference to model's current board -- this is your reference output board
3. Load a test input puzzle into the model.
4. Run your desired operation.
5. Get a reference to the model's current board -- this is the test output board.
6. Use `assertEquals` to compare the output board to the reference board.

It is highly recommended that you start testing with 2x2 boards. Carefully construct SIMPLE boards that will test just the feature in question.

After you are satisfied with your tests on 2x2 boards THEN move on to 3x3 boards. Note: testing irreducible boards with choices that lead to unsolvable boards may be impossible on a 2x2 board.

The supplied code does NOT contain a full suite of test boards!

Extra credit

If you are attempting an extra credit task, put a note to such at the top of your `GameModel` class. If you don't, the staff might not see it and grade it.

(10 pts) Incorporate the supplied `GeneratePuzzle` utility capability into the main Sudoku solver, both in the view and the model, to enable the user to generate and solve any of the 50,000 supplied puzzles. Remember that the view and the model MUST be kept separate--the only way they can communicate is via `IModelAdapter` and `IViewAdapter`, their respective adapters. *Neither `GameFrame` nor `GameModel` may contain a reference to the other!* (It is actually possible to add this feature without modifying `GameModel` at all, though you are not required to do this.)

(10 pts) As any seasoned Sudoku player knows, there are other reduction rubrics that can be applied to more quickly reduce a board. For instance, if an unsolved cell includes a value choice that no other unsolved cell in its cell contains, then the cell must have that value (this can be extended to a sub-set of values spread across multiple cells in a cell set). Can you incorporate additional reduction rubrics in a modular, flexible and extensible fashion?

(10 pts) Challenge! Come up with a Sudoku puzzle (any order) that your solver can solve but the staff's solution cannot!

Submission

Submit via Owlspace a `.zip` file containing the **entire** HW11 folder, including all the support code and data. Don't forget to add as header to the *`GameModel`* class, your names and IDs. Be sure to mention if you did any extra credit tasks too.