CS 445: Data Structures
Fall 2018

Assignment 3

Assigned: Friday, October 12 **Due:** Thursday, November 01 11:59 PM

---

1 Motivation

In this assignment, you will use backtracking to solve the Sudoku puzzle. If you are not familiar with this puzzle, review the rules here:

`https://www.sudoku.name/rules/en`

2 Provided files

First, carefully read the provided files. You can find these files on Pitt Box in a folder named `cs445-a3-abc123`, where `abc123` is your Pitt username.

2.1 Provided code

The `Queens` class includes a backtracking solution to the 8 Queens problem. If you run this class with the `-t` command line argument (called a *flag*), it will run tests of its `reject`, `isFullSolution`, `extend`, `next` methods.

The `Sudoku` class includes the basic skeleton of a backtracking solution. This class also includes the method `readBoard`, which reads in a Sudoku board and returns it as a two-dimensional 9×9 `int` array, with `0` designating each empty cell (yet to be filled). Finally, it includes method `printBoard`, which prints out a board to the terminal.

It is recommended that you use this provided code as a starting point. However, you may choose not to use this provided code, as long as your program reads the same input format and uses the required backtracking techniques.

2.2 Example Boards

Example Sudoku boards are available for you to test your code in the `boards/` directory.

Each of these files is plaintext (i.e., can be opened with a text editor such as Notepad or TextEdit). A board file contains 9 lines of text, each containing 9 characters. Each character is either a digit (1 through 9) to designate a filled cell or a non-numeric character (such as space or dot) to designate an empty cell. You can also use this format to create Sudoku boards of your own for testing.

3 Tasks

You must write a backtracking program to find values for all of the cells in a Sudoku puzzle, without changing any of the cells specified in the original puzzle. You must also satisfy the Sudoku rules: no number may appear twice in any row, column, or region. You *must* accomplish this using the backtracking techniques discussed and demonstrated in lecture and in `Queens.java`. That is, you need to build up a solution recursively, one cell at a time, until you determine that the current board assignment is impossible to complete (in which case you will backtrack and try another assignment), or that the current board assignment is complete and valid.

Your program will read in the initial board from a file. This initial board will have several cells already filled. The remaining cells will be empty. The goal is to find the values of all the cells in the puzzle, without changing any of the cells specified in the original puzzle. As described above, this file should contain 9 lines of text, each containing 9 characters. Digits 1 through 9 indicate filled cells while non-numeric characters indicate empty cells.

Your class must be named `Sudoku`, and therefore must be in a file with the name `Sudoku.java`. It must also be in the package `cs445.a3`. Your program must be usable from the command line using the following command:

```
java cs445.a3.Sudoku board_file.su
```

If there is a way to solve the puzzle described in `initial_board.su`, your program should output the completed puzzle. You do not need to find every possible solution, if there are multiple solutions—just one solution will do. If there is no solution, your program should output a message indicating as such.

3.1 Required Methods

As stated above, you **must** use the techniques we discussed in lecture for recursive backtracking. As such, you will then need to write the following methods to support your backtracking algorithm. Note that none of these methods is expected to use recursion itself; the `solve` method that uses them implements the recursion.

- `isFullSolution`, a method that accepts a partial solution and returns `true` if it is a complete, valid solution.
- `reject`, a method that accepts a partial solution and returns `true` if it should be rejected because it can **never** be extended into a complete solution.
- `extend`, a method that accepts a partial solution and returns another partial solution that includes one additional decision added on. This method will return `null` if no more decisions can be added to the solution.

Note: Be sure that your `extend` method creates a **new** partial solution, rather than modifying its argument in-place (recall that the runtime stack can only contain references, not objects themselves!).

- `next`, a method that accepts a partial solution and returns another partial solution in which the *most recent* decision that was added has been changed to its next option. This method will return `null` if there are no more options for the *most recent* decision that was made (even if there are other options for other decisions!).

Hints:

1. You should implement the above methods as efficiently as possible. 6 bonus points will be given for solutions efficient enough to solve a very hard puzzle within a fixed time limit. At a minimum, you should be able to solve each of the examples provided within 1 minute.
2. One key challenge in this assignment is differentiating between cells that were filled by your program (and thus *can* be changed) and those that were specified in the provided file (and thus *cannot* be changed). If you were solving a Sudoku by hand on paper, how would you differentiate between these two types of cells? I can think of two main approaches, each of which has a programming analogue.

3.2 Test Methods

When developing a complex program such as this one, it is important to test your progress as you go. For this reason, in addition to the backtracking-supporting methods above, you will be required to test your methods **as you develop them**. Re-read starting at Chapter 2.16 to review the concepts behind writing test methods. To test each of the backtracking methods, you need to write the following test methods. For each one, you should create a variety of partial solutions that cover as many corner cases as you can think of. Then, call the method you are testing on each partial solution, and ensure that the result is as expected. Include enough test cases that the correct output **convinces** you that your method works properly in all situations. Your tests should print out the test cases and results to demonstrate their effectiveness.

- `testIsFullSolution`, a method that generates partial solutions and ensures that the `isFullSolution` method correctly determines whether each is a complete solution.
- `testReject`, a method that generates partial solutions and ensures that the `reject` method correctly determines whether each should be rejected.
- `testExtend`, a method that generates partial solutions and ensures that the `extend` method correctly extends each with the correct next decision.
- `testNext`, a method that generates partial solutions and ensures that the, in each, `next` method correctly changes the most recent decision that was added to its next option.

You may either hardcode your test partial solutions, or submit `.su` files containing the boards you want to test. In either case, when your program is run with the `-t` flag, your program must run each of the above four test methods and output the results.

4 Grading

Your grade for this assignment will be based on:

- Your program's success at finding solutions to a series of unknown Sudoku puzzles (60%). This includes correctly stating that there is no solution, where applicable.
- The thoroughness of your test methods (40%).

5 Submission

Upload your java files in the provided Box directory as edits or additions to the provided code.

All programs will be tested on the command line, so if you use an IDE to develop your program, you must export the java files from the IDE and ensure that they compile and run on the command line. Do not submit the IDE's project files. Your TA should be able to download your `cs445-a3-abc123` directory from Box, and compile and run your code. Specifically, `javac cs445/a3/Sudoku.java` and `java cs445.a3.Sudoku board_file.su` must compile and run your program, when executed from the root of your `cs445-a3-abc123` directory.

In addition to your code, you may wish to include a `README.txt` file that describes features of your program that are not working as expected, to assist the TA in grading the portions that do work as expected.

Your project is due at 11:59 PM on Thursday, November 01. You should upload your progress frequently, even far in advance of this deadline: **No late submissions will be accepted.**