Sudoku Solver - Recursive Backtracking Solver

Submit Assignment

Due Feb 1 by 11:59pm **Points** 25 **Submitting** a website url or a file upload
File Types java

It's finally time to solve your sudoku program! If you're not satisfied with your solution to the last Sudoku assignment, or if you just want to compare your code with mine, check out this **Starter Code (https://bitbucket.org/kelsiec/sudokusolverpart3)** . Optionally fork the starting code **using these instructions**.

Add two new methods to your program, `getNeighbors` and `solve`. Write JUnit tests for your `solve` method.

ArrayList<Board> getNeighbors

This could either be a static method on your solver that takes in a Board parameter, or it could be a method on your Board class that generates neighbors for that particular board. Choose which place to put this based on where you think it would be cleanest. **Write a comment above your function justifying this choice** (in real life you would not need a comment here, but it's a convenient place for you to show your thought process to me).

Given a particular Board, find all valid boards that add just one additional number to a blank cell in the grid. Return the list of neighbors from your function.

solve

Add a solve() method to your main SudokuSolver class. Your `public static void main` should call this method and then print the solution to the console.

Use recursive backtracking to solve your problem. You may need a helper method that does the actual recursive backtracking, or you may not. Whatever method is doing the backtracking should take in a `Board` parameter. First it should determine whether that `Board` is a solution to the problem. If it is a solution return it from the function. If it's not a solution the method should generate valid neighbors and call itself N times with each neighbor.

Either create a member variable Board solution on your class and print that solution in your main, or return a Board representing the solution from your function. **Write a comment above your function justifying this choice**.

Duplicates (Added Fun)

Many search problems of this nature have the issue of duplicate states being reviewed by the search strategy. In the case of this problem two different boards can generate the same neighbor. If we try board A and its neighbors [B1, ..., BN], decide there is no path and then try A' and its neighbors [B'1, ..., B'N], there may exist some overlap between the two sets of neighbors. In order to prevent your program from following down paths that it has considered before you could track all boards that your program has already examined, and when you call your program on a new neighbor first determine that it is new.

Since you cannot get into a **cycle** where you loop infinitely around the same neighbors, detecting duplicates for this algorithm is optional. In other problems you will see in this class this will not be the case.

There is a tradeoff here between runtime and memory. It will take considerable memory to track every board you've already seen, in addition to the memory used by your recursive solution. On the other hand you could be wasting a huge amount of time by not tracking this information. The choice to do so or not is yours, and I consider it **Added Fun** because doing so now will prepare you for future problems in this course.

Testing

This solution takes an incredibly long time. If you take one of the sample boards that I've given you for previous projects and try to find a solution for it, you could leave your computer on overnight doing the calculation and in the morning it might not yet be solved. I highly recommend that you take a completed solution and then remove a few of the numbers and use that as your test case, which will allow it to complete in seconds or less so that you can actually test your code for correctness.

You'll notice that I don't expect you to write JUnit tests for getNeighbors. This is because for a board with 10 blank spaces, there could be as many as 90 different neighbors. Some of these won't be valid but many of them will. It's not reasonable or useful for you to do the data entry to confirm that all of these neighbors come out, and there's not a great way to break down your method so that it is testable.

Instead just write a test runner for solve(). Input a board with a few blanks as I describe above, and verify that it correctly finds the completed solution.