6.005 Elements of Software Construction | Fall 2011
Problem Set 4: Building a Sudoku Solver with SAT

The purpose of this problem set is to give you practice coding in Java, and to introduce you to immutable datatypes and SAT solving. This particular problem set is hard, so start early.

Do not change the signatures, specifications, or reps of any methods, classes, or packages that we have provided you, and do not add new public methods to the classes we have provided. Your code will be tested automatically, and will break our testing suite if you do so.

Background

A *Sudoku puzzle* is a kind of Latin square. The aim is to complete a 9x9 grid with a digit between 1 and 9 in each square, so that -- as in a Latin square -- each digit occurs exactly once in each row and in each column. In addition, the grid is divided into 9 blocks, each 3x3, and each digit must also occur exactly once in each block. Sudoku is normally solved by reasoning, determining one step at a time how to complete an additional square until the entire puzzle is finished. Solving Sudoku by SAT is not very appealing for human players but works well on a computer.

A *propositional formula* is a logical formula formed from boolean variables and the boolean operators and, or and not. The satisfiability problem is to find an assignment of truth values to the variables that makes the formula true.

A *SAT solver* is a program that solves the satisfiability problem: given a formula, it either returns an assignment that makes it true, or says that no such assignment exists. SAT solvers typically use a restricted form of propositional formula called CNF.

A formula in *conjunctive normal form (CNF)* consists of a set of clauses, each of which is a set of literals. A literal is a variable or its negation. Each clause is interpreted as the disjunction of its literals, and the formula as a whole is interpreted as the conjunction of the clauses. So an empty clause represents false, and a problem containing an empty clause is unsatisfiable. But

an empty problem (containing no clauses) represents true, and is trivially satisfiable.

Davis-Putnam-Logemann-Loveland (DPLL) is a simple and effective algorithm for a SAT solver. The basic idea is just *backtracking search*: pick a variable, try setting it to true, obtaining a new problem, and recursively try to solve that problem; if you fail, try setting the variable to false and recursively solving from there. DPLL adds a powerful but simple optimization called *unit propagation*: if a clause contains just one literal, then you can set the literal's variable to the value that will make that literal true. (There's actually another optimization included in the original algorithm for 'pure literals', but it's not necessary and doesn't seem to improve performance in practice.)

Wikipedia articles cover these topics nicely: Sudoku, CNF, DPLL, backtracking search, unit propagation.

Overview

The theme of this problem set is to solve a sudoku puzzle. To do this, we'll:

1. Read in a text file with an incomplete sudoku puzzle, and represent the puzzle using an immutable abstract datatype.
2. Traslate the puzzle into a propositional formula, represented using immutable list data structures.
3. Solve the formula with the SAT problem solver.
4. Translate the formula back into a solution to the Sudoku puzzle.

We're also providing you with packages that include implementations of immutable list data types and some of the propositional formula data types, as well as skeleton implementations of the Formula and Sudoku datatypes, which are in the sat.formula and sudoku package respectively. You only need to fill in the skeleton code to complete this problem set.

Your program should be efficient and should solve the 9x9 puzzles within 5 minutes (when assertions are turned off).

Note: You are **NOT** allowed to use an existing open source SAT solver as a part of your implementation in this problem set.

Problem 1: Loading Sudoku Puzzles

Sudoku is an immutable datatype representing a Sudoku puzzle, with creator methods and

observer methods. We have given you the specification of its methods, and its rep. It's your job to determine the rep invariant and implement the methods.

The datatype also has a factory method for loading a puzzle from a file. The file format is one line for each row of the puzzle, consisting of a sequence of digits (for known squares) and periods (for squares to be filled). We're providing you with two sample puzzles in this format, which are included in your repository along with this assignment.

Important: You will use assertions in this assignment to check rep invariants (see Programming with Assertions). Assertions are turned off by default in Java, which means that assert statements are completely ignored. To make sure assertions are on for all your JUnit tests:

1. Go to Preferences (either Windows / Preferences or Eclipse / Preferences).
2. Go to Java / JUnit.
3. Turn on "Add -ea to VM arguments when creating a new JUnit launch configuration."

If assertions don't seem to be enabled for a JUnit test, then enable them manually:

1. Go to Run / Run Configurations...
2. Find your JUnit class on the left side of the dialog box, and click on it.
3. Go to the Arguments tab, and enter `-ea` in the VM Arguments textbox.

a. [5 points] Write the rep invariant for Sudoku in a comment just after the instance fields, and implement the checkRep() method so that it checks your rep invariant using assert statements. (You can find examples of rep invariant comments and checkRep() in other classes in the provided code, such as Formula and Clause.)

c. [5 points] Write test cases for the two Sudoku constructors in SudokuTest, and then implement the constructors in Sudoku. Call checkRep() in your constructors to make sure the object you constructed satisfies the rep invariant, and implement Sudoku.toString().

d. [5 points] Write test cases for Sudoku.fromFile() and write the implementation of fromFile().

Problem 2: Representing SAT Formulas

Because SAT solving involves a lot of searching through different possible solutions, it turns out to be both more convenient and more memory efficient to implement using immutable data structures -- lists and maps that have no mutator methods. Instead of altering them (for example, adding an element to a list), you return a new object that has the modification, and that often shares much of its structure with the old list or map. In this pset, lists and maps are immutable, and you should NOT use the built-in List or Map classes from the Java API.

In this problem, you will use the immutable lists we provided you to implement formulas in conjunctive normal form.

a. [2 points] Write the datatype expression for Formula in a comment at the top of the Formula class. It should mention Formula, Clause, Literal, PosLiteral, and NegLiteral.

a. [3 points] Write test cases for Formula.

b. [10 points] Implement Formula.

Problem 3: SAT Solving

A SAT solver takes a propositional formula and finds an assignment to its variables that makes the formula true. In this problem, you will implement a SAT solver. You should develop and test your SAT solver independently of the Sudoku problem; i.e., don't feed it formulas produced from Sudoku puzzles (from Problem 4), but choose formula test cases appropriately.

a. [5 points] Write test cases for SATSolver.solve(). Some boolean formulas possible are:

- (a $\vee\neg$ b) $\wedge$(a $\vee$b) should return a: True, b: anything
- (a $\wedge$b) $\wedge$(a $\wedge\neg$ b) should return: null
- (a $\wedge$b) $\wedge$($\neg$ b $\vee$c) should return: a: True, b: True, c: True.

This should get you started, but we expect to see more tests than the above.

b. [25 points] Implement solve(). Here is the pseudocode.

- If there are no clauses, the formula is trivially satisfiable.
- If there is an empty clause, the clause list is unsatisfiable -- fail and backtrack.
- Otherwise, find the smallest clause (by number of literals).
    - If the clause has only one literal, bind its variable in the environment so that the

clause is satisfied, substitute for the variable in all the other clauses (using the suggested substitute() method), and recursively call solve().
    - Otherwise, pick an arbitrary literal from this small clause:
        - First try setting the literal to TRUE, substitute for it in all the clauses, then solve() recursively.
        - If that fails, then try setting the literal to FALSE, substitute, and solve() recursively.

Problem 4: Converting Sudoku to SAT

In order to use the SAT solver to solve a Sudoku puzzle, you need to represent the Sudoku puzzle as a propositional formula, using the Formula datatype you've already created. The variables in the formula are the occupies[i][j][k] variables in Sudoku's rep, which we'll write below as v_{ijk}. When the variable v_{ijk} is true, it means that square[i][j]==k in the final solution of the Sudoku grid.

You want a formula that will require all the following statements to be true, so you need to convert each of these statements into a formula and then AND them together.

- **Solution must be consistent with the starting grid.** You have some known entries in the square[][] array; each of those produces a clause in the formula. If square[0,2] starts out with digit 3, then you would have a clause containing the single positive literal v_{023}, which would require the SAT solver to set it true in the final solution. Don't create clauses for blank cells, so that the SAT solver is free to assign them.
- **At most one digit per square.** Without this requirement, you could get an assignment that sets both v_{000} and v_{001} to true, which would mean that cell (0,0) is occupied by two different digits at the same time. To prevent it, focus on one cell (i,j), look at each possible pair of digits k and k', and add to your formula a clause that guarantees that they can't both be in that square: $\neg v_{ijk} \vee \neg v_{ijk'}$.
- **In each row, each digit must appear exactly once.** For example, let's consider a 4x4 Sudoku grid, and let's focus on row i and digit k. Then the clause $v_{i0k} \vee v_{i1k} \vee v_{i2k} \vee v_{i3k}$ will guarantee that digit k appears *at least once* in row i. To guarantee that it appears *at most* once, we look at every pair of cells in the row, (i,j) and (i,j'), and require that they not both contain k: $\neg v_{ijk} \vee \neg v_{ij'k}$.
- **In each column, each digit must appear exactly once.** Like rows, but fixes the column j and the digit k. Generate one clause that guarantees k appears at least once in the cells of the column, and then one clause for each pair of cells in the column that guarantees they don't both contain k.

- **In each block, each digit must appear exactly once.** Same pattern as rows and columns, but the row and column indexes must vary over the cells within a given block.

Finally, after the SAT solver finds a satisfying assignment of true/false values to the v_{ijk} variables, you will need to convert this assignment back to numbers in a Sudoku grid. For every v_{ijk} that is assigned to true, you should have square[i][j]==k in the final grid.

a. [25 points] Implement Sudoku.getProblem(). Note: we don't require you to put tests for these two methods in SudokuTest, but you may if you want.

b. [5 points] Implement Sudoku.interpretSolution().

Problem 5: Putting it all together

[10 points] Test your code using the sample Sudoku puzzles (and any you want to add) through the Main method, and find out how long it takes your code to solve the puzzle. It should not take more than 5 minutes.

Extra

Make your Sudoku solver faster! (But without changing any method signatures.) This part will not affect your grade.

Before You're Done...

Double check that you didn't change the signatures of any of the code we gave you.

Make sure the code you implemented doesn't print anything to System.out. It's a helpful debugging feature, but writing output is a definite side effect of methods, and none of the methods we gave you to write should produce any side effects.

Make sure you don't have any outdated comments in the code you turn in. In particular, get rid of blocks of code that you may have commented out when doing the pset, and get rid of any TODO comments that are no longer TODOs.

Make sure your code compiles, and all the methods you've implemented pass all the tests that you added.

Does your code compile without warnings? In particular, you should have no unused variables, and no unneeded imports.

Make sure you check the last version of your code in SVN is the version you want to be graded.

Hints

- This problem set is quite challenging. You'll probably not succeed if you just start hacking and hope to make your way through it by brute force. Start early.
- Build your program incrementally. Try a very small SAT problem first that is small enough that you can trace the behavior of your solver with print statements if necessary. Try an unpopulated Sudoku puzzle before you try one that is partially completed, and try smaller puzzles (4x4) before you try the full-sized puzzle (9x9).
- You can create a standard propositional formula and convert it to CNF, but you'll find it easier if you generate CNF directly from the Sudoku grid, as described above.
- The simplest way to encode the puzzle in logic is to create one propositional variable for the possibility that each symbol can be in each square. So for a 9x9 puzzle, there will be 9x9x9 variables. This makes it clear that backtracking search alone will likely be completely infeasible, since the number of leaves of the search tree is 2 to the power of the number of variables.
- To solve the full-sized Sudoku problem you may need to increase the amount of memory that the Java interpreter has allocated for heap space. Recall that every time you run your project (as a Java program, with JUnit, etc.) Eclipse creates a *run configuration* that specifies what to run and how to run it. To increase the maximum heap space for a run configuration, open the Run dialog (**Run** → **Run Configurations...**), select the relevant configuration, select the **Arguments** tab, and enter under VM arguments `-Xmx512m` (for example, which sets the maximum heap size to 512MB).

6.005 Elements of Software Construction

Fall 2011