6.005 Elements of Software Construction
Fall 2008

6.005 Elements of Software Construction | Fall 2008
Exploration 2: Building a Sudoku Solver with SAT

The purpose of this exploration is to give you practice coding in Java, and to introduce you to the case study that will be presented in the lectures on programming with functions and immutable datatypes. Your solution will be judged by (1) correctness -- whether it meets the specification, and (2) clarity -- whether your code is well-organized, carefully commented, and makes proficient use of Java. You are *not* required to use any of the ideas or patterns that have not yet been taught in lecture.

Specification

Your task is to build a program that completes a Sudoku puzzle by (1) translating the puzzle into the problem of satisfying a propositional logic formula, and (2) solving the problem using a SAT solver. Your program should accept a Sudoku puzzle as input, and should display a solution on the console. The format for the puzzle is just one line for each line 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.

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 propositional 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 propositional 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.

Infrastructure

The `exploration2` project in your repository contains this assignment and the sample Sudoku puzzles. Your work should be submitted by committing to this project.

Hints

Here are some hints to help you get the most out of this exploration and make progress quickly:

- This exploration 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, and discuss your ideas with your TA.
- Build your program incrementally. Start with a version that does only backtracking and no unit propagation, and see if you can get it to work on a tiny problem. 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 (2x2 and 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. It's not hard.
- 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 fail, since the number of leaves of the search tree is 2 to the power of the number of variables.
- Use immutable datatypes to represent literals, clauses and clause sets. Search is much easier to program if you don't have to worry about mutation and undoing previous decisions.
- To solve the full-sized Sudoku problem you will probably 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).