Implementation of a Sudoku Solver Using Reduction to SAT

For this project you will develop a Sudoku solver that receives an input puzzle and computes a solution, if one exists. Your solver will:

- read an input Sudoku instance from a file,
- translate the input Sudoku instance into an equivalent SAT instance,
- solve the SAT instance using an existing SAT solver,
- convert the obtained solution of the SAT instance into a solution of the Sudoku instance, and
- print the obtained solution of the Sudoku instance.

This approach may seem like a lot of steps, but it results in a very efficient Sudoku solver and each step is relatively easy. Here are the details.

Step 1

Your program will read from the command line the name of a file, and then read from this file the input Sudoku puzzle (instance). An input file for a Sudoku instance contains:

- Zero or more comment lines at the beginning of the file, each starting with the character ‘c’, followed by
- a line with “3 3” that indicates the dimensions of each box in the puzzle, followed by
- a sequence of 81 integers in the range from 0 to 9. These integers are the values in Sudoku grid in row-major order, with a 0 indicating a cell that does not have a preset value.

Step 2

Your program will then translate the input Sudoku instance into an equivalent SAT instance.

SAT

The Satisfiability problem (SAT) is one of those classic problems that everyone in

CS should be familiar with. SAT deals with boolean formulae. A boolean formula is comprised of boolean variables, the negation operator (boolean **not**), the disjunction operator (boolean **or**), and the conjunction operator (boolean **and**). We use **x1**, **x2,** etc., to denote boolean variables.

A literal is a boolean variable or the negation of a boolean variable.

A clause is a disjunction (or) of literals, or a single literal.

A (boolean) ***formula*** (in conjunctive normal form) is a conjunction (and) of clauses, or a single clause.

An example formula with three boolean variables and four clauses is:
(**x1** or **x2 or x3**) and (not-**x1** or not-**x2)** and (not-**x1** or not-**x3)** and
(not-**x2** or not-**x3)**

An assignment for a formula is a setting of truth values (either **true** or **false**) for the variables in the formula. An example assignment for the example formula is: **x1** is set to **true, x2** is set to **false** and **x3** is set to **false**. We will write this assignment as (**x1, x2, x3**) = (**true**, **false**, **false**).

An assignment that makes a formula evaluate to true is a satisfying assignment. Observe that the assignment (**x1, x2, x3**) = (**true**, **false**, **false**) makes each clause in the example formula evaluate to **true**, and thus makes the example formula evaluate to **true** and is a satisfying assignment. On the other hand, the assignment (**x1, x2, x3**) = (**false**, **false**, **false**) makes the above formula evaluate to **false** since it does not satisfy the formula's first clause, and therefore it is not a satisfying assignment.

A boolean formula is satisfiable when it has a satisfying assignment.

The Satisfiability Problem (SAT) is stated as follows: Given a boolean formula, is it satisfiable?

A boolean formula can model or represent a "system." Let's illustrate this with some examples:

- (x1) and (x2) and (x3), where x1 represents sunny, x2 represents warm, and x3 represents relaxed, models a great beach day.

- (**x1** or **x2**) and (not-**x1** or not-**x2**) has two satisfying assignments, namely, (**x1, x2**) = (**true**, **false**) and (**x1, x2**) = (**false**, **true**). It models **x1** xor **x2**, in other words, exactly one option chosen from **x1** and **x2**.
- The example formula that we saw earlier (**x1** or **x2 or x3**) and (not-**x1** or not-**x2**) and (not-**x1** or not-**x3**) and (not-**x2** or not-**x3**) models exactly one option chosen from **x1**, **x2** and **x3**.

Translating a Sudoku instance into an equivalent SAT instance

The constraints of **a Sudoku puzzle can be modeled using a boolean formula** in conjunctive normal form. There are several ways to translate or encode a Sukoku problem into an equivalent SAT problem. The remaining of this section presents one of these encodings.

Create 729 (9x9x9) different boolean variables $v_{i,j,k,}$ where i, j and k take on values from 1 to 9. Intuitively, the boolean variable $v_{i,j,k}$ is set to true if and only if the cell at row i and column j takes the value k. Recall the constraints on a Sudoku puzzle solution:

1. Each row must have all the digits 1 through 9,
2. Each column must have all the digits 1 through 9,
3. Each 3x3 box must have all the digits 1 through 9,
4. Each cell must have exactly one value in the range 1 through 9.

Constraints 1, 2, and 3 are similar except that each refers to a different subset of cells.

We now present a SAT encoding for Constraint 1. Constraint 1, when applied to a row i, can be viewed as a conjunct (and) of the following nine constraints:

1.1. The number 1 is the value of exactly one cell in row i.
1.2. The number 2 is the value of exactly one cell in row i.
...
1.9. The number 9 is the value of exactly one cell in row i.

We will only address Constraint 1.1 since the other eight constraints can be translated in analogous mann er. Constraint 1.1, in turn, can be broken down as a conjunct of two constraints:

1.1.A. At least one cell of row i has the value 1
1.1.B. At most one cell of row i has the value 1

Constraint 1.1.A, for row i, yields the propositional clause:
($v_{i,1,1}$ or $v_{i,2,1}$ or $v_{i,3,1}$ or $v_{i,4,1}$ or $v_{i,5,1}$ or $v_{i,6,1}$ or $v_{i,7,1}$ or $v_{i,8,1}$ or $v_{i,9,1}$)

Constraint 1.1.B, for row i, yields a subformula comprising 36 clauses of the form
((not $v_{i,j,1}$) or (not $v_{i,k,1}$))
combined together with the *and* operator, for all distinct combinations of j and k, j and k in the range 1 through 9.

Constraints 2 and 3 can be encoded in a similar manner.

Constraint 4, for the cell at row i and column j, is encoded as a conjunction of two constraints:

4.1. The cell at row i and column j has at least one value in the range 1 through 9
4.2. The cell at row i and column j has at most one value in the range 1 through 9

Constraint 4.1, for the cell at row i and column j, yields the propositional clause:
($v_{i,j,1}$ or $v_{i,j,2}$ or $v_{i,j,3}$ or $v_{i,j,4}$ or $v_{i,j,5}$ or $v_{i,j,6}$ or $v_{i,j,7}$ or $v_{i,j,8}$ or $v_{i,j,9}$)

Constraint 4.2, for the cell at row i and column j, yields a subformula comprising 36 clauses of the form
((not $v_{i,j,x}$) or (not $v_{i,j,y}$))
combined together with the *and* operator, for all distinct combinations of x and y, x and y in the range 1 through 9.

Any preset values that appear in a Sudoku puzzle lead to additional clauses in its translation. For example, if the cell in row 3 and column 7 has the preset value 6, we add the clause ($v_{3,7,6}$) to the formula.

A Sudoku puzzle translates in this manner into a SAT instance with 729 variables and approximately 12000 clauses: 37 (36 + 1) clauses from the translation of Constraint 1.1; 333 (37 * 9) clauses from Constraint 1; 8991(333 * (9+9+9)) clauses from all 9 rows, 9 columns, and 9 boxes; 2997 ((9*9)*(36+1)) clauses from Constraint 4; and a few more for the preset values.

For Step 2 of this project, you will write a program that converts a Sudoku puzzle (input in Step 1) into an equivalent SAT instance. Notice that the great majority of the clauses in the formulas for all Sudoku puzzles will be identical, since the only differences from one puzzle to another will be in the clauses that encode their preset values.

Your program will output the computed SAT instance to a file in DIMACS format[1], which can then be given as input to an existing SAT solver. A file in DIMACS format contains:

- Zero or more comment lines at the beginning of the file, each starting with the character 'c'.
- The first data line has the format "p cnf number-of-variables number-of clauses". Note that variables are numbered 1 through number-of-variables.
- The specification of the clauses follows, one clause per line. A clause line ends with a '0'. A clause is specified by writing the numbers of the literals that occur in it. Note that the negated variable number **i** is specified as **–i**; thus, the clause (not-**x12** or **x3** or **x41**) is specified in this file format as –12 3 41 0.

It is customary to give SAT input files the .cnf extension. A sample SAT input file contains:

c This file illustrates the DIMACS input file format, and specifies the formula
c (x1 or x2) and (not-x2 or x3 or x4) and (not-x1 or not-x3 or not-x4)
p cnf 4 3
1 2 0
-2 3 4 0
-1 -3 -4 0

Step 3

You will use an existing SAT solver, a program that receives a boolean formula as input and returns an assignment for it, to solve the SAT instance created in Step 2. I have provided a SATSolver class for you. Its usage is simple once you have the input instance in the format from Step 2, as follows:

int [ ] assignment = SATSolver.solve ("path for encoded Sudoku instance.cnf");

The solve method returns a null array when the formula in the input file is not satisfiable, or an array of integers with a satisfying assignment. For example, the return value [1 -2 -3, 4] encodes the satisfying assignment (**x1, x2, x3, x4**) = {true, false, false, true}.

Step 4

You will convert the solution produced by the SAT solver into a solution of the input Sudoku instance as follows:

- A null array converts into "unsolvable"; otherwise
- The input puzzle is solvable. The value of cell (i, j), for i and j in the range from 1 to 9, is the only k such that $v_{i,j,k}$ is true in the satisfying assignment.

Step 5

Print the solution values in the Sudoku puzzle in row-major order.

Now you can put together a Sudoku solver that reads from the command line the name of the file containing a Sudoku puzzle, computes a solution, and prints the solution or a message that no solution exists. After your own testing convinces you that your solver is correct, run and time your solver on the required input set.

Submission instructions

By the deadline, submit your working solver, evidence of its correctness on the required input set, and a meaningful table with its runtimes.

This material is based upon work supported by the National Science Foundation under Grant No. 1140753.

References

1. The International SAT Competitions, http://www.satcompetition.org/, accessed January 19, 2014.