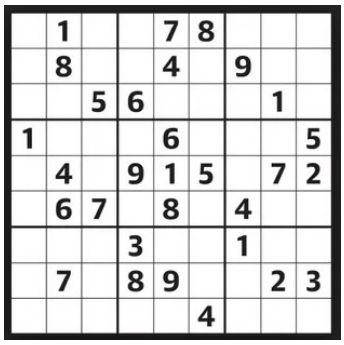
**CMSC 301 Project 3: Sudoku Solver via Reduction to SAT***
**Due 21.May.2019**



## Objective

A Sudoku puzzle consists of a square grid of size n (often n=9), containing preset values in some of its $n^2$ cells. The grid is subdivided into n $n^{1/2}$ x $n^{1/2}$ boxes. The objective of the puzzle is to fill each blank cell so that each column, each row, and each box contains all the digits from 1 to n. Sudoku puzzles range from easy to quite difficult.

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

1. read an input Sudoku instance from a file,
2. translate the input Sudoku instance into an equivalent SAT instance,
3. solve the SAT instance using an existing SAT solver,
4. convert the obtained solution of the SAT instance into a solution of the Sudoku instance, and
5. write an efficient certifier to verify that the solution is a solution to the Sudoku problem.

This approach may seem like a lot of steps, but it results in a very efficient Sudoku solver and each step is relatively easy. In the process, you will work with a real-world application of reduction as well as an approximation solution to SAT. Your verification algorithm should be an efficient certifier of this NP-complete problem. **I give you a solution with most of these steps. You must finish the solution I provide.**

**Download project3.tgz to get all necessary files for this project. I put //TODO on all places in the code where you must do some work. Most of them require work that is VERY similar, so the code will involve some copying and slight revision.**

## Step 1 (Done! Do not alter.)

Your program will read a file name from the command line 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 (You must complete the reduction!)

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

### *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 Sudoku problem into an equivalent SAT problem.

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

The remainder of this section presents one of these encodings.

Create 729 (9x9x9) different boolean variables $v_{ijk}$ where i, j and k take on values from 1 to 9. Intuitively, the boolean variable $v_{ijk}$ 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.

Here is an 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 manner. 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:

($v_{i11}$ or $v_{i21}$ or $v_{i31}$ or $v_{i41}$ or $v_{i51}$ or $v_{i61}$ or $v_{i71}$ or $v_{i81}$ or $v_{i91}$)

Constraint 1.1.B, for row i, yields a subformula comprising 36 clauses of the form

((not $v_{ij1}$) or (not $v_{ik1}$))

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:

($v_{ij1}$ or $v_{ij2}$ or $v_{ij3}$ or $v_{ij4}$ or $v_{ij5}$ or $v_{ij6}$ or $v_{ij7}$ or $v_{ij8}$ or $v_{ij9}$)

Constraint 4.2, for the cell at row i and column j, yields a subformula comprising 36 clauses of the form

((not $v_{ijx}$) or (not $v_{ijy}$))

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 ($v_{376}$) to the formula.
A Sudoku puzzle translates in this manner into a SAT instance with 729 variables and approximately 12,000 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 assignment, you will write a program that converts a Sudoku puzzle (input in Step 1) into an equivalent SAT instance. **You should complete the sections I have marked TODO. Try to be consistent with the code I have provided.** 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, 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. Name the output file using the ".cnf" suffix. For example, if the Sudoku file was TestProblem.sdk, call the output SAT file TestProblem.cnf.

## Step 3 (Done!)

You will use an existing SAT solver, a program that receives a boolean formula as input and returns a assignment for it, to solve the SAT instance created in Step 2.

You could use any SAT solver, but we recommend that you use the given java SATSolver class. Its usage is simple once you have the input instance in the format from Step 2, as follows:

```
int [ ] assignment = SATSolver.solve ("path to the 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}.

The SATSolver class uses sat4j, a well-know public domain SAT solver in Java. You will need to download the sat4j jar file from sat4j.org and insert it into your project.

## Step 4 (Some work to do!)

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 $v_{ijk}$ is true in the satisfying assignment.

## Step 5 (A bit of work to do.)

Write a method called certifier(problem,solution). This method should take two arguments: one

that represents an unsolved Sudoku problem and another that represents a proposed solution to that problem. It should return true if and only if the solution is a valid solution to that input problem. Incorporate your method as the final stage in your program.

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

## Submission instructions

By the deadline, submit your working solver (**SudokuSolver.java**), evidence of its correctness on the 9 puzzles I provided, and a meaningful table with descriptive report (in **report.pdf**) with its runtimes.