CSE 235H SAT Homework 06 Spring 2018

Due: Monday, March 5, 2018 Written by Daniel Geschwender

Programmatically Generating the PL sentence of a Sudoku in CNF (Part 3 of 3):
The goal of this homework is to write a simple program to generate the text file, in the DIMACS CNF 'format,' of a Sudoku instance. The homework is broken into three parts as follows:

- *Part 1*: Manually write a simple CNF file in the DIMACS format; solve with MiniSAT; and write code to generate the first set of clauses that model a Sudoku puzzle.
- *Part 2*: Write code to generate a CNF file expressing all the rules of an empty Sudoku board and solve with MiniSAT.
- *Part 3*: Write code to parse a string representing a partially filled instance of Sudoku. Add the corresponding clauses to the CNF file and solve with MiniSAT.

In this homework, you have to do only Part 3.

Grading Rubric for Part 3:

To Do	Points
Problem A: Code is clear and commented	3
Problem A: Program generates the correct clauses	6
Problem A: #variables and #clauses counts are correct	(1 bonus)
Problem B: CNF files are properly formatted DIMACS CNF	3
Problem B: MiniSAT successfully runs on all three CNF files	3
Total:	15

General Instructions:

- The program must be written in Java and compile and run on Webgrader (`cse.unl.edu/~cse235h/grade/`).
- Your program must use standard input (stdin) and standard output (stdout).
- The model should follow the Sudoku CNF formulation from the textbook (see page 33) and reproduced below.
- The generated output should conform to the DIMACS CNF file specifications described below.
- Submit your code and all accompanying files via handin. No hard copy is required. All submitted files must match the filenames specified in the assignment. Webgrader will use the files submitted through handin and requires exact filenames.
- This homework must be completed individually. LaTeX bonus and partner policy do *not* apply.

The Sudoku CNF Formulation:
We will adopt the following formulation of the Sudoku problem to generate the CNF file:

- The proposition $p(i, j, n)$ indicates that the cell in row i and column j is given value n. In the CNF file, represent $p(i, j, n)$ by ijn. (e.g., $\neg p(3, 8, 7)$ corresponds to -387 in the CNF file)
- Every row contains every number:

$$\bigwedge_{i=1}^{9} \bigwedge_{n=1}^{9} \bigvee_{j=1}^{9} p(i, j, n) \tag{1}$$

- Every column contains every number:

$$\bigwedge_{j=1}^{9} \bigwedge_{n=1}^{9} \bigvee_{i=1}^{9} p(i, j, n) \tag{2}$$

- Every 3x3 block contains every number:

$$\bigwedge_{r=0}^{2} \bigwedge_{s=0}^{2} \bigwedge_{n=1}^{9} \bigvee_{i=1}^{3} \bigvee_{j=1}^{3} p(3r + i, 3s + j, n) \tag{3}$$

- No cell contains more than one number:

$$\bigwedge_{i=1}^{9} \bigwedge_{j=1}^{9} \bigwedge_{n=1}^{8} \bigwedge_{m=n+1}^{9} (\neg p(i, j, n) \vee \neg p(i, j, m)) \tag{4}$$

DIMACS CNF Format Specification:

- Comment lines begin with the character 'c'.
- A problem line must be included before any clauses. The problem line uses the following format: `p cnf <# variables> <# clauses>`.
- Each clause is given by a line of non-null numbers, separated by spaces, and ending with a '0'. The numbers correspond to variables. A negative number represents a negated variable in the clause.

Note on MiniSAT Variables:
Because of the way the variables are specified in our Sudoku model, there are gaps in the numbering of the variables. MiniSAT will see that the highest variable is 999 and will assume that there are 999 variables. This is a 'feature' of MiniSAT. The solution generated by MiniSAT will include all the variables in 1...999. The additional variables (1...110, 120, 130, etc.) should simply be ignored.

Tasks for Part 3

Problem A:
Modify the program you wrote for Part 2 (**GenerateSudoku.java**) by adding to it the following two functionalities:

1. Reading and parsing a Sudoku instance from an input file
2. Generating clauses corresponding to the filled cells and adding them to the clauses that model the Sudoku constraints.

Follow the guidelines below:

- The input file will consist of a single line of exactly 81 characters. Each character corresponds to a cell of the Sudoku board. A digit (1–9) indicates that the cell is filled with the given value. A '.' character indicates the cell is initially empty. The board is expressed in *row-major* order (i.e., the first nine characters correspond to the first row, the second nine characters correspond to the second row, etc.). Figure 1 shows an example input file (File 'example.txt' from `cse.unl.edu/~cse235h/files/SudokuInstances.tar.gz`).

```
4.....8.5.3.........7......2.....6.....8.4......1.......6.3.7.5..2.....1.4......
```

Figure 1: Format of the Sudoku instance input file.

- Each filled cell requires a unit clause representing the assignment of the given value to the corresponding cell. Figure 2 shows the clauses generated from the example input file.

```
114 0
178 0
195 0
223 0
347 0
422 0
486 0
558 0
574 0
651 0
746 0
763 0
787 0
815 0
842 0
911 0
934 0
```

Figure 2: Expected output after parsing the input file from Figure 1.

```
c <Description of problem>
p cnf <#variables> <#clauses>
c <Description of clauses>
<Clauses from Expression (1)>
c <Description of clauses>
<Clauses from Expression (2)>
c <Description of clauses>
<Clauses from Expression (3)>
c <Description of clauses>
<Clauses from Expression (4)>
c <Description of clauses>
<Clauses generated from input file>
```

Figure 3: Structure of Sudoku CNF file.

- Your program must read the file from standard input. The sample GenerateSudoku.java (`cse.unl.edu/~cse235h/files/GenerateSudoku.java`) provides an example. To read from a file (and write to a file) using stdin/stdout on the command line, use the input/output redirection operators (<,>):

  ```
  java GenerateSudoku < example.txt > example.cnf
  ```

- The complete output from your program should be a properly formatted CNF file resembling Figure 3.
- MiniSAT will run on a file with incorrect counts of # of variables and # of clauses but will report a warning. Having correct counts is worth one (1) bonus point.

Problem B:
Execute the following steps:

1. Download the provided Sudoku instances (`cse.unl.edu/~cse235h/files/SudokuInstances.tar.gz`). This folder contains four instances:
   (a) The instance given in Figure 1
   (b) An easy Sudoku instance
   (c) An unsatisfiable Sudoku instance
   (d) The Escargot Sudoku instance, one of the most difficult Sudoku puzzles (`www.sudokuwiki.org/Escargot`).
2. Use your code to generate CNF files (**easy.cnf**,**unsat.cnf**,**aiEscargot.cnf**) for each of the last three instances *excluding* the example.
3. Run each of the three CNF files through MiniSAT. For each CNF, save the MiniSAT results files (**easyResults.txt**,**unsatResults.txt**,**aiEscargotResults.txt**).

Files to Submit to Handin:

- Your code (**GenerateSudoku.java**)
- Three generated Sudoku CNF files (**easy.cnf**,**unsat.cnf**,**aiEscargot.cnf**)
- Three MiniSAT results files (**easyResults.txt**,**unsatResults.txt**,**aiEscargotResults.txt**)

Running on Webgrader:
After submitting your files on Handin, you can run the Webgrader to verify your submission. You can access the Webgrader at `cse.unl.edu/~cse235h/grade/`. The Webgrader script will print the contents of all required files, compile your code (using 'javac -J-Xmx256m GenerateSudoku.java'), run your code (using 'java -Xmx256m GenerateSudoku'), and print the program output.