15-414/614 : Bug Catching, Spring 2014

Homework 1 **Due : 5 Feb 2014 (Wednesday), 3:00 pm (before class)**

Please look at the course website for collaboration policy. Prepare a pdf for problems 1 to 2 separately, preferably using LaTeX. Print out your solutions to problems 1 and 2, bring them to class, and turn in them before the class. For problem 3, you will be submitting code to the TA through emails.

1 Classical Propositional Logic [20 points]

Let x, y and z be three propositions.

(a) (**8 points**) Show that the two propositional formulas, $(x \vee y) \rightarrow z$ and $(x \rightarrow z) \wedge (y \rightarrow z)$ are equivalent using truth-tables.

(b) (**8 points)** Show that the two propositional formulas, $(x \rightarrow y) \wedge (z \rightarrow x)$ and $z \rightarrow y$ are equi-satisfiable, using truth-tables. Also, if the formulas are satisfiable, give one satisfying assignment to x, y and z for each formula.

(c) (**4 points**) In part (b) above, give a simple, but precise, argument for why the two formulas are not equivalent. (*Hint* : Give values for x, y and z for which one formula is true while the other is false.)

2 Normal Forms [20 points]

Recall that, we have discussed about three different normal forms of propositional formulas - NNF, DNF, and CNF - in the class, and the transformation algorithms used to convert an arbitrary logical formula to its CNF.

(a) For each of the following propositional formulas, first write down another formula that is the negation of the given one. Then transform the new formula to NNF such that the connective $\neg$ appears only in front of atomic propositions. For example, to transform the negation of the formula $p \rightarrow q \rightarrow r$ to NNF, you may reason as follows:
$\neg\ (p \rightarrow q \rightarrow r)$
$\equiv\ \neg(\neg p \vee (q \rightarrow r))$
$\equiv\ p \wedge\ \neg\ (q \rightarrow r)$
$\equiv\ p \wedge \neg(\neg q \vee r)$
$\equiv\ p \wedge q \wedge \neg\ r$

   (a) (**3 points**) $p \leftrightarrow q$

   (b) (**3 points**) $p \vee (\neg q \wedge r)$

   (c) (**4 points**) $\neg(p \rightarrow q) \rightarrow (p \rightarrow \neg q)$

(b) Every propositional formula can be transformed to an equivalent formula that is in CNF. This transformation is based on the rules preserving logical equivalence: the double negative law, De Morgan's laws, and the distributive law. However, in some cases this conversion to CNF can lead to an exponential blow-up in the size of the formula. As discussed in the class, for the purpose of checking satisfiability, we can avoid this blow-up by means of another transformation algorithm - Tseitin transformation - which preserves satisfiability instead of equivalence.

For each of the following formulas, apply both the equivalent transformation (using the above mentioned rules) and the Tseitin transformation (as discussed in the class, and you can also turn to `http://en.wikipedia.org/wiki/Tseitin-Transformation`) algorithms to get the corresponding CNFs, and see the difference in the sizes of the resulting formulas.

(a) (**5 points**) $(p_1 \wedge q_1) \vee (p_2 \wedge \neg q_2) \vee (\neg p_3 \wedge \neg q_3)$

(b) (**5 points**) $((p_1 \vee \neg p_2) \wedge (\neg p_3 \vee p_4)) \leftrightarrow (p_5 \wedge \neg p_6)$

3 Sudoku [60 points]

(0 points) Using a SAT Solver.

Most fast SAT solvers use the `DIMACS` input format for a CNF formula. A CNF formula $\phi = C_1 \wedge \ldots \wedge C_m$, where $C_i = \ell_{i,1} \wedge \ldots \wedge \ell_{i,n_i}$, is encoded in DIMACS as follows:

```
c This is a comment line
p cnf <num-vars> <num-clauses>
l_1_1 l_1_2 ... l_1_n1 0
l_2_1 l_2_2 ... l_2_n2 0
...
l_<num-clauses>_1 l_<num-clauses>_2 ... l_<num-clauses>_n<num-clauses> 0
```

Variables are represented by natural numbers (1 and above) and negations of variables by the *minus* sign. Each clause is described in a line terminated by a zero. As an example, the formula
$(x_1) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_4 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_2 \vee x_4)$ would be encoded as:

```
p cnf 4 5
1 0
2 -3 0
-4 -1 0
-1 -2 3 4 0
-2 4 0
```

For this assignment we will use the `MiniSat` SAT solver, which is one of the fastest SAT solvers currently available.

You can find a pre-compiled 32-bit binary executable of the `MiniSat` solver for Linux at `http://minisat.se/downloads/MiniSat_v1.14_linux`. If you are interested, you can also download the latest sources of `MiniSat` from the website and compile it yourself. But for this homework, this binary suffices.

Given a DIMACS input file, say `input.dimacs`, you can run `MiniSat` as follows

```
./minisat input.dimacs output.txt
```

where, `minisat` is the binary for `MiniSat` (`MiniSat_v1.14_linux` for the pre-compiled binary) and `output.txt` is an output file where MiniSat will write `SAT` if the input is satisfiable and `UNSAT` otherwise. In case of SAT, it also outputs a satisfiable assignment. We strongly suggest you to create some small test inputs and play around with the SAT solver to get familiar with it. Some test inputs are also made available here :

`http://www.cs.cmu.edu/~emc/15414-s14/assignment/sample-cnfs.zip`

(60 points) Solving Sudoku with MiniSat

Sudoku is played on an $n \times n$ board, where n is a perfect square. Some cells are prefilled with numbers from 1 to n; the rest are blank. The board is subdivided into $\sqrt{n} \times \sqrt{n}$ blocks. The goal is to fill each cell with a number in such a way that each number from 1 to n occurs exactly once in each row, each column, and each block. While the usual case is to have a unique solution, we generalize the problem so that multiple solutions are also possible. For example, if no cell is prefilled, you can fill the cells however you wish as long as the above mentioned constraints are met.

For this problem, you will write a program which does the following:

1. read a Sudoku puzzle from an input file (input format described below),
2. encode the Sudoku puzzle as a CNF formula in DIMACS format,
3. use MiniSat to find a satisfying assignment to the CNF formula,
4. read the satisfying assignment produced by MiniSat to generate the solution to the original Sukodu problem given as input and output to a file (output format also described below).

We will now describe the format for the input. The first line has just one number n, the size of the puzzle grid, which is guaranteed to be a perfect square. Every other line is a single-space separated list of numbers between 0 and n (inclusive) corresponding to a row of the Sudoku puzzle. A 0 denotes a blank cell. There are n such lines for all the rows. The output format is similar except that the first line specifying n is absent and there are no 0s (as every cell has to be filled).

Here is an example.

Sample Input

```
9
0 6 0 1 0 4 0 5 0
0 0 8 3 0 5 6 0 0
2 0 0 0 0 0 0 0 1
8 0 0 4 0 7 0 0 6
0 0 6 0 0 0 3 0 0
7 0 0 9 0 1 0 0 4
5 0 0 0 0 0 0 0 2
0 0 7 2 0 6 9 0 0
0 4 0 5 0 8 0 7 0
```

Sample Output

```
9 6 3 1 7 4 2 5 8
1 7 8 3 2 5 6 4 9
2 5 4 6 8 9 7 3 1
8 2 1 4 3 7 5 9 6
4 9 6 8 5 2 3 1 7
7 3 5 9 6 1 8 2 4
5 8 9 7 1 3 4 6 2
3 1 7 2 4 6 9 8 5
6 4 2 5 9 8 1 7 3
```

You can use any CNF encoding you want in order to use the SAT solver. Here is a suggested encoding. Let the boolean variable $x_{r,c,d}$ be true iff the number d is in the cell at row r, column c. Encode the following constraints, along with the prefilled cells:
- Exactly one number appears in each cell. (*Hint* : *Exactly one* is the same as *at least one and at most one.*)
- Each number appears exactly once in each row.
- Each number appears exactly once in each column.
- Each number appears exactly once in each block.

Your solver will take five command-line arguments, as follows:

`./solver` *`InputPuzzle OutputSoln MiniSatExec TempToSat TempFromSat`*

where *InputPuzzle* is the file containing the input puzzle, *OutputSoln* is the file to which you will output the solution, *MiniSatExec* is the filename of the MiniSAT executable (e.g., "`MiniSat_v1.14_linux`"), *TempToSat* is the name of the temporary DIMACS file that your solver will produce, and *TempFromSat* is the solution that MiniSAT will generate.

You can write your program in any programming language you desire. You must include a README file with instructions to compile and your program file(s) should be compilable on `unix.andrew.cmu.edu` without any need to install new software.

Some sample 9×9 inputs are at

`www.cs.cmu.edu/~emc/15414-s14/assignment/sudoku-samples.zip`

and a program skeleton in Python is at

`www.cs.cmu.edu/~emc/15414-s14/assignment/sudoku-skel.py`

Use this skeleton should you have any confusion in the above description. It is not intended to serve any other purpose. And you are not required to use Python - we just used it to quickly generate a skeleton.