CSCI2243-Assignment 2

Assigned Thursday, September 7, due Thursday, September 14 at 11PM

The problems are based on Chapter 2 of the textbook. You can skip the section of the chapter about modeling and solving logic puzzles about liars and truth-tellers with propositional logic. (Or, you can read it! It's kind of fun.)

You will need to download some software for this assignment, and you will need to have both Java and Python running on your computer. You need Java to be able to run the `sat4j.jar` satisfiability solver. I've also created a simple Python interface to the satisfiability solver, which should make the second group of problems easier.

Each of the five problems in the Digital Logic section is worth 12 points. Each of parts (a)-(d) of the SAT solver section is worth 10 points.

Part (e) of the SAT solver section is extra credit worth 20 points. But unlike the extra credit from the preceding assignment, this needs to be handed in on the due date of Assignment 2.

1 Digital Logic

You will use Logisim for all but the first problem in this part. I will provide you with a Logisim file containing some circuits that I will demonstrate in class. You will add new circuits to this file, and include the resulting file in the folder you submit with your work.

1. Just as every propositional formula is equivalent to one using only the NAND operator, every propositional formula is also equivalent to one that uses only the NOR (not-OR) operator. We will write

$$(\phi \downarrow \psi)$$

as an abbreviation for this—that is,

$$(\phi \downarrow \psi) \equiv \neg(\phi \vee \psi).$$

(The $\downarrow$ notation was introduced by the American philosopher C. S. Peirce, who first discovered the fact that all propositional formulas could be expressed using only NOR.)

Find a formula, using only the $\downarrow$ operator, equivalent to $p \oplus q$. Show (and check) your work carefully. It will be a mess. I promise that I won't make you do this sort of thing too often!

2. Use Logisim to build a circuit equivalent to an exclusive-or gate, using only 2-input NOR gates. (Comment: (a) You can use your solution from the previous problem, of course. Although the formula might be very long, it is likely to contain a lot of repeated subexpressions. You only have to compute each of these subexpression *once,* because you can feed the output of a gate into several different inputs. (b) I was deliberately mean about this—Logisim will automatically generate circuits consisting only of NAND gates, but not NOR gates. Still, if you're clever about it, you can use the automatically generatef NAND-gate circuits to help you solve this problem....If you figure out what I mean and solve the problem this way, add a text box to the circuit to explain how you did it.)

3. *Not* **that** *again!* The soup-and-salad problem is back for what might just be it's last hurrah: Make a Logsim circuit with three inputs, labeled soup, salad, and cold (make little text labels for them, so I know which is which) and a single output. The output light will go on if the three input switches represent an allowable lunch combination, according to our old criteria for this, and will be off otherwise. You can use any kind of gates that you like.

4. Build a circuit with four inputs and three outputs that computes the sum of two 2-bit integers. In other words, the first two inputs $a \quad b$ represent one summand, the next two $c \quad d$ represent the other summand. (Make sure you label the inputs and outputs on your circuit.) The output $e \quad f \quad g$ represents the sum. Recall the scheme for encoding integers as strings of bits: For two bits, the integers 0 through 3 are encoded by 00,01,10,11 respectively. For three bits, the integers 0

through 7 are encoded by 000,001,010,011,100,101,110,111. (Observe that 7 can never occur anywhere in this problem.) Solve this problem by generating the circuit automatically from a truth table, using only two-input gates.

5. Now solve the last problem *again*, but this time use the canned full adder and half adder circuits provided in the Logisim document posted on the course website—you can make the two-bit adder just by wiring together the half adder and full adder circuits provided. The page containing your circuit will be quite austere—you will just see two rectangles with some wires and switches, instead of a lot of gates.

2 Satisfiability Solver

6. This problem is based on Exercise 11 in the text. I will demonstrate the sat4j solver and the Python interface in class. Here is a brief overview. On the website you will find posted the file `sat4j.jar`, which is the satisfiability solver itself. The satisfiability solver can be run on a CNF specification file `foo` by typing

   ```
   java -jar sat4j.jar foo
   ```

   in either the Terminal utility (Mac) or the Command Prompt (Windows). If you do this, you will see a lot of output, but the punchline is the line starting with `v` after the word `SATISFIABLE`, giving a satisfying assignment that the program found.

   But *don't* do this. I have supplied a Python function called `satsolve` that takes as input a list of lists, representing a CNF specification, creates the appropriate specification file, calls the solver, and returns the satisfying assignment that is found as a Python dictionary. (It returns the empty dictionary if no assignment is found.) In addition, I have provided an example program that is designed to create a CNF specification for the schedule problem in the textbook and call the satisfiability solver. At present, the example program chooses one of the eight available time slots (Wednesday is excluded) for each course without regard to the constraints given in the problem.

   For this part, hand in both the written answers to the questions and the all the Python code you wrote.

(a) Execute the Python function `schedule()`. Copy the satisfying assignment that is returned, and paste this into your writeup. What schedule does this correspond to? (Remember, the constraints about English, Philosophy, Chinese, History and Music, have not been incorporated into the specification.) If you do part (e) of this problem, you can use it to produce the schedule in less painstaking fashion.

(b) You will find that two text files have been added to your directory. One contains the input file for `sat4j` in DIMACS format. The other contains the output, with a lot of stats on the performance of the solver along with the solution it found. Examine the specification file. Translate the first line of the file

    1 2 3 4 5 6 7 8 0

into both a standard formula of propositional logic, and into succinct English, using the given interpretations of the variables. Do the same for the lines

    -12 -15 0

and

    -20 -44 0

(c) Now edit the code for the `schedule` function file so that it incorporates the four additional constraints listed in the text. Run the program again, and include both the satisfying assignment and the resulting schedule in your answer. You should find that it conforms to the specifications given in the text. (Do not forget to include the constraints about History, which were left out of the discussion on pp. 46-7 of the text.)

(d) Edit the program one more time so that it produces a schedule *different* from the one it found in (c): For instance, if the solution said to make variable 1 True, and variable 12 True, and variable 23 True, etc., you can add a constraint that says one of these has to be False. You should get a different schedule consistent with the student's requirements.

(e) (*Extra credit*). Devise a smart back end for this program: That is, write a function that takes the output of `satsolve` and produces a nicely formatted schedule. If you do this, then you can solve parts (a),(c),(d) above with your new function.