EECS 219C: Formal Methods **S. A. Seshia & K. Cheang**

Homework 1: SAT & BDDs

Assigned: February 6, 2019 *Due on bCourses: February 20, 2019*

Note: For this and subsequent homeworks, if a problem requires you to come up with an algorithm, you should prove your algorithm's correctness as well as state and prove its asymptotic running time. See the webpage for rules on collaboration.

1. **Horn-SAT and Renamable Horn-SAT** (40 points)

   (a) Recall from class that a HornSAT formula is a CNF formula in which each clause contains at most one positive literal. Give an algorithm to decide the satisfiability of HornSAT formulas in linear time (in the number of variables n).

   (b) A renamable Horn formula is a CNF formula that can be made into a Horn formula by complementing some of its variables (i.e., a variable is replaced by its complement, and if the formula turns out to be satisfiable, the relevant bits of the satisfying assignment are complemented to get a solution to the original problem). For example, consider the CNF formula

   $$(x_1 \vee \neg x_2 \vee \neg x_3)(x_2 \vee x_3)(\neg x_1)$$

   This is neither Horn nor negated Horn, but if we complement x_1 and x_2, it turns into a Horn formula.

   Give a polynomial-time algorithm to check whether a formula on n variables comprising m CNF clauses is renamable Horn. (Hint: try to express this problem itself as a SAT problem.)

2. **The Pigeon-hole Problem** (60 points)

   The *pigeon-hole SAT problem* expresses the problem of finding a way to place n pigeons in $n-1$ pigeon-holes such that no hole contains more than one pigeon. Obviously, this problem is unsatisfiable. If x_{ij} is true when pigeon i is placed in hole j, then this problem can be written as

   $$\bigvee_{j=1}^{n-1} x_{ij}, \quad \text{for all } i \in \{1, 2, \ldots, n\}$$
   $$\neg x_{ik} \vee \neg x_{jk}, \quad \text{for all } k \in \{1, 2, \ldots, n-1\},\ i, j \in \{1, \ldots, n\}, i \neq j$$

The first set of CNF clauses require a pigeon to be placed in some hole, while the second enforce the constraint that no hole contains more than one pigeon.

Interestingly, the basic DPLL search algorithm (without learning) will take exponential time (in n) to decide the unsatisfiability of the pigeon-hole SAT problem (for $n \geq 4$), no matter what order variables are branched on. (If you're interested, try to prove it. Hint: use an inductive argument)

(a) **Using SAT solvers** (30 points)

This exercise will familiarize you with using SAT solvers and help you understand how they work.

Encode the Pigeon-hole SAT problem describe above in the DIMACS CNF format for $n = 4, 5, 6, ..., 15$. (Write a program to generate these files.)

(You can simply search for "DIMACS CNF" to find out more about this format.)

Download a SAT solver (e.g., MiniSat) and learn to run it.

Run the SAT solver on the Pigeon-hole CNF files and plot how the runtimes vary with n. Describe your observations.

If you write a script to generate CNF files (as you must!), submit that script.

(b) **BDDs** (30 points)

In this problem, you will use BDDs to repeat the above part (a). Construct the BDD for the CNF formula and verify that it simplifies to *false*.

Describe what happens when you increase n. Does the variable order matter?

Memory required by the BDD package may blow up after some point, and if that happens, you obviously need not continue beyond that value of n.

Submit your code for constructing the BDD with this assignment.

We suggest you use the Python-based dd package: which contains a fully Pythonic implementation of Binary Decision Diagrams. We also provide skeleton code for this part of the assignment.

However, if you so choose, you can also use other interfaces to BDD libraries such as CUDD from the University of Colorado, Boulder or other BDD packages available online.

INSTALLATION INSTRUCTIONS:

1. If you don't already have it, install pip. The pip website is at `https://packaging.python.org/key_projects/#pip`. You can check if you have pip by running the command '`pip --version`'.
2. Install the dd package: '`pip install dd`'. You may need to run this command with superuser privileges ('`sudo pip install dd`').
   Note: if you are using Python 3.x, you may need to invoke `pip3` instead of `pip`. The skeleton script we have provided has been tested with Python 2.7 and Python 3.6.

3. Test if the installation works by running '`python hw1.py`'.
   Note: if you run into package errors with pydot or visualization, you may want to try (re)installing the pydot and graphviz python packages using pip/pip3.

EXAMPLE CODE:
Example Python code is provided, and can be found in the folder Homeworks/HW1 under Files on bCourses. There are two examples provided in the code to introduce you to the dd package. If you need more information, visit the dd package's documentation page on GitHub: `https://github.com/johnyf/dd/blob/master/doc.md`. The usage instructions for `hw1.py` are:

```
$ python hw1.py --help

usage: hw1.py [-h] [--example {1,2,3}] [--n N] [--pdf PDF]

optional arguments:
  -h, --help          show this help message and exit
  --example {1,2,3}   Example to run (1-3). Default=1.
  --n N               Value of n (default=2). (Only for examples 2 and 3.)
  --pdf PDF           PDF image output filename (Only for example 1.)
```

Replace example 3 (the pigeonhole function) with your solution.

(c) **Extra Credit** (15 points)

Does the way pigeons and holes are encoded affect the scalability of the BDD-based proof? The encoding described above uses one variable for each pigeon/hole combination, resulting in $n \times (n-1)$ variables in total. Does an encoding which uses $O(n \times log\ n)$ variables help?

Reformulate the problem using such an encoding and compare its scalability to the other encoding.