CS 3100, Fall 2020, Asg-11, 100 pts

Please fill the notebook `u1234567_asg11_Prob1234.ipynb` within `21_NPC_Lambda` and submit that. You are also required to submit a `.png` file called `bdd1.png`.

1. **(25 points), AR** The topic of this assignment problem is *Running the BDD tools located at*
   `http://formal.cs.utah.edu:8080/pbl/BDD.php`
   to understand Boolean logic, and the core of NP-completeness reductions.

   Someone wants to implement the following functions:

   - The magnitude comparison function `<` (written **lt**) defined over bits `a3,a2,a1,a0` and `b3,b2,b1,b0`. In particular, `a3 a2 a1 a0 < b3 b2 b1 b0` must be true exactly when the magnitude of the unsigned binary nibble `a3 a2 a1 a0` is less than that of the nibble `b3 b2 b1 b0`.

     Examples: `0100 < 1100`, `0100 < 0101`, etc.

   - The magnitude comparison function `>` (written **gt** in the BDD file) defined over bits `a3,a2,a1,a0` and `b3,b2,b1,b0`. It is defined similar to how `<` was defined.

     Examples: `1100 > 0100`, `0101 > 0100`, etc.

   - The equality function `=` (written **eq** in the BDD file) defined over bits `a3,a2,a1,a0` and `b3,b2,b1,b0`.

     Examples: `1100 = 1100`, `0101 = 0101`, etc.

   - Someone using our BDD tool has come up with the following definitions. Your task is to help the person debug his/her construction. Due to the symmetry of their construction, they trust `lt1` and `gt1` to be both right or wrong. Similarly, they trust `lt2` and `gt2` to be both right or wrong. They also trust `eq`'s definition. They need help with which pair `lt1,gt1` or `lt2,gt2` to trust. They also need help with size-control of the BDD:

```
#1 Var_Order : a3, a2, a1, a0, b3, b2, b1, b0
#2 Var_Order : something else

lt1 = (~a3 & b3) | (~a2 & b2) | (~a1 & b1) | (~a0 & b0)
gt1 = (a3 & ~b3) | (a2 & ~b2) | (a1 & ~b1) | (a0 & ~b0)

lt2 = ~a3 & b3 | (a3<=>b3) & (~a2 & b2 | ((a2<=>b2) & (~a1 & b1 | (a1<=>b1) & ~a0 & b0)))
gt2 = a3 & ~b3 | (a3<=>b3) & (a2 & ~b2 | ((a2<=>b2) & (a1 & ~b1 | (a1<=>b1) & a0 & ~b0)))

eq = (a3 <=> b3) & (a2 <=> b2) & (a1 <=> b1) & (a0 <=> b0)

#3 Main_Exp  : lt1 & gt1
#4 Main_Exp  : lt2 & gt2
#5 Main_Exp  : (lt1 & ~gt1) <=> lt1
#6 Main_Exp  : (lt2 & ~gt2) <=> lt2
```

(a) **(5 pts)** Which `Var_Order` (`#1` or `#2`) do you recommend, and why? What is a good criterion for picking a variable ordering that results in smaller BDD sizes? (about 2 bullets)

(b) **(5 pts)** When you enable `#3` as the `Main_Exp`, does the resulting BDD suggest that `lt1` and `gt1` are correct? Also, when you enable `#4` as the `Main_Exp`, does the resulting BDD suggest that `lt2` and `gt2` are correct? Explain in clear bullets (about 2) by reading the resulting BDDs, saying which of the expressions are correct (`lt1,gt1` or `lt2,gt2`).

(c) **(5 pts)** When you enable `#5` as the `Main_Exp`, does the resulting BDD suggest that `lt1` and `gt1` are correct? Also, when you enable `#6` as the `Main_Exp`, does the resulting BDD suggest that `lt2` and `gt2` are correct? Explain in clear bullets (about 2) by reading the resulting BDDs, saying which of the expressions are correct (`lt1,gt1` or `lt2,gt2`).

(d) **(5 pts)** What is the main flaw in `lt1/gt1`? (about 2 bullets)

(e) **(5 pts)** How is this flaw fixed in `lt2/gt2`? (about 2 bullets)

2. **(25 points), XL** Refer to Figure 16.9 of our book. Here, a formula called ϕ is given. Drop the last conjunct of ϕ, calling the resulting formula ϕ_1.

(a) **(5 points)** Enter this formula in the syntax of the provided BDD tool. Answer in your notebook, in the space provided, how you entered this formula.

```
Var_Order : x1 x2
...
```

(b) **(5 points)** Submit the PNG file you obtain for this BDD in your ZIP as a file called `bdd1.png`. You can save the image by right-clicking on it.

Next, obtain the satisfying assignment for ϕ_1. Write it in the syntax

```
x1 = Value (True or False)

x2 = Value (True or False)

x3 = Value (True or False)
```

as given by the outline in the notebook.

(c) **(10 points)** In the NP-completess proof, there is a mapping reduction employed in Figure 16.9. Describe the cliques generated by ϕ_1 according to the construction rules of this mapping reduction.

How many 3-cliques (or *triangles*) are allowed by ϕ_1? (Answer in the notebook)

Also list them as follows, one per line. Recall that a literal is something like `x` or `!x`.

List as `x1,x1,x2: literal`, `x1,x1,!x2: literal2`, `!x1,!x1,x2: literal3`

(d) **(5 pts)** Suppose someone comes up with a P-time solver for cliques. How does this allow you to obtain a P-time solver for 3-SAT? Describe

in **two clear sentences** reflecting your understanding. Use any two sentence forms to express: we just want to see how you are thinking.

3. **(25 points, LT)** SAT, while being NP-complete, is a "workhorse of a tool." This problem asks you to get a taste of running a SAT tool and seeing how things are encoded. Specifically, you will be running CryptoMinisat on a SAT formula. You don't need to install this tool: merely go to page 265 of our book, consult Figure 16.10, and presto—there is a link to this tool that you can click! When you do this, the tool comes up with a prefilled formula. There is a Play button that you can click whereupon it solves the SAT instance.

This assignment asks you to replace this SAT instance with something bigger: specifically, the Pigeonhole problem (`hole6.cnf`) from `https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html`. Just click the above link, and get the `hole6.cnf` file, and plunk the CNF into the buffer.

Hit "play" and report on the execution time (you can look at your phone's clock). If under 2 seconds, say "negligible" for your answer!

How much time would such a problem take through brute-force enumeration of 2^n combinations on a computer that takes a microsecond per variable combination (the n is the number of variables used in the Pigeonhole problem)? **HINT:** Here is how you read the contents of a CNF file:

```
c File:  hole6.cnf <--- these are comment lines - starts wth a "C"
c...
c
p cnf 42 133     <--- CRUCIAL !! Tells you there are 42 variables and 133 clauses
-1     -7    0   <--- This line says (!x1 + !x7). The "0" is just end-of-a-clause marker!
-1     -13   0   <--- This line says (!x1 + !x13)
...
 12     11     10     9      8      7    0 <--- This clause reads
                                           (x12 + x11 + x10 + x9 + x8 + x7)
...
```

(a) **(5 points)** CryptoMinisat runtime: **SOLUTION: some time OR Negligible.**

(b) **(5 points)** 2^n runtime estimation. (Your solution here.)

(c) **(15 points)** List six facts that you found interesting about Boolean SAT in these articles:

`https://cacm.acm.org/magazines/2009/8/34498-boolean-satisfiability-from-theoretical-hardness-to-practical-success/fulltext`

and

`https://en.wikipedia.org/wiki/Boolean_satisfiability_problem`

Anything that interested you is fine – theoretical or practical. Please offer 1-2 sentences per point that interested you.

4. **(25 points), SV** Euclid's method to compute the greatest common divisor of two natural numbers can be specified in the Lambda syntax as:

```
gcd = lambda x: lambda y: y if (x==y) else gcd(x-y)(y) if (x>y) else gcd(x)(y-x)
```

   (a) **(5 points)** Much like we computed `fact` to be `Ye(prefact)` (see Chapter 18), compute the following: `pregcd` using a `Ye` application. Notice that `pregcd` is *curried* (Page 311 defines *curried functions*); but that does not matter yet (computing `gcd` from `pregcd` works the same despite having a curried function of two arguments).

   Define `pregcd` in this manner, and then `gcd`

   (b) **(5 points)** Evaluate these:

   - `gcd(450)(6000)`
   - `gcd(450)(6001)`
   - `gcd(450)(6002)`
   - `gcd(450)(6003)`
   - `gcd(453)(6003)`

   (c) **(15 points)** (In this problem, we use () or [] interchangeably, for visual clarity.) Show that Y_e is indeed a fixed-point combinator. That is, show that for any G, we get

$$Y_e\, G = G(Y_e\, G)$$

   Here are a few steps of the derivation; finish this:

   - $Y_eG = (\lambda f.(\lambda x.(xx)[\lambda y.f(\lambda v.((yy)v))])G$
   - *(Apply Beta reduction to pull in G, and get)*
   - $= (\lambda x.(xx)[\lambda y.G(\lambda v.((yy)v))])$
   - ... finish the remaining steps; it will involve two more Beta reductions and one Eta reduction.