Assignment 6

CSC 530 with Professor John Clements
Joe Emenaker, AJ Raftis

Section 1

Q.1

See *hw1_q1_classify.rkt* in the git repo. Basically, the classification conducts at least one, and maybe two, tests: We first test if we can solve F (that is, find an assignment of the variables in F which make it evaluate to *true*). If we *cannot*, then F is always false and is a CONTRADICTION.

If we *can* find an interpretation that satisfies F, then it's either *always* true or *sometimes* true. So, we try to solve for ¬F. If we can find an interpretation which makes ¬F evaluate to *true*, then F is a CONTINGENCY (we can assign values that make it evaluate to true or to false). If we *cannot* make ¬F be true, then F is *always* true and, hence, is a TAUTOLOGY.

Q.2

See *hw1_q2_tseitin_encoding.rkt* for the full conversion.

We are asked to convert:

$$\neg(\neg r \rightarrow \neg(p \wedge q))$$

to CNF via Tseitin' encoding. This converts to:

$$\neg x1$$
$$\neg x1 \vee r \vee \neg x0$$
$$\neg r \vee x1$$
$$x0 \vee x1$$
$$\neg x0 \vee p$$
$$\neg x0 \vee q$$
$$\neg p \vee \neg q \vee x0$$

To check our work, we used the 'classify' function developed in Q.1. Basically, set up a *bi-implication* between the original expression and the CNF we found, then we run the classifier on it, expecting TAUTOLOGY.

```
(define comparison1 (<=> new_exp old_exp))
(classify comparison1)  ; This produces CONTINGENCY. Why?
```

Interestingly, we got CONTINGENCY. After some digging, we surmised that this was because the original expression isn't affected by the auxiliary variables introduced in the Tseitin transformation. Therefore, with an assignment for *p*, *q*, and *r* which satisfy the *original* expression, it's still possible to find assignments for x0 and x1 which make the converted CNF *true or false*.

The solution was to add constraints for x0 and x1 in the *original* expression:

```
(define old_exp_w_x0_x1
     (&& old_exp
          (<=> x0 (&& p q))
          (<=> x1 (=> (! r) (! (&& p q))))))
(define comparison2 (<=> new_exp old_exp_w_x0_x1))
(classify comparison2) ; <-- THIS will give TAUTOLOGY
```

Thankfully, this gave TAUTOLOGY, assuring us that the converted expression is equisatisfiable to the original.

Q.3

First, a brief discussion of the implications of NNF and the *pos()* predicate proposed in the homework.

In NNF form, *all negations* accompany *variables only, never larger expressions*.

With the *pos()* predicate, we form a set of all literals which have a value of true *after applying any negation which may be accompanying the variable*.

What we're asked to prove is that, if *pos*(I, ϕ) $\subseteq$ *pos*(I', ϕ), then I satisfying ϕ means that I' *also* satisfies ϕ. Intuitively, this makes sense, since *I* has enough positive literals to make ϕ evaluate to true. *I'* can only have *the same or more* true literals, which (since no non-literal expression can be negated), adding more true literals can only make more expressions true, and (since these expressions are never negated) this true-ness can only contribute to more true-ness of the overall expression.

To prove it, let's look at the grammar for NNF:

Atom := Variable | true | false
Literal := Atom | ¬Atom
Formula := Literal | Formula op Formula
op := $\wedge$ | $\vee$

Since we're never really going to have true or false in an expression (otherwise, we could just simplify the expression to leave them out), and since we only have two operators, we can simplify our grammar:

Literal := Variable | ¬Variable
Formula := Literal | Formula ∧ Formula | Formula ∨ Formula

We can now show that, for every case of the non-terminals, if *pos*(I, ϕ) ⊆ *pos*(I', ϕ) and I causes that non-terminal to be true, then I' will *also* cause that non-terminal to be true.

For a *Literal*: Literal => *true* iff Literal ∈ *pos*(I, ϕ). It is pretty much the definition of a subset that, if *element* ∈ S, and S ⊆ S', then *element* ∈ S', so Literal ∈ pos(I', ϕ), so Literal => true under I'.

For Formula ∧ Formula: Formula1 ∧ Formula2 => *true* iff *pos*(I, ϕ) contains the elements necessary for Formula1 to be true AND for Formula2 to be true. In this case, *pos*(I', ϕ) *also* contains these elements, so Formula1 ∧ Formula2 => *true* under I'.

For Formula ∨ Formula: Formula1 ∨ Formula2 => *true* iff *pos*(I, ϕ) contains the elements necessary for Formula1 to be true OR for Formula2 to be true. In this case, *pos*(I', ϕ) *also* contains these elements, so Formula1 ∨ Formula2 => *true* under I'.

Q.4

We're asked to show that the *single*-implication Tseitin transformation of an NNF expression is equisatisfiable to a *bi*-implication form. In other words, the expression on the left is satisfiable if *and only* if the expression on the right is satisfiable:

x_0 $x_0 \rightarrow p \vee x_1$ $x_1 \rightarrow q \wedge r$	x_0 $x_0 \leftrightarrow p \vee x_1$ $x_1 \leftrightarrow q \wedge r$

I was able to find a stackexchange post dealing with this very problem (https://math.stackexchange.com/questions/2909321/one-sided-tseytin-encoding-is-equisatisfiable), but I found it difficult to understand the approach used.

Instead, I tried the following (although I got stuck): It is easy to see that, if the *bi-implication* is satisfiable, then the *single-implication* will be also, since bi-implications turn into two single-implications. In other words, **the single-implication version should have a little as *half* of the criteria of the bi-implication version**. To write this as a truth table, we can see that, any time p↔q is true, p⟶q is *also* true:

p	q	p↔q	p⟶q
T	F	F	F
T	T	T	T
F	T	F	T
F	F	T	T

So, what remains is to show that it is impossible to satisfy the *single-implication* without also satisfying the *bi-implication*. There is only one case when a single-implication is satisfied and a bi-implication is not: when the left side is *false* and the right side is *true*. **So, we must show that, for all forms of a Tseitin transformation, it is *impossible* to have a case where the right side of a single-implication is true and the left side is false**.

One way to show this is to add those constraints (that being the left side being false and the right side being true) and then show that the set of constraints is unsatisfiable.

There are three forms of expression to consider: a literal (eg. 'p'), a conjunctions (eg. $\alpha \wedge \beta$), and disjunctions of two expressions (eg. $\alpha \vee \beta$):

Case 1: A literal

Tseitin transformations don't convert bare literals, so there is no introduction of new implications.

Case 2: Conjunctions

The expression ($\alpha \wedge \beta$) converts to

x_0
x_1
$x_0 \longrightarrow \alpha$
$x_1 \longrightarrow \beta$

We want to see if it's possible for either of the implications to have their right side be true while the left side is false, so we want to add the criteria: $(\neg x_0 \wedge \alpha) \vee (\neg x_1 \wedge \beta)$ and converting that to CNF, we get $(\neg x_0 \vee \neg x_1) \wedge (\neg x_0 \vee \beta) \wedge (\alpha \vee \neg x_1) \wedge (\alpha \vee \beta)$

x_0
x_1
$\neg x_0 \vee \alpha$
$\neg x_1 \vee \beta$

$\neg x_0 \vee \neg x_1$
$\neg x_0 \vee \beta$
$\alpha \vee \neg x_1$
$\alpha \vee \beta$

Immediately, we can see that lines 1 and 2 contradict line 5. So, it is impossible to satisfy the *single-implication* Tseitin transformation (of an NNF formed) conjunction.

Case 3: Disjunctions

The expression ($\alpha \vee \beta$) converts to

$x_0 \vee x_1$
$x_0 \longrightarrow \alpha$
$x_1 \longrightarrow \beta$

Our additional requirement is a little more stringent, this time. We must add the same criteria that we did in case 2, that *either* of the implications have the $\bot \longrightarrow T$ case.

$x_0 \vee x_1$
$\neg x_0 \vee \alpha$
$\neg x_1 \vee \beta$
$\neg x_0 \vee \neg x_1$
$\neg x_0 \vee \beta$
$\alpha \vee \neg x_1$
$\alpha \vee \beta$

This is where I get stuck, since this is satisfiable (by x_0, $\neg x_1$, α, β), so I'll definitely need to ponder this some more.

Section 2

Q.5

We were asked to trace the following CNF:

$$(\neg x_1 \vee x_2 \vee x_4) \wedge (x_1 \vee x_3) \wedge (\neg x_4 \vee \neg x_2) \wedge (\neg x_4 \vee \neg x_1 \vee x_2) \wedge (x_3 \vee \neg x_1) \wedge (\neg x_3 \vee \neg x_2 \vee x_4) \wedge (x_1 \vee x_4) \wedge (\neg x_2 \vee x_1)$$

Using CaDiCaL's log output in order to create a .dot file, which we'll use to create an implication graph using GraphViz

Primary Steps:

1. Install CaDiCaL. This works pretty much as described in the homework. The biggest challenge was just remembering how to get on the correct tag via Git. I

had a couple minor issues here, as I was actually on the correct tag, but thought I wasn't, which resulted in my futzing around a bit with the incorrect tags.

2. Install GraphViz. This was easily accomplished via MacPorts.
3. Test. I did two tests:
    a. Make sure we can use CaDiCaL's logging, so I did this by testing the sample.cnf provided in the HW. Basically command: "cadical -l sample.cnf". This produces the expected log output.
    b. Make sure I can use GraphViz. This was done by generating output for impl-graph.dot. Basic command: "dot -Tpdf impl-graph.dot -o test.pdf", although changing -Tpdf to -Tpng will probably be preferred if I can get the actual graph for the above problem working.
4. Then, since impl-graph.dot was supposed to be a representation of sample.cnf, I thought I'd figure out how the log output from sample.cnf was translated to the .dot file. Unfortunately, after *way* too much time spent, I finally realized that the graph presented isn't the same graph that would be generated by CaDiCaL. It may have been at one point, but I noticed that the master branch of CaDiCaL was generating a different result than our branch, so apparently there's been tweaks made to CaDiCaL that change how the algorithm works precisely.
5. At this point, I gave up trying to match the .cnf and .dot files, so I went ahead and created the CNF file for the HW's problem:

```
c Sample CNF for HW 1, Problem 2
p cnf 4 8
-1 2 4 0
1 3 0
-4 -2 0
-4 -1 2 0
3 -1 0
-3 -2 4 0
1 4 0
-2 1 0
```

    a.
6. Anyways, at this point, based on a post by the Prof on Piazza, I decided I needed to go back and read the lecture 4 slides.

So, at this point, I pretty much got stuck. Stuff started to gel a little bit more after the last Wednesday lecture, but not quite enough to get everything to fall into place. I even re-read the slides.

The best I've come up with is I think the nodes for the problem will be 1@1, 3@1, 2@2, -4@2, -2@1, 4@1, -1@0, 3@0, 4@0, and -2@0. I'm also (I think) following some of the lines, for example, I'm pretty sure we have a link between 1@1 → 3@1 via "-1 3".

Section 3

Q.6

a) To assert that every node is colored, we can merely state that (color(v)=c_1 OR color(v)=c_2 OR color(v)=c_3 ...OR color(v)=c_k) for *k* colors. If we define variables for these, like p_v^c ::= (color(v)=c), then we can state ($p_1^1 \vee p_1^2 \vee p_1^3 \dots p_1^k$) for *k* colors. We repeat this clause for each node.

b) To assert that every node have *at most* one color, we would need (in CNF form) probably $k^2/2$ more clauses looking like: ($\neg p_1^1 \vee \neg p_1^2$) $\wedge$ ($\neg p_1^1 \vee \neg p_1^3$) ... ($\neg p_1^1 \vee \neg p_1^k$) $\wedge$ ($\neg p_1^2 \vee \neg p_1^3$) $\wedge$ ($\neg p_1^2 \vee \neg p_1^4$) ... ($\neg p_1^2 \vee \neg p_1^k$) … wherein p_1^1 could only be true if all of the other color assignments were false.

c) To assert that no two adjacent vertices had the same color, we could need to assert that, for every edge connecting nodes i and j, one of them would need to *not* have a particular color: ($\neg p_i^1 \vee \neg p_j^1$) $\wedge$ ($\neg p_i^2 \vee \neg p_j^2$) ... ($\neg p_i^k \vee \neg p_j^k$)

d) To optimize the criteria, we can actually toss the criteria in section b), since, if the solver produces a solution where a node can have more than one color, *we can just pick <u>any</u>* of those colors for that node.

e) For V vertices and *k* colors, we'll need *V*k* total variables.The clauses we'll need from part a) will be *V* disjunctive clauses (each with *k* variables conjoined) and part c) will be *E*k* clauses (each with 2 variables).

Q.7

See files in 'graph-coloring' in github submission. The setup of the constraints all happens in the 'k-coloring.rkt' file. 'examples.rkt' has been modified to allow us to run the test on any of the problems simply by index number. The solver operates by using integers, instead of letters, to refer to variables, and the mention of a variable asserts it to be true. Asserting the *negative* integer asserts it false. CNF form is used, so the items in the inner lists are conjoined and those lists are disjoined. For example,

(1 2) (-2 3)

Translates as: "(V1 OR V2) AND (NOT V2 OR V3)"

To translate our clauses from Q.6 to this form, we use the conversion: p_i^c is replaced by the integer (i*k + c + 1), where i is the vertex number (starting from 0), c is the color number (starting from 0), and k is the total number of colors we can assign. We add 1 to prevent p_0^0 from evaluating to 0, which can't be made negative. As an example, in a solution with 3 nodes and 2 colors, (1 -2 3 -4 -5 6) would mean that node 0 had color 0, node 1 had color 0, and node 2 had color 1.

The k-coloring function operates by appending two CNF lists (into one, larger CNF list):

List #1 will contain the assertions discussed in Q.6.a, that every node has at least one color. We do this with a series of conjunctions, like (1 2 3) (4 5 6) (7 8 9) … which is saying that at least one of colors 0-2 are assigned to node 0 and at least one of colors 0-2 are assigned to node 1, etc.

List #2 contains the assertions from Q.6.c, that every edge has different colors at either end. Using a list of edges (in the form of (node . node) pairs), we can create a set of pairs of negations for each color. So, (-1 -4) would mean that *either* node0=color0 OR node1=color0 must be false. We do this for all colors for any edge, (-1 -4) (-2 -5) (-3 -6).

Combining these lists is enough to hand to the solver. The last step is to convert the solution (by selecting the positive values in the solution) into a list of colors.

Q.8

For any given vertex, instead of representing *each* color assignment with a separate variable, we could actually use binary encoding (with the variables as bits) to represent the vertex colors. So, (p_1^1 , $\neg p_1^2$, p_1^3) would signify vertex 1 having color 101b (or 5 in decimal). This reduces the number of variables to |V| * log(k), and a few other interesting things happen. First, ensuring that vertices only have one color is a non-problem, since the binary encoding can only refer to a single color. However, *limiting* the number of available colors to a certain number can get strange, as we'd have to get either very clever or very verbose in order to prevent bit combinations beyond a certain value *k*.

Q.9

Probably the most obvious way would be to start with the clauses which only involved the first vertex and limit the number of colors to *1*. Then, introduce the clauses for another vertex and check to see if it's still satisfiable with 1 color. If not, then increment the number of colors available, which should guarantee satisfiability with the recent additional vertex. Repeat this process until all vertices have been added.

I find this solution a little problematic, however. We cannot just assign the new color to the new vertex and continue on, since we might not get optimal coloring (i.e. we might end up using more colors than we need). So, we must completely *re-solve* our partial logic expressions each time we add a color, but I'm not sure that this is going to be any faster than just trying to solve the *entire* graph with 1 color, then 2 colors, etc, until we can satisfy.

Section 4

Q.10

Q.11