NP-COMPLETENESS STUDY QUESTIONS Solutions
Fundamental Algorithms, Fall 2004, Professor Yap

Due: Do not submit.
SOLUTION PREPARED BY Instructor and T.A.s Ariel Cohen and Vikram Sharma

INSTRUCTIONS:

- These Study Questions are in lieu of Homework 6, and focuses on NP-Completeness (Lecture XXX). This should be your priority in studying Lecture XXX. We want you to be familiar with the concept of reducibility, and to be able to do simple reductions. In particular, reducing problems to the satisfiability problem SAT.

---

1. Prove Lemma 1 (p.7), which shows that the set of well-formed Boolean formulas is in P .

   HINT: You must describe a deterministic Turing machine that accepts iff the input has the form $\#F$ for some Boolean formula F. Moreover, M must run in polynomial time. This is an exercise in programming Turing machines!

   SOLUTION:

   Let w be the input string. From the definition of a formula, we see that each application of Boolean operator $(\neg, \vee, \wedge)$ is accompanied by a pair of parenthesis. In the following algorithm, w will be continually being modified and in fact, $|w|$ is reduced in each iteration. If w is well-formed, we will finally convert it to the string $w = \mathtt{x}$, otherwise, convert it to the string $w = \mathtt{y}$. In the following, if a string has the form

   $$\mathtt{x}\{\mathtt{0},\mathtt{1}\}^* \tag{1}$$

   we call it a "variable string"

```
While $|w| > 1$ do:
1.   Suppose there are no parenthesis symbols in $w$.
         If $w$ is a "variable string" we erase all the $\mathtt{0}$'s and $\mathtt{1}$'s in $w$, leaving just the symbol $\mathtt{x}$.
         Otherwise, we replace $w$ by the string $\mathtt{y}$ and exit.
2.   Suppose there is some parenthesis symbol.
         Then we locate any innermost matching pair, say $w = w_0(w_1)w_2$ where $w_1$ has no parenthesis.
         If this is not possible, we replace the entire string by $\mathtt{y}$.
3.   We verify that $w_1$ has exactly one operator symbol ($\neg, \vee$ or $\wedge$) and has one of the 3 forms:
                 $w_1 = \neg w_3$, $w_1 = w_3 \vee w_4$, or $w_1 = w_3 \wedge w_4$,
         where $w_3, w_4$ are "variable strings". If so, we replace the substring $w$ by $w_0\mathtt{x}w_2$.
         Otherwise, we replace $w$ by the string $\mathtt{y}$ and exit.
```

   Upon termination, we accept iff $w = \mathtt{x}$. Each iteration of the while loop, we strictly reduce the length of w. Each iteration takes time $O(n^2)$ where n is the length of the original input string. Hence the overall complexity if $O(n^3)$.

   To implement the above using a STM, we use various tricks: (1) We can "mark" the substring $\ldots(w_1)\ldots$ in w, by introducing a marked version of each symbol in Σ. I.e., for each symbol $\mathtt{a}$, we introduce another symbol $\underline{\mathtt{a}}$ (2) We can "delete" a substring (w_1) in $w = w_0(w_1)w_2$ in $O(n^2)$ time. If you do not understand this $O(n^2)$ behavior, you need to work out the details yourself.

2. Exercise 4.2 (p. 10), showing that $L \in NP$ iff L is verified in polynomial time by a verification machine.

   HINT 1: As usual, an "iff" results require two demonstrations, the "if" part and the "only if" part. In the Turing machine transition

   $$(u \overset{(a,a',D)}{\longrightarrow} v) \tag{2}$$

we call (u, a) the precondition of the transition. Note that the Turing machine must satisfy this precondition (i.e., be in state u and scanning symbol a) in order to be able to execute this transition. A nondeterministic Turing machine M is said to be in **nondeterministic normal form** if it has this property: *for each precodition (u, a) there are exactly two transitions with this precondition.* It is not hard to convert any nonterministic Turing machine to this normal form; furthermore, the running time of the normalized Turing machine on any input is $O(T(n))$ if the original running time is $T(n)$.

By some convention, we can specify these two transitions as the "first" and the "second" transition for (u, a). Hence, a computation of M on any input is completely deterministic once we also know the sequence of choices (i.e., either the first or second transition is taken) made in every step!

HINT 2: A simple Turing machine M can simulate a vertification machine V as follows. Note that V has two tapes (and two independent tape heads). We let M's tape be organized into 3 "tracks". Two of the tracks correspond to the 2 tapes of V, and the third track is for book keeping. Since V has 2 tape heads and M has only one head, in general, M must take $\Theta(T(n))$ steps for each step of V. Hence the simulation will take $O(T(n)^2)$ time.

SOLUTION:

Suppose $L \in NP$. Let L be accepted by a nondeterministic STM M. We may assume M is in nondeterministic normal form. We can convert M into a verification machine V for L as follows: V is obtained from M by giving it another input tape to store a "choice string" $c \in \{1, 2\}^*$, representing the nondeterministic choices in every step of the computation. Then $x \in L$ iff there is a choice string c such that (x, c) will cause V to accept.

Suppose L is verfied by a verification machine V. We construct a nondeterministic machine M as follows: on input x, M will first "guess" a choice string c on another track of its input tape. Then it simulates the behavior of V on the pair (x, c)

3. The proof of Lemma 5 lists three propositions (1)-(3) that must be constructed. We only described (1). Please do the same for (2) and (3).

   HINT: the formula for (3) must depend on the graph G.

   SOLUTION:

   (2) For each i, there is a unique j such that x_{ij} is true:

$$\left(\bigvee_{j=1}^{n} x_{ij}\right) \wedge \left(\bigwedge_{1\le j<j'\le n} (\overline{x}_{ij} \vee \overline{x}_{ij'})\right)$$

   (3) For $i \neq i'$, if x_{ij} and $x_{i',j+1}$ are true then (i, i') is an edge of G. We construct the following formula based on the graph G:

$$\bigwedge_{(i,i')\notin E} \bigwedge_{j=1}^{n} \neg(x_{ij} \wedge x_{i',j+1})$$

   where we assume that $j + 1 = 1$ when $j = n$.

4. Let the addition predicate $A(a, b, c)$ where a, b, c represent numbers in binary notation, and $A(a, b, c)$ is true iff $a + b = c$. Show how to construct in polynomial-time a Boolean formula $F(a, b, c)$ that is true iff $A(a, b, c)$ is true.

   HINT: Assume a and b are m-bit binary numbers. So let $a = (a_1, \ldots, a_m)_2$, $b = (b_1, \ldots, b_m)_2$ and $c = (c_0, c_1, \ldots, c_m)_2$. where a_m, b_m, c_m are the least significant bit of the respective binary notations. The formula $F(a, b, c)$ involves these a_i's, b_j's and c_k's. We can introduce the sequence of carries $d = (d_0, d_1, \ldots, d_m)$ where $d_m = 0$ and $d_0 = c_0$ and d_i's is the carry bit into column i. Thus we have $a + b = c$ iff for all $i = 1, \ldots, m$,

$$a_i + b_i = c_i + 2d_{i-1} + d_i.$$

This equation can be expressed as a boolean formula. E.g., $a = (1, 0, 0, 1, 1)$, $b = (1, 1, 0, 0, 1)$, $c = (1, 0, 1, 1, 0, 0)$ and $d = (1, 0, 0, 1, 1, 0)$. This gives the formula

$$G(a, b, c, d) = \bigwedge_{i=1}^{m} \text{“}(a_i + b_i = c_i + 2d_{i-1} + d_i)''$$

Since the d_i's are not given, you need to replace them by 0's and 1's, and take a disjunction over different copies of $G(a, b, c, d)$:

$$F(a, b, c) = \bigvee_{d} G(a, b, c, d).$$

Unfortunately, this leads to an exponential size formula as there are 2^m possible choices for d. To get around this, use a recursive formulation where we may assume m is a power of 2.

SOLUTION:

We split up each m-bit string into two $m/2$-bit strings. Let $a = (a_L, a_R)$ where a_L, a_R are $(m/2)$-bit numbers. Similarly for b. In the case of c, we assume c_L has $(m/2) + 1$ bits while c_R has $(m/2)$ bits. Then we can reformulate

$$F(a, b, c) = \bigvee_{d=0}^{1} F_d(a_L, b_L, c_L) \wedge F(a_R, b_R, (d, c_R))$$

where $F_0(a, b, c)$ is just the same as $F(a, b, c)$ but $F_1(a, b, c)$ means that $a + b + 1 = c$. It is clear that the formula is correct. But how do we write $F_1(a, b, c)$? We will use $m + 1$ copies of $F(a^{[i]}, b, c)$ where $a^{[i]}$ is a suitably modified version of a. The basic idea is that $a^{[i]}$ corresponds to $a + 1$ provided $a_i = 0$ and $a_{i+1} = a_{i+1} = \cdots = a_m = 1$, and there is also the special case where all the a_i's are 1:

$$F_1(a, b, c) = \left(\bigvee_{i=1}^{m} \left(F(a^{[i]}, b, c) \wedge \overline{a}_i \wedge (\bigwedge_{j=i+1}^{m} a_j) \right) \right) \vee \left(c_0 \wedge (\bigwedge_{j=1}^{m} b_j = c_j) \wedge (\bigwedge_{j=1}^{m} a_j) \right)$$

and $a^{[i]} = (a_1, a_2, \ldots, a_{i-1}, 1, \underbrace{0, \ldots, 0}_{m-i})$.

Thus $F_1(a, b, c)$ uses m copies of $F(a, b, c)$.

We must estimate the size $T(m)$ for the size of our recursive formula $F(a, b, c)$ where a, b, c are m-bits. Initially, assume each variable has unit size. Hence

$$T(m) = mT(m/2) + m.$$

This has solution $T(m) = O(m^{\lg m}$, which is a great improvement from before, but still not polynomial size. To further reduce this, we need to reuse our formula in F_1. We leave this as an exercise.