Undergraduate Complexity Theory **CMU 15-455, Spring 2017**

HOMEWORK 5

Due: 5:00pm, Thursday February 23

NEWISH FEATURE: As before, if your homework is typeset (as opposed to handwritten), you will receive 1 bonus point.

---

0. **(UNSAT in NP?)** (0 points, do not turn in.) Let

$$\text{UNSAT} = \{\langle\phi\rangle : \phi \text{ is an } \textit{unsatisfiable} \text{ Boolean formula}\}.$$

Can you show UNSAT $\leq_m^P$ SAT?

(Probably you can't, because in fact no one knows whether or not this is possible. But almost everyone believes it's *impossible*. Unfortunately, I can't ask you to show it's impossible because... well, nobody knows how to *prove* it's impossible. But you should think about why it's challenging/unlikely, and what goes wrong with some naive attempts to do it.)

1. **(Reduction Timing.)** (10 points.) Suppose we show that $A \leq_m^P B$, using a reduction algorithm R with the following two properties:

   - On inputs x of length n, algorithm R runs in $O(n^r)$ time.
   - On inputs x of length n, algorithm R outputs a string of length $O(n^\ell)$.

   Furthermore, assume we find an algorithm S that, on inputs x of length n, decides whether or not $x \in B$ in time $O(n^b)$. (Here $r, \ell, b \geq 1$ are all constants.)

   Now since $A \leq_m^P B$ and $B \in \mathsf{P}$, we know we may conclude $A \in \mathsf{P}$. But suppose we care in more detail about running times. What is the least constant a for which you can conclude that there's an algorithm, running in time $O(n^a)$ on inputs x of length n, that decides whether or not $x \in A$? Express your answer a as a function r, ℓ, b.

2. **(NTMs.)** Sipser defines TMs and TM computation in Section 3.1. Recall that in our class we depart from his definition in two slight ways. First, we use a two-way infinite tape rather than a one-way infinite tape. Second, I think Sipser should have directly defined the transition function δ as $\delta : Q' \times \Gamma \to Q \times \Gamma \times \{L, R\}$, where $Q' = Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}$. (Sipser first defines the domain of δ as $Q \times \Gamma$, but such a definition implies kind of weirdly that you have to specify how the TM transitions out of halting states. Note that later, on page 169, he comes to our preferred definition with Q'.)

   Recall also in Lecture 3, we considered a TM variant in which, in addition to allowing the head to move Left or Right, the head could also Stay. The resulting form of the transition function was $\delta : Q' \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$. Recall we showed that you could simulate this new Staying-allowed-TM with a normal Left-or-Right-only-TM with just a constant-factor slowdown.

   Sipser further defines *nondeterministic* Turing Machines (NTMs) and NTM computation in Section 3.2. Let us call this definition "Sipser-NTMs" but make the above three small changes: two-way infinite tape, Q' instead of Q, and Staying allowed. Thus in Sipser-NTMs, the transition function has the following form:

$$\delta : Q' \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R, S\}),$$

where $\mathcal{P}(\cdot)$ denotes power set. (Recall the two-way infinite tape aspect shows up not in the definition of the machines but in the definition of how machines compute, involving "configurations".) Finally, Sipser is not exactly clear on what happens if $\delta(q, \gamma) = \emptyset$; for explicitness, let's say the NTM halts and rejects if this arises.

In this problem, we will give a somewhat *different* definition of NTMs, call it "binary-NTMs". Basically, your task will be to show that Sipser-NTMs and binary-NTMs are essentially the same.

First, recall from Lecture 2 the definition of computation for a *deterministic* Turing Machine M. Given its transition function $\delta : Q' \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$, there is a natural associated function $\text{NextConfig}_\delta(C)$ that, given a nonhalting configuration C, outputs the configuration C' resulting from doing one step according to transition function δ. Then, to define the computation of $M(x)$, we form $C_0, C_1, C_2, \ldots$, where C_0 is the "initial configuration" given x, and where $C_{t+1} = \text{NextConfig}_\delta(C_t)$. We keep doing this till we either get a halting configuration C_t, or forever (if a halting configuration is never reached). In the former case, we define the running time to be t, and we say that "$M(x)$ accepts" or "$M(x)$ rejects" depending on whether the final configuration has the accepting state or the rejecting state.

Let us now define a "binary-NTM" as follows: It is like a deterministic TM (with two-infinite tape and Staying put), except you specify *two* transition functions, called δ_0 and δ_1:

$$\delta_0, \delta_1 : Q' \times \Gamma \to Q \times \Gamma \times \{L, R, S\}.$$

Then, we'll say that in a binary-NTM N, the computation $N(x)$ explores *all possible* computation traces $C_0, C_1, C_2, \ldots$ where $C_{t+1} = \text{NextConfig}_{\delta_{i_t}}(C_t)$ for bits $i_t \in \{0, 1\}$. We say that $N(x)$ "halts" if all the possible traces hit halting configurations (else it "loops"). In the former case, we say that "$N(x)$ accepts" if *at least one* possible computation trace halts with the accepting state, and "$N(x)$ rejects" if *all* possible computation traces halt with the rejecting state. Either way, we define the running time of $N(x)$ to be the maximum length of all the possible configuration traces.

Please show two things:[1] First, every binary-NTM can be viewed as a special case of a Sipser-NTM. Second, every (decider) Sipser-NTM can be equivalently simulated by a (decider) binary-NTM with at most constant-factor slowdown in running time.

3. **(Implicit Coloring.)** (10 points.) Recall our definition of the complexity class NEXP:[2]

$$\mathsf{NEXP} = \{L : \text{there exists a nondeterministic algorithm } N \text{ deciding } L \text{ with running time } O(2^{n^k}) \text{ for some } k \in \mathbb{N}\}.$$

[1] Honestly, a lot of the point of this problem is to just make you understand all the definitions carefully. For this reason, I was pretty pedantic about writing out all the definitions. On the other hand, you don't have to be insanely pedantic in writing your solution. Basically, on page 49 of Sipser you see some pictures involving trees. For binary-NTMs, the idea is that the trees have a certain form... Indeed, we basically discussed the idea for how to solve the second part of this problem in class. But I would like you to think through the definitions a bit carefully, and spell things out a bit.

[2] As with the classes P, NP, and EXP, this class is "robust" with respect to the model of computation. That is, since TMs, multi-tape TMs, Python, "pseudocode", etc. can all simulate each other with polynomial slowdown (and the same is true of the nondeterministic versions), the definition of NEXP does not change depending on whether "nondeterministic algorithm" means "nondeterministic TM", "nondeterministic multi-tape TM", "nondeterministic Python", etc. Therefore, in this problem you can and should use "nondeterministic pseudocode" (pseudocode with "goto-both" instructions) when describing NEXP algorithms.

The task in this problem is to prove that IMPLICIT-4COL $\in$ NEXP.

In the rest of the problem, we define the language IMPLICIT-4COL. At a high-level, the decision problem IMPLICIT-4COL involves being given a Boolean circuit C which *implicitly defines* a graph G_C with 2^n vertices; the task is to decide if G_C is validly 4-colorable.

More precisely: Let C be a Boolean circuit and suppose (slightly contrary to normal conventions) that we write $2n$ for the number of input gates it has. (So we assume it has an even number of input gates.) Then we define an associated undirected graph $G_C = (V, E)$ as follows. First, $V = \{0, 1, 2, \ldots, 2^n - 1\}$, so G_C has 2^n vertices. Next, for distinct $i, j \in V$, the pair $\{i, j\} \in E$ if and only if either $C(i, j) = 1$ or $C(j, i) = 1$ (or both). Here $C(i, j)$ means that i and j are each written as base-2 numbers with exactly n bits (so "leading 0's" are used if necessary), and the resulting $2n$ bits are fed in as inputs to C. Roughly speaking, C computes the adjacency matrix of G_C, but with some small twists: (i) It may be the case that $C(i, j) \neq C(j, i)$; so as we said, we define $\{i, j\}$ to be present in E if and only if *either* $C(i, j) = 1$ or $C(j, i) = 1$. (ii) Whatever $C(i, i)$ is, it doesn't matter; by definition we say that G_C does not have self-loops. Finally,

$$\text{IMPLICIT-4COL} = \{\langle C \rangle : C \text{ is a Boolean circuit with an even number of inputs such that } G_C \text{ has a valid 4-coloring}\}.$$

4. **(Reducing P to 3COL.)** (10 points.) Show that for every $L \in \mathsf{P}$ it holds that $L \leq_m^P$ 3COL.

   (Note: although we will talk about the Cook–Levin Theorem on Tuesday, you may not use it, and do not need it, for this problem. Give a direct argument.)