CS 182, Complexity Theory
Fall 2010
Homework 4
Due Wednesday, September 29

1. **[30 Points] PSPACE = NPSPACE!**
Before embarking on this problem, make sure that you understand the proof we showed in class that QBF is PSPACE-complete.

Recall that we defined PSPACE to be the set of all languages L such that there exists some polynomial $p(n)$ and some **deterministic** TM M such that on input w, M uses at most the first $p(|w|)$ tape cells to determine whether w is in L.

Similarly, we can define NPSPACE to be the set of all languages L such that there exists some polynomial $p(n)$ and some **nondeterministic** TM M such that on input w, if $w \in L$ then there exists some accepting computation path, if $w \notin L$ then there exists no accepting computation path, and *every* computation path uses at most the first $p(|w|)$ tape cells. While it is not known whether P = NP, it is known that PSPACE = NPSPACE. In this problem we'll prove this! Note that NPSPACE is defined in terms of nondeterminism and *not* defined in terms of verifying certificates.

   (a) Professor I. Lai proposes the following approach: "Look, if L is accepted by a nondeterministic TM M in polynomial space, I will build a deterministic TM M' that simulates M, and thus accepts the same language, and still only uses polynomial space! How? Easy! Nondeterministic TM M has at most b nondeterministic choices at each step. M' will enumerate sequences in base b on a work tape and then simulate M on input w using the current sequence to dictate the choices that M would make. If, for a given base b sequence the simulation reveals that M accepts w, then M' will accept w. Otherwise, we will write down the next base b sequence and try again. If none of these sequences work, well then, M can't accept w so M' also rejects w. In order to save space, M' will overwrite the new base b sequence over the old base b sequence. It's so easy that I want to eat ten cans of SPAM to celebrate!" Explain why Professor Lai is celebrating a bit too soon. That is, explain what's wrong with this so-called proof.

   (b) Now we will fix things to make the proof work correctly! The idea is to use techniques similar to those employed in Stockmeyer's Theorem to show that QBF is PSPACE-complete! Notice that since nondeterministic

TM M never uses more than polynomial space, $p(n)$, it cannot enter more than $c^{p(n)}$ different configurations for some constant c. Moreover, any one of these configurations can be recorded in polynomial space.

Using this observation, argue carefully that a deterministic polynomial space TM M' can be constructed to simulate TM M. Make sure to show that no more than polynomial space is really being used in your simulation. Pretty nifty!!

2. **[20 Points] Generalized Lunar Lockout is in PSPACE** In class, we described the Generalized Lunar Lockout game and showed that it is PSPACE-hard. Your job is to prove that the problem is in PSPACE, thus establishing that the problem is PSPACE-complete. Recall that the game is played on a $m \times m$ board. The initial configuration is some set of objects placed on the board. An object may be a stationary robot (a "rock"), a movable robot, or the special movable target robot. One of the cells is designated as the target cell. The question is whether there exists a sequence of moves that bring the special target robot to the target cell. Notice that the number of moves in such a "winning" sequence could potentially be very long!

3. **[20 Points] Regular expressions and PSPACE.**

   Many problems related to regular expressions are known to be in PSPACE but are not known to be in any "lower" complexity class. For example, consider the problem of determining whether a given regular expression over the alphabet $\Sigma = \{0, 1\}$ is equivalent to Σ^*. For example, $0^*1^* + (0 + 1)1^*$ is not the same same as Σ^* but $1^*((\epsilon + 1)(\epsilon + 0))^*$ is equivalent to Σ^*.

   So, we are considering the following problem: Given an encoding of a regular expression, is that regular expression equivalent to Σ^*? We'll call this problem REGEQ (REGular expression EQuivalence).

   (a) First, very briefly sketch the arguments for each of the following claims (CS 81 review).

      i. Every regular expression of length n has a corresponding NFA with a number of states that is linear in n. (Give just a few sentences that indicate why this is so.)

      ii. Every NFA with n states has a corresponding DFA with at most 2^n states. (Give a two sentence explanation.)

      iii. If a DFA with k states accepts some string w of length k or more then it must also accept a shorter string. (Very short explanation here!)

(b) Next, consider the problem of determining whether the language of a given DFA is Σ^*. Describe a finite-time algorithm (it need not be efficient in any sense) that takes as input a DFA and determines whether that DFA accepts Σ^*. Explain briefly why your algorithm is correct.

(c) Now, prove that REGEQ is in PSPACE. Be careful to show that your solution works correctly and really uses only polynomial space!

(d) **OPTIONAL 10 POINT BONUS PROBLEM!** *This bonus problem is interesting and not terribly tricky, I just made it optional to help moderate the workload on this assignment.* Consider the more general problem REGPAIR that takes as input the encodings of two regular expressions, R_1 and R_2, and determines whether or not these two regular expressions represent the same language. Prove that REGPAIR is in PSPACE. (Here, it may be useful to remind yourself that the regular languages are closed under union, intersection, and complement and remind yourself of how we prove this.)

4. **[30 Points] Finishing Up the NL = co-NL Proof!** Your task is to write out *the entire proof* that NL = co-NL. Our approach was to show that co-PATH is in NL.

   In class we started the proof that co-PATH is in NL. The funny thing that we assumed in that proof is that the non-deterministic log space TM for co-PATH was given not just $< G, s, t >$, but also r, the number of vertices reachable from s in G. In order to complete the proof, you'll need to show that r can be computed by the non-deterministic log space TM by itself before launching into the step that we described in class. In other words, the non-deterministic log space TM for co-PATH will construct a tree that has two "parts". The "upper part" will compute r and the lower part will use r to determine that there is no path from s to t.

   (a) Professor I. Lai of P.I.T. has proposed the following approach for computing r: "It's easy!" explains Professor Lai nonchalantly. "You simply have your non-deterministic machine first use non-determinism to guess every possible value of r and place this guess in a counter. Writing r down takes log space. Now, each guess of the counter's value appears in a different subtree of our non-deterministic tree. In each subtree, we now verify that guess as follows: Use non-determinism again to sequentially guess the vertices that we believe to be reachable from s. For each such guessed vertex we first verify that we can reach that vertex (in the same

way that we checked if s can reach t in our proof that PATH is in NL) and, if the answer is "yes", we decrement the counter. If we get to a point that the counter is zero, we conclude that we guessed the correct value of r and we now have r to use in the rest of the construction!" Now "r-gue" that Professor Lai is wrong!

(b) Here's an approach that does work. Let S_i denote the set of vertices that are reachable from s and at distance i or less from s. If there are ℓ vertices in the graph, we'd like to find S_ℓ since this will be the set of all vertices reachable from s. The size of S_ℓ is the value r that we need. Unfortunately, even one such set may be too large for us to store (in its entirety) on tape since we are log-space bounded! So, instead we'll compute the *sizes* of these sets where $r_i = |S_i|$ and we're seeking $r = r_\ell$.

We know that $r_0 = 1$ since there is exactly one vertex reachable from s using 0 or fewer edges (it's s itself!). Now, we'll try to iteratively compute r_1 and then r_2, etc. Each r_i will be computed based on the previous r_{i-1} value that we've computed. How do we do that? Notice that the vertices in set S_i are those that are reachable from s in i or fewer hops. Thus, they are all of the vertices in S_{i-1} and all of the vertices reachable in one hop from S_{i-1}. Another easier and more convenient way of characterizing this, is that a vertex is in S_i if it is either s or is reachable from some vertex in S_{i-1} in one hop (notice that this includes the vertices that are at distance i from s as well as those that are at distance less than i from s).

So now what? Again, we're assuming that we've computed r_{i-1} in the previous iteration. We'll now use non-determinism to guess and verify the vertices in S_{i-1} (but without ever writing all of them down at once since that takes too much space) and use that to compute r_i.

In your write-up first describe the process by which r_i is computed from r_{i-1}, then put this together to explain how $r = r_k$ is computed, and then briefly explain how this fits together with what we showed in class to conclude that co-PATH is in NL.