CS500, Theory of Computation Homework #5 Solutions

1. Show that the bracket language $\{\epsilon, (), (()), ()(), \ldots\}$ is in L.

   Answer. Read the input from left to right. Maintain a counter on the worktape with initial value zero, and increment or decrement it when reading '(' or ')' respectively. Reject if the counter ever becomes negative, and accept if the counter is zero at the end of the input. Since the counter can never exceed the input length n, it is a $(\log_2 n)$-bit number, so this only requires a worktape with $O(\log n)$ bits.

   More generally, any language which can be recognized by a one-counter machine — or a machine with any constant number of counters — is in L. □

2. Show that the language of palindromes over the alphabet $\{a, b\}$ is in L.

   Answer. Our goal is to check that each symbol matches the one in the position opposite to it, i.e. that $w_i = w_{n+1-i}$ for all i. Now, a log-space TM can keep track of its current position by incrementing or decrementing a counter as it moves left or right. It can also carry out 'for' and 'while' loops as long as the counters have values polynomial in n, since then the counters are $O(\log n)$-bit numbers and can fit in $O(\log n)$ space. Thus the pseudocode is

```
i = 1
do until i exceeds the length of the input {
  move left until you hit the left end of the input
  for j = 1 to i-1: move right
  read w1    // w1 = w(i)
  move right until you hit the right end of the input
  for j = 1 to i-1: move left
  read w2    // w2 = w(n+1-i)
  if w1 != w2 reject
  i++
}
accept
```

   (Note that this shows that L is more powerful than 1-counter or constant-counter machines that read their input from left to right, since by the equivalence class argument these cannot recognize palindromes.) □

3. Let L_2 be the two-type bracket language $\{\epsilon, (), [], [()], ([]), ()[], \ldots\}$. Note that the brackets must be properly nested, so words like $([)]$ are not allowed. Show that $L_2 \in \mathrm{L}$. This is tricky! Hint: consider an inductive strategy that checks a property of every substring.

   Answer. Let us call say that each symbol has a *type*, either round or square, and say that each symbol is a left or right bracket regardless of type. Each left bracket has a right bracket which is its "partner", and our goal is to check that every left bracket's partner is of the same type. To find its partner we use a counter as in question #1 above. First, we check that the word is in the bracket language of question #1 if we ignore round vs. square, so that every left bracket has a right partner. Then, to check that partners match, we use the following pseudocode:

```
i = 1
do until i exceeds the length of the input {
  move i-1 steps from the left end of the input
  read the input symbol a    // a = w(i)
  if a is a left bracket {
    c = 1
    do until c = 0 {    // find w(i)'s partner
      move right and read the next input symbol b
      if b is a left bracket, increment c
      if b is a right bracket, decrement c
    }
    if a and b are of different types, reject
  }
}
accept
```

□

4. A directed graph G is *strongly connected* if for every pair of vertices u, v there is a directed path from u to v. Show that, given G as input, STRONG CONNECTEDNESS is NL-complete. Hint: show how to modify the graph to reduce REACHABILITY (where two particular vertices are given as input) to STRONG CONNECTEDNESS.

   Answer. First we show that STRONG CONNECTEDNESS is in NL. If G has n vertices, we do a pair of nested 'for' loops with u and v ranging from 1 to n; for each pair u, v we guess a path from u to v. We only need to keep track of u, v, and the current vertex in our path, and all of these are $O(\log n)$-bit numbers.

   To reduce REACHABILITY to STRONG CONNECTEDNESS, we need to show how, given a directed graph G and two vertices s and t, we can generate a graph G' such that G' is strongly connected if and only if there is a path in G from s to t. We do this by letting G' be G with some additional edges: specifically, for all vertices v we add edges $v \to s$ and $t \to v$.

Now, suppose that G has a path from s to t. Then G' is strongly connected, since we can get from any vertex u to any other vertex v by going from u to s, along this path to t, and then from t to v.

Conversely, suppose that G does not have a path from s to t. Adding these edges to G' cannot create such a path; the $v \to s$ edges can only help us return to s, and the $t \to v$ edges are useless if we can't reach t in the first place. Thus G' does not have a path from s to t, so it is not strongly connected.

Finally, although we have not been very precise in class with regard to what type of reductions we use for NL-completeness, this reduction is clearly very easy to carry out; in particular, we can do it in L. $\square$

5. Show that Graph 2-Colorability (also known as Bipartiteness) is in NL.

   Answer. A graph is *not* 2-colorable if it has an odd loop. An NL machine can guess an initial vertex v, guess a series of vertices w, check that they are connected, flip a bit at each step to keep track of parity, and accept if $w = v$ after an odd number of steps.

   Since the *complement* of Graph 2-Colorability is in NL, Graph 2-Colorability is in co-NL; but co-NL = NL by the Immerman-Szelepczenyi theorem. $\square$

6. I made a comment in class that NL is the set of problems for which a certificate exists that can be verified by a Turing machine with $O(\log n)$ workspace, with the additional restriction that we only read this certificate left-to-right (i.e., the certificate is given on an additional read-only tape on which the TM can only move to the right). Formalize and justify this comment.

   Answer. Formally, we are saying that a language L is in NL if there is a *deterministic* Turing machine M with a read-only input tape, a left-to-right read-only certificate tape, and an $O(\log n)$-space worktape, such that $x \in L$ if and only if there exists a certificate y such that M will accept the pair $\langle x, y \rangle$. This is analogous to the two definitions of NP: in one, a non-deterministic machine accepts an input x if it makes the right guesses; in the other, a deterministic machine checks x and a certificate y, and $x \in L$ if and only if a certificate exists that makes it accept.

   First let's show that such a language is in NL. A non-deterministic machine M' can simulate M by guessing a symbol of the certificate each time M moves forward on its certificate tape. M' doesn't have to store these guesses (which is good, since the certificate can be of polynomial size); since M can only move left-to-right on its certificate tape, there is no way to go back and check them later.

   Now we show that any NL machine M' can be described with this kind of certificate. We have M simulate M', where the certificate tells M which choices M' makes on its computation path. Clearly, the certificate only needs to be read from left to right. Then an accepting computation path exists for M' if and only if there is a certificate that leads M to accept. $\square$

7. In the previous problem, what happens if we remove the restriction that the certificate is read left-to-right? What complexity class do we get? (What's the hardest problem you can think of where the certificate can be verified with $O(\log n)$ workspace, if we can go back and look at the certificate many times?)

   Answer. NP! It turns out that read-only access to the input, and $O(\log n)$ workspace, is enough to check solutions to NP-complete problems. The only difference from NL is that we might have to visit the same place in the certificate many times.

   For example, let's prove that 3-SAT is in this class. The read-only input tape holds the formula, and the certificate holds a satisfying assignment. The TM then reads the formula from left to right; for each clause $(x_i \vee \overline{x}_j \vee x_k)$, say, it looks up the ith, jth and kth bits of the certificate, and rejects if none of them agree with the clause.

   To do this lookup, we need just 6 counters: one for the clause we're currently checking, three for i, j, and k, and one each for our current position on the input tape and the certificate tape. All of these are $O(\log n)$-bit numbers, so we only need a workspace of $O(\log n)$ bits.

   Since this class includes an NP-complete problem (you can find a similar solution for, say, Hamiltonian Path) it includes all of NP. But, this machine model is clearly a restriction of the one for NP, since if it can accept, it can accept in polynomial time. To see this, note that the total number of configurations of the machine is the number of strings we can write in the workspace, times the number of positions the heads can take on the input and work tapes. This gives

$$2^{O(\log n)} \times \text{poly}(n) \times O(\log n) = \text{poly}(n) \ .$$

   If there exists an accepting path, there exists one of this size or less. Thus any language in this class is in NP, and the two are identical. □

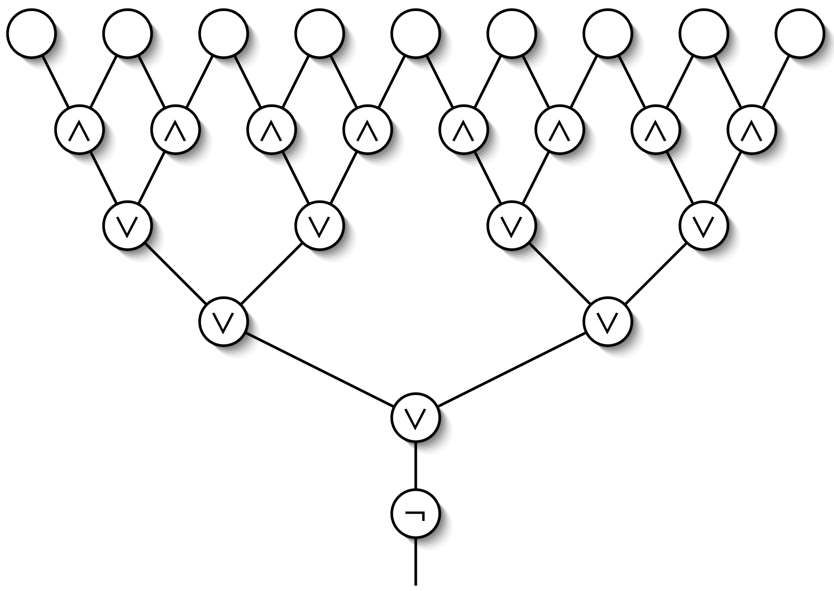
8. Remember our old friend

$$L = \{w \in \{0,1\}^* \mid w \text{ does not contain } 11\} \ .$$

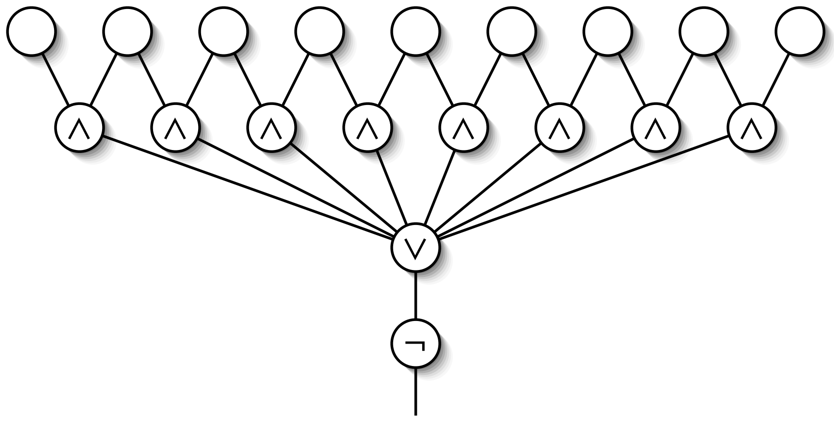
   Give an NC^1 family of circuits, consisting of binary AND and OR gates, for membership in L. If we are allowed to use AND or OR gates of arbitrary *fan-in* (i.e., which take arbitrary numbers of inputs) instead of binary gates, how shallow can the circuit be?

   Answer. See next page...

For an input of length n, we first have a layer of $n-1$ AND gates, which check if an adjacent pair of inputs are both 1s. We then need to take the OR of all of these, and finally add a NOT gate so that the input is accepted if it is in L. With binary gates we do this with $\log_2 n$ layers of OR gates like this:



This has width n and depth $O(\log n)$, so it is in NC^1. With arbitrary fan-in, we can do all of this with one OR gate, like this:



Circuits of this kind lie in a class called AC^0, which is known to be a proper subclass of NC^1. Do you see a proof that all regular languages are in NC^1? Not all are in AC^0...
□