Introduction to the Theory of Computation
Homework #6 Solutions

To avoid confusion with $L$, which often means a language, I will refer to $\text{SPACE}(\log n)$ and $\text{NSPACE}(\log n)$ as LOGSPACE and NLOGSPACE respectively instead of $L$ and $NL$.

1. (Problem 8.11) Show that the bracket language $L = \{\epsilon, ((), ()(), \ldots)\}$ is in LOGSPACE.

Answer: We will use a single counter to keep track of the depth of a PDA which reads the word from left to right. We increment and decrement the counter on reading a ‘(’ or a ‘)’ respectively. Start with the counter at zero, accept if it ends at zero, and reject if it ever becomes negative. Since the maximum value this counter will ever take is $n$, we can represent it with $\log n$ bits of a Turing machine’s workspace, so $L$ is in LOGSPACE.

2. Show that the language of palindromes over a given finite alphabet is in LOGSPACE.

Answer: We use several counters, each with maximum value $n$. First scan the input to learn its length $n$. Then we just need 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 $i$. It is easy to see how to do this with a TM moving left and right; note that by incrementing or decrementing a counter as it moves left or right, a LOGSPACE TM can keep track of its current position $x$. It can also carry out ‘for’ loops where $i$ ranges from 1 to $n$. Thus the TM needs three counters for $n$, $i$, and $x$, and all of these fit in LOGSPACE.

3. (Problem 8.12) Let $L$ be the two-type bracket language $\{\epsilon, (), [], ([(), ()], \ldots)\}$. Note that the brackets must be properly nested, so words like $([], )$ are not allowed. Show that $L$ is in LOGSPACE. Hint: this is tricky. Consider an inductive strategy that checks a property of every substring.

Answer: Recall that this language is recognized by a PDA with two stack symbols, say $r$ (round) and $s$ (square). If we read a left bracket of either type we push the appropriate symbol, and if we read a right bracket of either type we pop the top symbol and make sure it matches that bracket’s type.

To recognize this in LOGSPACE we can do the following: first scan left to right, using a counter to keep track of the stack depth, accepting if it starts and ends at zero, and rejecting if it ever becomes negative (as in question 1 above).

Then, using a pair of nested ‘for’ loops, for each $1 \leq i < j \leq n$ scan the subword $w_i \cdots w_j$. For each of these subwords, use a counter to check if the total change in the depth is zero and never becomes negative in the middle of the subword; call such a subword balanced. Not all subwords have to be balanced, but if $w_i \cdots w_j$ is balanced, then its leftmost and rightmost symbols $w_i$ and $w_j$ are partners, and should be left and right brackets of the same type.
Thus $w \in L$ if and only if the leftmost and rightmost symbols of every balanced subword form a matching pair of brackets. The ‘for’ loops and checking the depth can all be done with a fixed number of integer counters whose maximum value is $n$, so $L$ is in LOGSPACE.

4. (Problem 8.19) A directed graph is strongly connected if there is a directed path between every pair of vertices. Show that telling whether a directed graph is strongly connected is NLOGSPACE-complete.

Answer: First we show that Strong Connectedness is in NLOGSPACE. Again using counters, we do a pair of nested ‘for’ loops with $i$ and $j$ ranging from 1 to the number of vertices $n$; for each pair we solve Reachability($G, i, j$) by guessing a path from $i$ to $j$, just as we did when we showed that Reachability is in NLOGSPACE.

To show that Strong Connectedness is NLOGSPACE-complete, we reduce Reachability to it as follows. Given a directed graph $G$ and two vertices $s$ and $t$, Reachability is the question of whether a path exists from $s$ to $t$. For every vertex $u$, we add an edge leading from $u \to s$, and another from $t \to u$. It is clear that if the original graph had a path from $s$ to $t$, the new graph is strongly connected, since we can get from any $u$ to any $v$ by the edge $u \to s$, along the path $s \to t$, and finally the edge $t \to v$. The converse is also true, since even with these new edges there is no path from $s$ to $t$ if there wasn’t one before. Since Reachability is NLOGSPACE-complete this shows that Strong Connectedness is as well.

5. (Problem 8.20) Show that Graph 2-Colorability is in NLOGSPACE.

Answer: A graph is not 2-colorable if there is an odd loop. We can guess such a loop in NLOGSPACE just as we guess a path for Reachability. Thus Graph 2-Colorability is in co-LOGSPACE; but co-LOGSPACE = LOGSPACE by the Immerman-Szelepczenyi theorem.

6. Recall that Geography is played on a directed graph. The two players take turns moving along an edge of the graph, we cannot visit any vertex that we have already visited, and whoever gets stuck without a move loses. Geography then asks, given a graph and an initial vertex, whether the first or second player has a winning strategy. We proved in class that Geography is PSPACE-complete.

Suppose we modify Geography so that there is no rule against visiting the same vertex multiple times. Note that we can now have draws, in which the two players visit some set of vertices an infinite number of times. Show that in this version of Geography, telling whether the first player will win, lose or draw under optimal play is in P.

Answer: Consider the following local update procedure. Start with all vertices unlabeled, except for vertices with out-degree zero, which we label Losing. Now do the following until we reach a fixed point: let $v$ be an unlabeled vertex. If $v$ has an edge pointing to a Losing vertex, label $v$ Winning, and if all of $v$’s outgoing edges point to Winning vertices, label it Losing. It is easy to see that this correctly labels the vertices: a Winning vertex is one from which you can move your opponent to a Losing vertex, and a Losing vertex is one from which any move puts your opponent on a Winning vertex. The unlabeled vertices are then the Draw positions, from which both players have an incentive to keep moving on an infinite path. Since there are $n$ vertices, we will reach a fixed point after at most $n$ iterations of this procedure, so this runs in polynomial time.