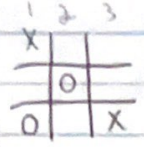
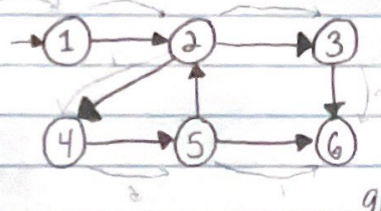
HW3
Theory of Computation

Brian DeFlaminio

1) 

Optimizing your moves to block what the opponent is trying to do while furthering your own victory chance is important. The best move from the current position is X on A3 to block the other player and secure a victory as no matter where O is placed they can no longer win. As far as a general winning strategy, trying to get the board to look the way it currently is would be a valid strategy. At minimum you want 2 of your placements of X or O to be diagonal so you can maximize winning potential.

2) 

Player 1 always choses 1, player 2 goes to 2, player 1 has a choice now, but if they want to win they must choose 4 from 2. Doing so guarantees that player 2 chooses 5 because no other options are available and then player 1 secures the victory with 6.

3) $A_{DFA} \in L$ To prove this we can construct a TM M to decide A_{DFA} in log space.

$$A_{DFA} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts } w\}$$

and M = "on input $\langle B, w\rangle$ where B is a DFA and w is a string:
1) Simulate B on input w by keeping track of its head location.
2) If it ends in an accept state, accept.
3) If it ends in a reject state, reject.

This decides A_{DFA} in $O(\log(n))$ space by trading time for space because M only computes individual symbols at a time before restarting. (credit to theorem 8.23 in Sipser)

4) A. If $A \leq_L B$ and $B \in L$ then $A \in L$. We build TM M such that for every positive integer x the inverse is computed. The "inverse" is always going to add with the regular computed value to 1 meaning that the computation $A_2 = \{(x, y) \mid R^+(x) = y\}$ is equivalent to $R^+(x) = x + x^R$ since there will be no leading 0s in x and the rest are cancelled out by binary addition $0 + (0)^R = 0 + 1 = 1$

B) Since the only change to A_3 is another round of computation and the inverse of an inverse is just the original value, we can safely say that A_2 is functionally equivalent to $R^+(x) = x + x^R$, which we said to be in L. Using a transitive property of equality we can then say $A_3 \in L$ because A_3 is poly-space reducible to $R^+(x) = x + x^R$.

5) Show UCYCLE $\in$ L. Let M be a deterministic TM that decides UCYCLE.

M = "On input (G)
1) Select a vertex x
2) Select an edge (x, y)
3) Start traversing the graph G through (x, y), if we come back to x through an edge that isn't (x, y) then accept otherwise reject."

If we are able to get back to x without encountering (x, y) then we can say the graph has a cycle, this is done in log space so UCYCLE $\in$ L.

6) Given $G = (V, E)$ the n nodes, the reduction produces $H = (V', E')$ Where $V' = \{(i, v, b) \mid 1 \leq i \leq n, v \in V, \text{ and } b \in \{0, 1\}\} \cup \{s, t\}$ and $E' = \{((i, v_1, b_1), (i, v_2, b_2)) \mid (v_1, v_2) \in E \text{ and } b_1 = 1 - b_2\} \cup \{(s, (i, v_1, 0)), (t, (i, v_2, 1)) \mid i < n\}$

(Credit to the Sipser text for this. I'm not 100% on it but did my best following along with the reduction examples online & in the Chapter)

7) Show that $BOTH_{NFA}$ is NL-complete. In order to non-deterministically compute M_1 and M_2 (NFAs) and to have $L(M_1) \cap L(M_2) = \emptyset$ we simulate both M_1 and M_2 in parallel for each guessed symbol and accept when they both end in accept. In order to prove completeness we reduce $BOTH_{NFA}$ to PATH. PATH is NL-complete so if we reduce $BOTH_{NFA}$ to PATH then $BOTH_{NFA}$ is NL-complete too.

HW3

Theory of Computation

Brian DeFlaminio

7 continued) Given a generic instance of PATH $\langle G, s, t \rangle$ we map it to $BOTH_{NFA}$.

M_1 = "M_1 is an NFA with $\Sigma = \{0\}$ and a state q_i for each node i in G. Then, for every edge (i,j) in G there's a δ from state q_i to q_j for input 0. $\delta_{M_1}(q_i, 0) = \{q_j \mid (i,j)$ is an edge in $G\}$. Next we use M_2 = "M_2 is an NFA and $\Sigma = \Sigma^*$". Since $L(M_1) \cup L(M_2) \neq \emptyset$ we know that neither language can include $\emptyset$ which is functionally equivalent to there being a path from s to t in G. Since we can map $BOTH_{NFA}$ to PATH like this, we can safely say $BOTH_{NFA}$ is NL-complete.

8) 2SAT is NL-complete. Since NL = coNL and that $2SAT = \{\langle\phi\rangle \mid \phi$ is a satisfiable boolean formula in 2CNF$\}$ we can show 2SAT is NL-complete by showing $\overline{2SAT}$ is NL-complete. The first condition for NL-completeness is proving $\overline{2SAT} \in NL$. For NTM M

M = "On input ϕ

1) Construct graph G such that ϕ is not satisfiable if for vertice (u,v) there are the paths uv and vu in G

2) for every variable x create a node x and $\bar{x}$

3) nondeterministically choose a vertex in G

4) check if G has both $x\bar{x}$ and $\bar{x}x$ paths via the NL-algorithm

5) Accept if both paths exist reject otherwise"

This decides $\overline{2SAT}$ in log space, so the first requirement is met: $\overline{2SAT} \in NL$.

Next we prove $\overline{2SAT}$ is NL Hard by reducing PATH to $\overline{2SAT}$ by mapping G to a 2cnf expression with a variable x_v for every v in G. $\phi = (\bar{x}_i \vee x_j) \wedge (\bar{x}_t \vee \bar{x}_s) \wedge (x_s \vee x_s)$ If we want ϕ to be true then x_s can't be false (i.e. G has a path from s to t) but we can reach a satisfiable assignment with x_i = False, x_j = False, x_T = False and x_s = True. This mapping of nodes in G to clauses in ϕ, and the subsequent assignment, shows 2SAT is NL-complete because $\overline{2SAT}$ is and NL = coNL