

CS500 Homework #4, Spring 2008

Note: Working with others on this homework is *allowed* and carries no penalty. However, you must do your own writeup, and you must state on your homework who you worked with. Due in class or by the end of the day on Tuesday, May 6th.

1. Problem 6.2 (parentheses in L)

Answer. Read the input from left to right. Maintain a counter c on the workspace with initial value zero, and increment or decrement it when reading ‘(’ or ‘)’ respectively. Reject if c ever becomes negative, and accept if $c = 0$ at the end of the input. Since c can never exceed the input length n , it is a $(\log_2 n)$ -bit number, so this only requires a workspace with $O(\log n)$ bits.

2. Problem 6.5 (addition in L)

Answer. Let x_i and y_i denote the 2^i -bit of x and y respectively. If we start with the least-significant bit and move to the left as we did in grade school, at each point we only need to keep track of two bits: r_j , the j th bit of the result, and c_j , the carry bit. Initially, we set $c_0 = 0$; then for each j , we have $r_j = x_j \oplus y_j \oplus c_j$ where \oplus denotes addition mod 2, and $c_{j+1} = 1$ if at least two of x_j, y_j, c_j are 1. When $j = i$, we return “yes” if $r_i = 1$.

3. Problem 6.6 (Reachability with pebbles)

Answer. We can define a graph G' where each vertex corresponds to a possible placement of the pebbles, and where two placements are adjacent if we can change one to the other by moving a single pebble. This problem is then just REACHABILITY on G' . If G has n vertices, there are at most $|G'| = n^k$ placements, so we can solve this problem in $\text{NSPACE}(\log |G'|) = \text{NSPACE}(k \log n)$. If $k = O(1)$, then $|G'| = \text{poly}(n)$ and the problem is in NL. If $k = \Theta(n)$, on the other hand, all we can say is that it is in $\text{NSPACE}(n^2) \subset \text{PSPACE}$.

4. Problem 6.7 (witnesses where we don't have to move left to right)

Answer. This is true of many of the NP-complete problems we have discussed. Let's take 3-SAT as an example. Given read-only access to a formula ϕ and a witness, i.e., a truth assignment for the variables x_1, \dots, x_n , we can scan the clauses of ϕ and check that each one is satisfied. We can do this in $O(\log n)$ workspace, since we just need a few counters to keep track of which clause we are currently checking, and which

variable x_i we are currently looking up. However, we will have to move back and forth on the truth assignment many times, since we have to look up the value of x_i once for each clause it appears in.

5. Problem 6.8 (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 the witness provides a path from u to v and we check it by looking at G . 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.

Now we will show that STRONG CONNECTEDNESS is NL-complete by reducing REACHABILITY to it. 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 change G into G' by adding some additional edges: specifically, for all vertices v we add edges $v \rightarrow s$ and $t \rightarrow v$.

To prove that this works, 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, since the $v \rightarrow s$ edges can only help us return to s , and the $t \rightarrow v$ 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, to show that this is a log-space reduction, we just need to show that given read-only access to G , s and t , we can answer the question whether there is an edge from u to v in G' using $O(\log n)$ workspace. This is easy: just answer “yes” if there is such an edge in G , or if $u = t$ or $v = s$, and answer “no” otherwise.

6. Problem 6.11 (SPACETIME)

Answer. Being in NSPACETIME($f(n), g(n)$) means that there is a single non-deterministic algorithm which works in $O(f(n))$ space and $O(g(n))$ time. Being in NSPACE($f(n)$) \cap NTIME($g(n)$), on the other hand, means that there is an algorithm that works in $O(f(n))$ space, and another, possibly different, algorithm that works in $O(g(n))$ time. Since optimizing an algorithm for space often makes it inefficient in time and vice versa, there might not be a single algorithm that satisfies both bounds.

Now, a problem in NSPACETIME($f(n), g(n)$) corresponds to a REACHABILITY problem in a graph of size $2^{O(f(n))}$, but where we are only interested in paths of length up to $O(g(n))$. Reviewing the proof of Savitch's Theorem, we see that applying middle-first search gives a stack of depth $O(\log g(n))$. Each state we store on the stack again takes $O(f(n))$ bits, so the total size of the stack is $O(f(n) \log g(n))$.

7. Problem 6.15 (Circuits to Geography)

Answer. One way to solve this is to first convert all the gates to NAND gates, using de Morgan’s laws and adding NOT gates if necessary. Then each NAND gate is simply a vertex with edges pointing to its inputs: a NAND gate is true if at least one of its inputs is false, and a vertex is winning if and only if at least one of the vertices we can move to is losing. Each false input becomes a dead-end vertex, while each true input becomes a vertex with one outgoing edge pointing to a dead-end vertex.

Alternately, suppose there are no NOT gates, so that the circuit consists just of ANDs and ORs. Let’s again call the first and second players the Prover and the Skeptic, since their goals are to show that the output is **true** or **false** respectively. At each OR gate it should be the Prover’s turn, since she can show that the OR is true by moving to a true input. Similarly, at each AND gate it should be the Skeptic’s turn, since she can prove that the AND is false by moving to a false input. If the AND and OR gates don’t alternate, we can switch whose turn it is by adding “dummy” vertices with one edge in and one edge out. These dummy vertices are just NOT gates in disguise, which switch the roles of Prover and Skeptic; in fact, they are the same NOT gates we would add to turn everything into NAND gates in the previous approach.

8. Problem 6.21 (Geography where we can visit the same vertex twice)

Answer. We can label the vertices using the following rules.

- (a) If every outgoing neighbor of v is a Win (in particular, if v has no outgoing neighbors) label v a Loss.
- (b) If v has at least one outgoing neighbor which is a Loss, label it a Win.

Iterate by sweeping through the vertices and applying these rules wherever they can be applied. As soon as a sweep results in no new changes, we have reached a fixed point. Then, label all the remaining unlabeled vertices as Draws.

9. Problem 6.22 (Go in EXPTIME)

Answer. A game where we are not allowed to return to any previous position, as in the “superko rule,” is essentially a game of GEOGRAPHY, except that the graph is exponentially large, with one vertex for every possible game position. (We can change whatever criterion the game uses to define a win or a loss to the one in GEOGRAPHY, where whichever player gets stuck loses, by allowing no moves from a losing endgame while adding one more move from a winning one.) Since GEOGRAPHY on polynomial-size graphs is PSPACE-complete, such a game is potentially EXPSPACE-complete.

Without this restriction, we can follow the same strategy as in Problem 6.21, labeling each game position as a Win, Loss, or Draw. The time it takes to do this is polynomial in the number of game positions, so we can solve such a game in EXPTIME. We can implement the simple ko rule by keeping track of both the current position and the previous one; this just squares the size of the graph, so we again get EXPTIME with a larger exponent.

In general, any ko rule which only requires us to remember only a constant, or even polynomial, number of game positions—such as a rule that we cannot visit any position we have seen in the last k moves for some $k = \text{poly}(n)$ —can be handled in EXPTIME. The problem with the “superko rule” is that, like the restricted form of GEOGRAPHY, it requires us to remember the entire past history of the game.

10. Problem 6.25 (Printing a puzzle solution)

Answer. The state space of a sliding block puzzle has N states, where $\log N = \text{poly}(n)$. Let s and t denote the initial and final positions respectively. As discussed in the text, we can tell in PSPACE whether there is a path from s to t . Specifically, we can solve the problem $R(s, t, \ell)$ of whether there is a path from s to t of length ℓ or less. Using $\log_2 N = \text{poly}(n)$ steps of binary search, we can find the length ℓ of the shortest such path.

Start by printing s . Then set $u = s$, and do the following loop. Go through all possible positions v we could move to from u , until we find one such that $R(u, t, \ell - 1)$ is true. This v lies on one of the shortest paths from u to t ; so print v , set $u = v$, and decrement ℓ . Continue this loop until $\ell = 0$ and $u = t$.