CS–172 Computability & Complexity, Spring 2021

Homework 10 Solutions

Note: These solutions are not necessarily model answers. Rather, they are designed to be tutorial in nature, and sometimes contain more explanation than is required for full points. Also, bear in mind that there may be more than one correct solution. The maximum total number of points available is 37.

1. (a) Our NL machine to decide 2-UNSAT runs as follows. Given as input a formula ϕ in 2CNF, it nondeterministically guesses an "offending" variable, say x, and a clause that contains it, say $(\neg x \vee y)$. Note that this clause expresses the Boolean implication $x \rightarrow y$, so we nondeterministically guess a clause containing $\neg y$, say $(\neg y \vee \neg z)$, which exprsses the implication $y \rightarrow \neg z$. We repeat this process, following that clause to one containing z, and so on. By guessing a sequence of such clauses we build up a chain of logical implications. We accept if the chain of implications starts and ends with x, and has passed through $\neg x$. We reject if the chain has run for more steps than there are clauses in the formula (since by that point we must be looping), or if there are no more clauses to follow. *7pts*

The machine runs in log space because it only needs to keep track of the current clause, the initial offending variable x, a flag recording whether or not it has visited the negated variable $\neg x$, and a step counter. Each of these items requires space $\log n$, where n is the number of variables in the formula.

Note that this algorithm is nondeterministically searching a directed graph defined from the input formula as follows: there are $2n$ vertices, one for each variable and for its negation, and there is an edge from a to b iff the clause $(\neg a \vee b)$ occurs in the formula. (Note that a, b here are arbitrary literals, so if we have, say, the clause $(x \vee y)$ then we include the edges $(\neg x, y)$ and $(\neg y, x)$.) To show the correctness of the algorithm, we prove the following claim: ϕ is unsatisfiable iff for some variable x there is a path in this graph from x to $\neg x$ and back to x.

Proof of $\Longleftarrow$: Proof by contradiction. Assume there is some truth assignment that satisfies ϕ, and that we have found such a cycle including both x and $\neg x$. Suppose that in this assignment x is set to true. Since there is a path from x (true) to $\neg x$ (false), there must be some edge (u, v) along the way such that the literal u is assigned true and the literal v is assigned false. But such an edge can only occur if the clause $(\neg u \vee v)$ is in the original formula, and clearly the assignment does not satisfy this clause. If x is set to false, we can apply exactly the same reasoning to the fact that there is a path from $\neg x$ to x. In both cases we get a contradiction, thus completing the proof.

Proof of $\Longrightarrow$: We prove the contrapositive. If there is no cycle of the above kind, then we can construct a satisfying assignment for ϕ as follows. Repeat the following process until all variables are assigned: select an unassigned variable (say, x). Then there must be either no path from x to $\neg x$, or no path from $\neg x$ to x. Suppose the first possibility holds. Then assign x and all nodes reachable from it true. Also assign false to the negations of these nodes. This step will never result in a conflicting assignment; for there to be one, there must be a path from the initial node x to both y and $\neg y$ for some y. However, by the symmetry of the graph (since $(a \vee b)$ induces both the edges $(\neg a, b)$ and $(\neg b, a)$), there must also be paths from y and $\neg y$ to $\neg x$. Hence there would be a directed path from x to $\neg x$, a contradiction. If on the other hand the second possibility holds (i.e., there is no path from $\neg x$ to x), then we perform the same procedure setting x to false, and by the same argument there will be no conflicts. Thus all nodes can be assigned a truth value respecting the constraints of all clauses and we are done.

[There were many issues with solutions to this part. Some students claimed that ϕ is unsatisfiable if there's a path from x to $\neg x$ for some x. This isn't enough, because it only gives us a contradiction from

*setting x true, not from setting x false: you need a path from x to $\neg x$ and back to x (i.e., a cycle that includes both x and $\neg x$) in order to ensure that ϕ is unsatisfiable. Some students failed to handle the case where such a cycle is not detected, i.e., they didn't specify when the algorithm should stop in that case. A serious error made by some students was to construct and store the implication graph derived from the clauses: this is not OK because it requires $O(n \log n)$ space to write down, where n is the number of clauses. Note that the above algorithm doesn't write down the entire graph! Some students gave a log-space reduction from 2-*UNSAT *to* PATH *and argued that this implies 2-*UNSAT *is in NL because* PATH *is in NL. This is correct, except that you need to explain how to combine the reduction and the algorithm for* PATH *into a logspace algorithm for 2-*UNSAT *(without writing out the output of the reduction, as we discussed in class). Finally, points were also deducted for failing to justify the algorithm or for failing to analyze its space complexity.]*

(b) Given an input $\langle G, s, t\rangle$ for PATH, we construct an input ϕ for 2-UNSAT as follows: label s with x, t with $\neg x$, and every other vertex with a unique identifier (such as its identifier in G itself). For each edge (u, v) in G, output the clause $(\neg u \vee v)$. Finally, output the clause $(x \vee x)$. *6pts*

This reduction can be done in log space, as identifiers need only $\log n$ bits and we process each edge separately. No other information needs to be stored. We now show that the reduction is correct.

Firstly, if there is a path from s to t in G, then there is a sequence of implications in the formula $x \Rightarrow ... \Rightarrow \neg x$ in the form of the clauses $(\neg x \vee y), (\neg y \vee z), ..., (\neg q \vee \neg x)$. This means that, in any satisfying assignment for ϕ, x must be false. But this in turn means that the clause $(x \vee x)$ cannot be satisfied. Hence ϕ is unsatisfiable.

Conversely, suppose there is no path from s to t in G. We will show that ϕ is satisfiable. If ϕ is unsatisfiable then, as proved in part (a), there must exist a cycle of implications that includes both a variable and its negation. But since there is no path from s to t, this cycle cannot include the variable x and its negation, so the contradiction must come from some other variable. However, every other variable occurs only in its positive, non-negated form in the graph, and so can never imply its own negation. (In fact, a valid assignment for this formula is to set all the variables to true.) Therefore ϕ is satisfiable.

[Some students did not include the extra clause $(x \vee x)$ to force x to be true. In this case, the CNF may still be satisfiable despite the existence of a path from s to t (similar to the issue with paths from x to $\bar{x}$ and from $\bar{x}$ to x in part (a)).]

(c) Note first that 2-SAT $= \overline{\text{2-UNSAT}}$ (modulo mal-formed inputs which we call "Junk" and can be detected in logarithmic space). Thus, since NL $=$ CO-NL and 2-UNSAT $\in$ NL by part (a), we know that 2-SAT $\in$ NL. Also, by part (b) above we know that any language $A \in$ NL is log-space reducible to 2-UNSAT; but this immediately implies that $\overline{A}$ is log-space reducible to $\overline{\text{2-UNSAT}} =$ 2-SAT. Hence 2-SAT is NL-complete. *1pt*

*[Some students only argued that 2-*SAT *is in NL but omitted to argue that 2-*SAT *is NL-hard.]*

2. To show that STRONG-CON is NL-complete, we must show that it belongs to NL and that it is NL-hard. *10pts*

To show that STRONG-CON $\in$ NL, we use two counters on the worktape to run through all (ordered) pairs of vertices u, v of G and, for each such pair, we nondeterministically guess a path from u to v (in the same way we did when we showed in class that PATH $\in$ NL). We accept only if we successfully guess a path for all pairs u, v. Here is the corresponding pseudocode (where n denotes the number of vertices in G):

```
for $u := 1$ to $n$ do
```

```
    for $v := 1$ to $n$ do
        $w := u;\ t := 0$
        while $t < n$ and $w \neq v$ do
            guess an edge $(w, w')$ and set $w := w'$
            $t := t + 1$
        if $w \neq v$ then halt and reject
halt and accept
```

This algorithm clearly runs in space $O(\log n)$ as it only needs to store the labels of three vertices and one additional counter up to n, each of which uses space $O(\log n)$.

To show that STRONG-CON is NL-hard, we give a reduction from PATH, which we already know from class is NL-hard. Given an input $\langle G, s, t\rangle$ for PATH, we construct an input $\langle G'\rangle$ for STRONG-CON as follows. The directed graph G' is the same as G, except that for every vertex $u \notin \{s, t\}$ we add a directed edge (u, s) from u to s, and a directed edge (t, u) from t to u. We also add the edge (t, s). (If any of these edges exist already, we just keep them.) This reduction can clearly be carried out in logarithmic space: first we just copy G to the output, and then we add the additional edges as needed, storing only two vertex labels at a time on the worktape. To verify that the reduction is correct, we need to prove that

$$G' \text{ is strongly connected} \iff \exists \text{ an } s \rightsquigarrow t \text{ path in } G.$$

Proof of $\Longleftarrow$: Suppose that there exists an $s \rightsquigarrow t$ path in G. Then for any two vertices u, v we can build a path $u \rightsquigarrow v$ in G', by first following the edge (u, s) (unless $u = s$), then the given $s \rightsquigarrow t$ path, and finally the edge (t, v) (unless $t = v$). Hence G' is strongly connected.

Proof of $\Longrightarrow$: Suppose that G' is strongly connected. Then in particular there exists an $s \rightsquigarrow t$ path in G' (which we may assume is a simple path since we can always remove loops). Note that such a path cannot make use of the extra edges added to G, since all these edges either point into s or out of t. Hence indeed there must exist an $s \rightsquigarrow t$ path in G itself.

[Students generally did pretty well on Question 2.]

3. (a) First, we show $\text{ALL}_{\text{DFA}} \in$ NL. Since NL = co-NL, we instead show that $\overline{\text{ALL}_{\text{DFA}}} \in$ NL. Ignoring malformed inputs, $\overline{\text{ALL}_{\text{DFA}}} = \{\langle D\rangle : D$ is a DFA and $\exists$ an input string w that is rejected by $D\}$. Suppose D has n states. If $\langle D\rangle \in \overline{\text{ALL}_{\text{DFA}}}$, then there must exist some w of length no more than n such that D rejects w (and if $\langle D\rangle \notin \overline{\text{ALL}_{\text{DFA}}}$, no such w exists). We can decide $\overline{\text{ALL}_{\text{DFA}}}$ in nondeterministic logarithmic space by guessing w one character at a time and simulating D on w. At each step we must remember the current state of D, a single character of our guessed w, and a counter for the length of the portion of w we have already guessed, all of which can be stored in space $O(\log n)$. We repeat until either we reach a rejecting state of D (in which case we accept) or we have already guessed n symbols of w (in which case we must have repeated a state, so we reject). *6pts*

To show ALL_{DFA} is NL-hard, we give a log space reduction from $\overline{\text{PATH}}$ (which we know is NL-complete, because PATH is NL-complete and NL = co-NL). Given a graph G with source and target vertices s and t (an instance of $\overline{\text{PATH}}$), we construct a DFA D whose states correspond to vertices in G and whose transitions correspond to directed edges in G. The start state of D is the state corresponding to vertex s. All states in D are accepting, except for the one corresponding to vertex t, which is rejecting. It should be clear from this construction that D accepts all input strings iff G has no path from s to t. What remains is to finish our construction of D to ensure it is a proper DFA, with exactly one outgoing transition from each state on each possible alphabet symbol. Let d be the maximum out-degree of any vertex in G; then we define the alphabet of D to be $\{1, 2, ..., d\}$. For each state in D, we label each outgoing transition with a unique alphabet symbol, and if there are fewer than d outgoing transitions then we add self-loops with all remaining alphabet symbols. We can construct D using logarithmic work space by, at each step, remembering the current vertex v in G, the current outgoing edge from v, and the number of outgoing edges from v already processed.

[An alternative approach to showing that $\text{ALL}_{\text{DFA}} \in \text{NL}$ *is to nondeterministically traverse the states of the DFA, starting at* q_0 *and accept if a non-accepting state is ever reached. The search rejects if no such state is found after* $|Q|$ *steps. This solution received full credit. Some students simulated the DFA without putting any restriction on the length of the input strings; the resulting TM may therefore never halt. In the reduction part, points were deducted for failing to fully specify or justify the DFA constructed.]*

(b) First, we show $\text{ALL}_{\text{NFA}} \in \text{PSPACE}$. Rather, we show $\overline{\text{ALL}_{\text{NFA}}} \in \text{NPSPACE}$, which suffices thanks to Savitch's theorem. Let N be an NFA with n states, input to $\overline{\text{ALL}_{\text{NFA}}}$. If $\langle N \rangle \in \overline{\text{ALL}_{\text{NFA}}}$, then $\exists w$ such that N rejects w. We guess w one symbol at a time, and maintain a list of which states N could be in after processing each symbol of w. We also maintain a counter for how many symbols of w have been guessed so far, and halt if either the current set of states is all rejecting (in which case we accept N, having found an input it rejects) or if we have already guessed 2^n symbols of w (in which case we reject N, having repeated a set of states without finding a rejecting set of states). We can maintain a subset of states with n bits, and a counter up to 2^n with another n bits, so the entire algorithm uses polynomial space. *6pts*

To show ALL_{NFA} is PSPACE-hard (and thus PSPACE-complete), we reduce any language $A \in \text{PSPACE}$ to $\overline{\text{ALL}_{\text{NFA}}}$. Since $A \in \text{PSPACE}$, there exists a TM M that decides A in polynomial space. Our reduction takes an input w for problem A, and must construct in polynomial time an NFA N such that N does not accept all input strings iff M accepts w.

We construct N to accept precisely those input strings that are not encodings of valid accepting computation histories of M on w. Then if M accepts w (i.e., if $w \in A$), the corresponding accepting computation history will be rejected by N (i.e. $\langle N \rangle \in \overline{\text{ALL}_{\text{NFA}}}$). If N rejects some string v (i.e., $\langle N \rangle \in \overline{\text{ALL}_{\text{NFA}}}$), then v must be a valid accepting computation history of M on w (so $w \in A$).

We now show how to construct N with the desired behavior in polynomial time. Recall that a computation history of M on w takes the form $\#C_0\#C_1\#C_2\#\ldots\#C_t\#$, where C_0 is the initial configuration and each C_{i+1} follows from C_i via a valid move of M. Since the space used by M is $f(n) = O(n^k)$ for some k, we may assume that $t \leq 2^{O(f(n))}$. Also, we may assume that all configurations C_i have the same length $f(n)$ (by padding with blanks as needed). N works by nondeterministically guessing a portion of its input string v that violates the requirement that v be an accepting computation of M on w. This guess may take one of several forms:

- **Bad start:** N reads the initial portion of its input v and accepts if any symbol fails to match the correct initial segment $\#C_0$. Clearly this requires only $f(n)$ states.
- **Bad end:** N reads its entire input and accepts if the symbol q_{accept} is not present. This trivial check can be done with a constant number of states.
- **Bad computation:** N guesses a pair of adjacent configurations C_i, C_{i+1} (delimited by the # markers) that do not follow from one another by a valid move of M. The (in)consistency check between C_i and C_{i+1} is handled using the 2x3 window idea from the proof of Cook's Theorem: i.e., N will guess a 2x3 window that is incorrect. To do this, N must first guess C_i by choosing

its beginning # delimiter, and then guess a position within C_i, which it can do using $f(n)$ states (one corresponding to each position in the configuration). It then remembers in its state the three symbols of C_i starting at this chosen position, and uses a further $f(n)$ states to skip over input characters until it reaches the corresponding position in C_{i+1}. Then it accepts if the three following symbols are not consistent with the three remembered symbols from C_i. This entire procedure requires $O(f(n)^2)$ states (since there is a separate set of skipping states for each position in C_i and each set of three remembered symbols), which is still polynomial.

Since the size of N is polynomial in the length of w, and its construction is mechanical given the (fixed) TM M and input w, the entire reduction can be carried out in polynomial time.

[Some students used the exact same argument as in part (a) to prove that $\text{ALL}_{\text{NFA}} \in \text{PSPACE}$. *As a result, they missed several details. For instance, they didnt realize they need to keep track of a subset of states instead of just one state. They also didnt realize it might take up to* 2^n *steps to exhaust all possible computations of the NFA. The proof that* ALL_{NFA} *is* PSPACE*-hard was quite involved and was graded rather leniently (and with relatively few points). However, this is really just a modification of the proof in class for* $EG_{REX\uparrow}$*. Students are encouraged to review the above solution for details of this part.]*

(c) DFAs and NFAs are equivalent in terms of the languages they can represent, but not equivalent in the efficiency of their representation. In general, given an NFA with n states, an equivalent DFA may require a number of states that is exponential in n (recall that in the NFA to DFA conversion, each DFA state corresponds to a subset of NFA states). Given that an NFA can encode a regular languages exponentially more efficiently (in the length of the representation) than a DFA can, it is not surprising that the complexity of answering some questions about NFAs should be exponentially larger than answering the same questions about DFAs. *1pt*