Recitation 10: Exercises in L, NL, and PSPACE

In this recitation we gain familiarity with space complexity through examples. We begin with an example of a game for which assessing whether a player has a winning strategy is a PSPACE-complete problem. We studied TQBF as a canonical example of a PSPACE-complete language. We’ll explore the essence of TQBF and relate it to a similar language which is also PSPACE-complete. We then move on to a discussion of log space. We will review log space reductions and demonstrate an important property they hold. Finally, we give an example of a NL-complete language.

Winning Strategies

What is a winning strategy? The word winning implies some competition. A competition has to have at least two competing entities which we will call Adam and Eve. Now that we have players, we need a game (i.e., some common framework with associated rules the players can use to compete). We start with a simple game called the Formula Game.

In the formula game, players Adam and Eve are presented with a fully quantified Boolean formula $\phi$ of the form:

$$\phi = \forall x_1 \exists x_2 \ldots \forall x_n \psi$$

Where $x_i$ are the variables and $\psi$ is a Boolean formula as we’re used to seeing from SAT. For every ‘for all’ $\forall$ quantifier, Adam selects a truth value of the corresponding variable. For every ‘there exists’ $\exists$ quantifier, Eve does the same. Players take turn correspondingly to the order of the quantifiers in $\phi$. At the end, if $\psi$ is True, Eve wins. Otherwise, Adam wins.

What does it mean for Eve to have a winning strategy? It means that Eve can always win even if Adam is infinitely intelligent and plays each move optimally. In other words, no matter what Adam does, Eve can always counter Adam’s moves to force a win. Going back to the formula game example, Eve has a winning strategy if and only if for any truth assignment Adam gives $x_1$, Eve can find a truth assignment for $x_2$. 
such that for any truth assignment Adam gives $x_3$, Eve can find a truth assignment for $x_4$ and so on... such that Eve wins (or such that $\psi$ is True). This is exactly TQBF! So here, the problem of deciding whether Eve has a winning strategy is exactly the problem of deciding whether $\psi$ is True.

**Generalized Geography**

In generalized geography, the game is as follows. We have a directed graph $G$ and a starting node $s$. We represent this instance as $(G, s)$.

Player 1 starts at node $s$. Player 1 has to transition to some neighbor node of $s$. Call it $v$. Now, starting from $v$, Player 2 has to move to a neighbor of $v$. The players continue making moves in this manner. Repeating moves is not allowed, so no player can transition to a node that has been visited before. A player loses (and the other player wins) if they get stuck and can’t transition anywhere anymore.

We define the following language for a game of Generalized Geography:

$$GG = \{ (G, s) \mid \text{Player 1 has a winning strategy starting from node } s \}$$

![Figure 1: A GG Example](image)

**Theorem 1.** $GG \in \text{PSPACE}$

While the proof of the Theorem is in the book, let’s go over the idea. What does it mean for player 1 to have a winning strategy? It means that there exists a first move for player 1 such that, whatever the first move of player 2 is, player 1 can counter it such that, whatever the second move of player 2 is, player 1 can counter it... Continuing in this manner player 1 eventually wins. Does that remind us of anything? Indeed, it should remind of the Formula Game we just presented.

How did we prove that TQBF $\in \text{PSPACE}$? We had a recursive algorithm that ran over every quantifier at each level of the recursion and tried possible Boolean values of the variables according to the corresponding quantifier type ($\forall$ vs. $\exists$). At its base case, it checked that
ψ is true. Here we do exactly the same thing. The following Turing machine \( M \) tests membership in \( G \) in PSPACE.

\[
M: \text{On input } (G, s) \text{ where } G \text{ is a directed graph and } s \text{ is a node of } G:
\]

1. For each node \( u \) where there exists an edge \( s \rightarrow u \) in \( G \):
   
   1. Remove node \( s \) and all connected edges from \( G \) to create a new directed graph \( G_{\neg s} \)
   2. Recursively call \( M \) on \( (G_{\neg s}, u) \)

   2. If all calls accept, Player II has a winning strategy, so \( M \) rejects. Otherwise, Player I has a winning strategy, so \( M \) accepts.

Instead of possible Boolean values, we have possible moves along the graph \( G \). Instead of a \( \exists \) quantifier, we have player 1 making a move. Instead of a \( \forall \) quantifier, we have player 2 making a move. The base case is when the current player loses by no longer being able to make a move. We’re essentially implementing a depth-first search on \( G \) to find whether player 1 has a winning strategy. Throughout the recursion, we have at most \( O(m) \) recursive levels where \( m \) is the number of nodes in the graph (because no repetitions). At each level, we’re only storing the current choice of node for that player which takes space \( O(\log m) \). Thus, the algorithm runs in \( O(m \log m) \) space.

**PSPACE Completeness**

You may be wondering, when showing that a language is PSPACE-complete, why do we use a polynomial time reduction rather than some sort of space reduction? In order to understand this question, take a look at the following diagram:

At each level of the recursion, we remove the last node and all of its edges and swap roles, so that after player 1 has moved the algorithm checks whether player 2 has a winning strategy starting at the next node.
The diagram emphasizes that the class of PSPACE-hard languages, consisting of languages that everything in PSPACE is reducible to, should only include a few problems that can be solved in PSPACE. If we took a reasonable definition for PSPACE-reducibility and used this in the definition of hardness then it’s trivial to see that all languages in PSPACE (except $\emptyset$ and $\Sigma^*$) would be PSPACE reducible to each other.

In order to have a meaningful definition of completeness we require reductions to be weaker than the language class in question. The class P appears to be sufficiently less powerful than PSPACE which makes it a good candidate for such reductions.

In order to have a meaningful definition of completeness we require reductions to be weaker than the language class in question. The class P appears to be sufficiently less powerful than PSPACE which makes it a good candidate for such reductions.

![Diagram](image)

**The Classes L and NL**

To define what it means for a TM to run in log space, we tweak the definition of our computational model a bit. Our TM now has two tapes: a read-only tape that is bounded and only includes the input, and a read/write work tape that is unbounded. When we compute the space used, we only consider the work tape. For instance, if a machine runs in $\log(n)$ space, then the machine uses at most $\log(n)$ cells on the work tape for any input of size $n$ on the read-only tape (or for any read-only tape of size $n$). Intuitively, we can think of programs that run in log space as using a finite number of pointers and counters (each requiring $\log n$ space).

Formally, we define log space and nondeterministic log space as follows:

**Definition 1.**

$$L := \text{SPACE}(\log(n)), \quad NL := \text{NSPACE}(\log(n)).$$

To show that languages are NL-complete we can’t use polynomial time reducibility for reasons similar to the PSPACE discussion, we
need a weaker form of reduction. Instead we use log space reducibility \( \leq_L \) which requires us to define a special kind of Turing Machine known as a log space transducer, which we demonstrate in the following figure.

![Diagram](image)

\( M \) is a log space transducer. It takes \( n \) symbols as input and computes \( f(w) \) on its output tape. The work tape has \( O(\log n) \) symbols.

This machine computes a log space computable function \( f \) for which the machine halts with \( f(w) \) on its output tape having started with \( w \) on its input. We now show that this type of reduction is transitive.

**Theorem 2 (Transitivity of \( \leq_L \)).** If \( A \leq_L B \) and \( B \leq_L C \), then \( A \leq_L C \).

**Proof.** Suppose that \( A \leq_L B \). This means there exists some log space transducer \( M_f \) which on input \( w \) halts with \( f(w) \) on the tape such that for all strings \( w \),

\[
    w \in A \iff f(w) \in B.
\]

Similarly \( B \leq_L C \) implies that there exists some \( M_g, g \) such that for all strings \( x \),

\[
    x \in B \iff g(x) \in C.
\]

Given these definitions, we wish to show the existence of a log space transducer \( M_{g \circ f} \) such that on input \( w \), \( M_{g \circ f} \) halts with \( g \circ f(w) = g(f(w)) \) on the tape. It’s not hard to see that for all strings \( w \),

\[
    w \in A \iff g \circ f(w) \in C.
\]

The naive solution would simply be to run \( M_f \) and then \( M_g \) on our input. However, this solution involves writing \( f(w) \) on our work tape as an intermediate step with no guarantee it can be written in log space.

Instead we design \( M_{g \circ f} \) such that on input \( w \), \( M_f \) computes individual symbols of \( f(w) \) as needed by \( M_g \). The machine \( M_{g \circ f} \) keeps track of where the head for \( M_g \)'s input tape would be on \( f(w) \). Each time a new symbol of \( f(w) \) is needed, we fire up \( M_f \) and delete the output until we get to the desired position on \( f(w) \). Note that the output tape is write-only, so there is no issue with deleting the symbols that we do not need to access, and only a single symbol of \( f(w) \) need be stored at any point in the computation. \( \square \)
Now that we have shown the \( \leq_L \) relation is transitive, we may use log space reductions to show completeness by reducing from complete problems in a particular language. In the following example, we demonstrate a language is NL-complete by reducing from \( PATH \) which we already showed in lecture to be NL-complete.

**Definition 2.** Given a directed graph \( G \). We say that \( G \) is strongly connected if for every pair of nodes \( u, v \), we have a directed path going from \( u \) to \( v \) and one going from \( v \) to \( u \).

![Figure 4: A simple strongly connected graph](image)

Given this definition, we define the following language:

\[
SC = \{ \langle G \rangle \mid G \text{ is a strongly connected directed graph} \}
\]

**Theorem 3.** \( SC \) is NL-complete

To show that \( SC \) is NL-complete we must prove the following:

1. \( SC \in NL \)
2. \( PATH \leq_L SC \)

\( SC \in NL \). We proceed by iterating deterministically over all pairs of nodes \( u, v \) in the directed graph \( G \). For every pair of nodes, we guess the path going \( u \to v \) like we did when we proved \( PATH \in NL \). We reject if for any pair of nodes \( u, v \) we cannot actually reach \( v \) from \( u \).

**M:** On input \( \langle G \rangle \):

1. For all pairs \( u, v \) of nodes in \( G \):
2. Use nondeterminism to guess a path \( u \to v \) in \( G \)
3. If any two nodes do not have a path connecting them, reject.
   
   Accept if a path exists connecting each pair.

Keeping track of the pair \( u, v \) requires two pointers which is logarithmic in space. We know that \( PATH \in NL \) so each guess of the paths \( u \to v, v \to u \) can be done in log space. Overall, the algorithm runs in nondeterministic log space.

If the graph is indeed strongly connected, then some sequences of guesses will succeed in finding a path for each pair of nodes, and we
accept. Otherwise, no sequence of guesses will succeed and we reject.

**PATH \( \leq_L \) SC.** We show that \( \text{PATH} \leq_L \text{SC} \). The idea will be to use the path from \( s \) to \( t \) if such a path exists to connect every two nodes in the graph to one another. The reduction, given \( \langle G, s, t \rangle \), will construct a new graph \( G' \) that has the same nodes as \( G \) and contains all of \( G \)'s edges. It will also add an edge from \( t \) to any other node and an edge from any other node to \( s \). If there is a path from \( s \) to \( t \), then there is a path from \( u \) to \( v \): \( u \to s \to t \to v \).

The reduction can be computed in \( L \). We assume that the graph is given as list of edges, ordered such that we start with edges that go out from \( s \) and end with those that go out of \( t \). An example input: \( (s,u_1)(s,u_2)\#(u_1,u_2)\#(u_2,t)\#(t,u_1) \). The reduction will copy this input to the output tape edge by edge while additionally adding the following edges:

1. At the end of the edge list of every node \( u_i \) the reduction adds the edge \( (u_i,s) \) if it does not already exist (need only remember the current node \( u_i \) and the node \( s \));

2. The reduction replaces \( t \)'s edge list with the edges \( (t,u_i) \) for every node \( u_j \) (need only remember current node \( u_j \) and \( t \)).

To show the the reduction is correct we argue the following:

\[ \Rightarrow \] Assume that \( \langle G, s, t \rangle \in \text{PATH} \), then there is a path from \( s \) to \( t \). Let \( u, v \) be two nodes. There is a path from \( u \) to \( v \): \( u \to s \to t \to v \). So, \( \langle G' \rangle \in \text{SC} \).

\[ \Leftarrow \] Assume that \( \langle G' \rangle \in \text{SC} \). There is a path from \( s \) to \( t \) in \( G' \). We argue that this path does not contain edges added by the reduction. Edges added by the reduction are either going into \( s \) or out of \( t \). Such edges cannot be a part of a path from \( s \) to \( t \). So, any path from \( s \) to \( t \) in \( G' \) must contain edges from \( G \) that connect \( s \) to \( t \). Hence, \( \langle G, s, t \rangle \in \text{PATH} \).