Recitation 10: Exercises in L, NL, and PSPACE

In this recitation, we’ll do some exercises that span the concepts of L, NL, and PSPACE. This is a lot! But what better way to get better than to practice?

Let’s start with PSPACE. We studied TQBF as a canonical example of a PSPACE-complete language. We’ll explore the essence of TQBF and relate it to similar languages that are also PSPACE-complete in the next sections.

Winning Strategies

What is a winning strategy? The word winning implies some competition. A competition has to have at least two competing entities so we call them players $A, E$. Now that we have players, we need a game (i.e. some common framework with associated rules the players can use/agree on to compete). We start with a very simple game, we call it the Formula Game.

In the formula game, players $A, E$ are presented with a fully-quantified boolean formula $\phi$ of the form:

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

For every $\forall$ quantifier, player $A$ selects a truth value of the corresponding variable. For every $\exists$ quantifier, player $E$ does the same. Players take turn correspondingly to the order of the quantifiers in $\phi$. At the end, if $\psi$ is True, player $E$ wins. Otherwise, players $A$ wins.

So what does it mean for player $E$ to have a winning strategy? It means that player $E$ can always win even if player $A$ had infinite intelligence. In other words, no matter what player $A$ does, player $E$ can always counter player A’s moves forcing a win. Going back to the formula game, player $E$ has a winning strategy if and only if for any truth assignment player $A$ gives $x_1$, player $E$ can find a truth assignment for $x_2$ such that for any truth assignment player $A$ gives $x_3$, player $E$ can find a
truth assignment for $x_4$ and so on... such that player $E$ wins (or such that $\psi$ is True). This is exactly TQBF! So here, the problem of deciding whether player $E$ has a winning strategy is exactly the problem of deciding whether $\phi$ is True.

The Dirty Dawg Vs. The Poor Sheep

One cartoonish way of visualizing winning strategies is to think about a "dirty dawg" chasing a "poor sheep". Assuming the dirty dawg corners the poor sheep against some high fence, then whatever move the poor sheep chooses to make, the dirty dawg can just mirror it while moving towards the sheep. Eventually, the dirty dawg will capture (and eat) the poor sheep. As you can see, the poor sheep had been doomed from the beginning because the dirty dawg had a winning strategy.

Generalized Geography

In generalized geography, the game is as follows. We have a directed graph $G$ and a starting node $b$. We represent this instance as $\langle G, b \rangle$. Player 1 starts at node $b$. Player 1 has to transition to some neighbor node of $b$. Call it $c$. Now, starting from $c$, Player 2 has to move to a neighbor of $c$, and so on... Note that repetitions are 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 can’t transition anywhere anymore.

Now, define the following language:

$$GG = \{ \langle G, b \rangle \mid \text{Player 1 has a winning strategy} \}$$

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

While the proof of the Theorem is in the book, let’s go over the idea. Again, 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, player 1 can counter and so on... such that player 1 wins eventually. Doesn’t that remind us of anything? Indeed, it should remind of the Formula Game we just presented. This, in turn, should remind us of what it means for a fully quantifiable boolean formula $\phi$ to be true (i.e. there exists ($\exists$) an assignment for $x_1$ such that, whatever ($\forall$) the assignment for $x_2$ and so on... such that $\psi$ is true).
To that end, how did we prove that TQBF ∈ PSPACE? We had a recursive algorithm going over every quantifier at each level of the recursion and trying out possible boolean values of the variables according to the corresponding quantifier type (∀ vs. ∃) and finally checking that ψ is true at the base case. Here we do exactly the same thing.

Instead of possible boolean values, we have possible moves along the graph G. Instead of a ∃ quantifier, we have player 1 making a move. Instead of a ∀ quantifier, we have player 2 making a move and so on... 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 node/choice for that player which is constant space. Thus, the algorithm runs in O(m) space.

The Classes L and NL

To define what it means for a TM to run in logarithmic 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 tape that is unbounded. When we compute the space used, we only consider the read/write tape. For instance, if a machine runs in log(n) space, then the machine uses at most log(n) cells on the read/write tape for any input of size n on the read-only tape (or for any read-only tape of size n).

Given this, we have the following definition

Definition 1

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

where determinism/non-determinism are defined similarly as before.

Let’s begin with the first exercise

Example 1 Let A be the language of properly nested parentheses, show that A ∈ L.

As an example of strings in A we have ( ∈ A, (()())( ∈ A but ()/ ∈ A. Let w = w₁...wₙ. To check that w ∈ A, at every index i, we have to ensure that the number of )’ parentheses seen so far (i.e. in w₁...wᵢ) doesn’t exceed the number of (’ within w₁...wᵢ. To that end, we keep a counter initialized at 0. For every (’ parenthesis, we increment by 1, for every )’, we subtract 1. At every point, we ensure that the counter is non-negative. We also ensure that the counter is exactly
Let’s move to a slightly more challenging example.

**Example 2** Let $B$ be the language of properly nested parentheses and brackets. Show that $B \in L$.

As an example of strings in $B$, we have $([], ()[()])() \in B$, but $[] \notin B$. Notice that the simple algorithm we had for $A$ where we increment/subtract every time we see a left/right parenthesis/bracket doesn’t work here anymore. Take $[])$ for example, in which case we have our $A$ counter going $0, 1, 2, 1, 0$; always non-negative and ends at 0 even though $[])$ \notin B.

The $A$ algorithm isn’t that far away from the correct solution though. In particular, it is sufficient to check that for every left parenthesis/bracket we have a matching parenthesis/bracket of the same type. For every such left parenthesis/bracket, we can find the matching right parenthesis/bracket using the algorithm in $A$. We then check that it’s of the same type (i.e. '(' matches with ')' and '[' matches with ']'). We repeat this for every such left parenthesis or bracket. We are done.

In addition to the space taken by the algorithm in $A$ (which is logarithmic), we have to keep a counter to track which left parenthesis/bracket we’re dealing with. Storing the counter also takes $\log(n)$. This proves $B \in L$.

Enough of the class $L$, let’s take an example of a language in $NL$! To that end, we make the following definition:

**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 $u$ (I’m kidding it’s $u$).

Define the following language:

$STRONGLY-CONNECTED = \{ \langle G \rangle \mid G \text{ is a strongly connected directed graph} \}$

**Theorem 2** $STRONGLY-CONNECTED \in NL$

Since we’re allowed to use a non-deterministic machine, we might get tempted to start guessing left and right. Calm down, and take a deep breath. Life has more to it.

Instead, we’ll proceed by iterating *deterministically* over all pairs of nodes $u, v$. This is where the guessing comes in. For every pair of
nodes, we guess the path going $u \to v$ and the one going $v \to u$ just like we did when we proved that PATH $\in$ NL. If we couldn't find a path in either direction, that branch rejects. Otherwise, we keep going (i.e. we keep iterating over pairs $u,v$). If we reach the end of the loop (i.e. if we iterated over all pairs of nodes), then we accept.

Keeping track of the pair $u,v$ requires two pointers which is logarithmic in space. We know that PATH $\in$ NL so guessing the path $u \to v, v \to u$ is logarithmic space. Overall, the algorithm runs in logarithmic space.

Now for correctness. Suppose that $G$ is strongly connected. Then there exists a branch (a very lucky one) that guesses the correct path $u \to v, v \to u$ for every pair $u,v$. That branch will accept so the machine will accept and we’re good. On the other hand, if $G$ is not strongly connected, then we can find some pair of nodes $u,v$ such that there’s no path $u \to v$. Then for every branch (even the lucky ones), will eventually reach this pair, won’t be able to guess a path $u \to v$ because it doesn’t exist, and it will reject. Since every branch rejects, the machine rejects. So we are good.