Reading.

- Arora-Barak § 4.2, 4.3

**Last time:** Space complexity, Savitch’s Theorem, PSPACE

## 1 TQBF is PSPACE-complete

Let’s pick up where we left off in exhibiting a PSPACE-complete problem. Recall that we defined a (fully) quantified Boolean formula as

$$\Psi = Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \varphi(x_1, \ldots, x_n)$$

where each quantifier $Q_i = \exists$ or $\forall$.

**Definition 1.**

$$\text{TQBF} = \{\Psi \mid \Psi \text{ is a true quantified Boolean formula}\}.$$  

**Theorem 2.** TQBF is PSPACE-complete.

*Proof.* We showed last time that TQBF $\in$ PSPACE by giving a space-efficient recursive algorithm. Now we need to show that for every language $L \in$ PSPACE, we have $L \leq_p$ TQBF. Let $L$ be such a language and let $M$ decide $L$ in (polynomial) space $S(n)$. Our goal is to, in poly-time, convert an instance $x$ into a QBF $\Psi$ such that $M(x) = 1 \iff \Psi \in \text{TQBF}$.

Our first idea will be to define a two-player game such that Player 1 has a winning strategy in this game iff $M(x) = 1$. Then we’ll formalize this game into a QBF.

Recall from our discussion of configuration graphs that

$$M(x) = 1 \iff \text{there exists a path from } C_{\text{start}} \text{ to } C_{\text{acc}} \text{ in } G_{M,x}.$$  

Consider the following (informal) game:

**Player 1:** The goal is to show that there exists a path (of length $2^{O(S(n))}$) from $C_{\text{start}}$ to $C_{\text{acc}}$.

**Player 2:** The goal is to show that there is no such path.

When it’s Player 1’s turn to move, they’ll pick a vertex $v$ that’s on the alleged path from $C_{\text{start}}$ to $C_{\text{acc}}$.

When it’s Player 2’s turn, they’ll issue a “challenge” to recurse either to the left or right of $v$, i.e., to force Player 1 in the next round to either exhibit a vertex on the path from $C_{\text{start}}$ to $v$ or from $v$ to $C_{\text{acc}}$. And so on and so forth...
The point of this game is that Player 1 has a winning strategy iff there indeed exists a path from $C_{\text{start}}$ to $C_{\text{acc}}$ in $G_{M,x}$.

Now let’s turn this intuitive description of a game into a QBF. Let $m = O(S(n))$ be the number of bits needed to encode one configuration (vertex of the configuration graph). The idea will be to recursively construct formulas of the form $\Psi_i(C, C')$, for $C, C' \in \{0, 1\}^m$, such that

$$\Psi_i(C, C') \in \text{TQBF} \iff \exists \text{ a path of length } \leq 2^i \text{ from } C \text{ to } C'.$$

The final formula we want will be $\Psi_m(C_{\text{start}}, C_{\text{acc}})$.

**Base case:** If $i = 0$, we use the proof of the Cook-Levin Theorem to encode the question of whether there is a transition from $C$ to $C'$ as an unquantified formula $\Psi_0(C, C')$.

**Recursive case:** As a first attempt, we might want to define $\Psi_i(C, C') = \exists v \Psi_{i-1}(C, v) \land \psi_{i-1}(v, C')$. The problem with this is that the size of the formula doubles with each call, so we’d end up with an exponentially long formula.

A better idea is introduce auxiliary variables to capture an equivalent condition without blowing up the formula size. One way to do this is to define

$$\Psi_i(C, C') = \exists v \forall x, y (x = C \land y = v) \lor (x = v \land y = C') \implies \Psi_{i-1}(x, y).$$

Note that one can unpack the $\implies$ connective using ORs and negations, and “push all quantifiers” in $\Psi_{i-1}$ to the left of the whole expression.

The time it takes to generate each formula is polynomial in the size, which by induction, is at most $|\Psi_i| \leq O(m^2)$.

## 2 Logspace Computation

Recall our scaled down complexity classes

$$L = \text{SPACE}(\log n)$$

$$NL = \text{NSPACE}(\log n)$$

These classes are quite a bit more practically motivated than $\text{PSPACE}$, intuitively capturing algorithms that use much less working memory than the size of the input. This models situations like web-crawling, database search, DNA sequence analysis, etc.

An important special case of space-bounded computation is streaming, were the goal is to design (poly)-logspace algorithms that make a single pass over the input.

The open question of the day is: Is $NL = L$? Recall that while Savitch’s Theorem tells us that $\text{PSPACE} = \text{NPSPACE}$, it doesn’t quite bring us to answering this question.

As usual, we’ll use reductions to help us study this question, but poly-time reductions are now too coarse-grained.

**Definition 3 (Logspace Reductions).** A language $A$ is logspace reducible to $B$, written $A \leq_l B$, if there exists a logspace computable function $f : \{0, 1\}^* \to \{0, 1\}^*$ such that

$$x \in A \iff f(x) \in B \quad \forall x \in \{0, 1\}^*.$$
Important: Recall our convention that the output tape of a TM is write-once and write-only, and we do not charge for space. This definition allows for logspace reductions to compute functions whose output length is \( \text{poly}(n) \).

Note that Arora-Barak uses a different convention for charging for space usage. As such, they need to introduce a different definition of logspace reductions to take care of this issue. They say that a function \( f \) is “implicitly” logspace computable if given as input \( \langle x, i \rangle \), a TM can compute whether \( i \leq |f(x)| \) and whether \( (f(x))_i = 1 \) in logspace. The two definitions are equivalent.

**Claim 4.** if \( f \) and \( g \) are logspace computable, then so is \( h(x) = f(g(x)) \).

Here’s a first idea to prove this, which doesn’t work, but for an instructive reason.

**Idea 1:** Let \( M_f \) compute \( f \) and \( M_g \) compute \( g \). On input \( x \):

1. Run \( M_g(x) \) and write the output to a work tape.
2. Run \( M_f \) on \( g(x) \).

The problem with this idea is that \( |g(x)| \) might be too long to write down in logspace.

To fix this, we use

**Idea 2:** Compute each bit of \( g(x) \) “on the fly” as needed. That is, on input \( x \):

1. Initialize a simulation of \( M_f \) on a “virtual” input \( y \).
2. Every time \( M_f \) wants to read a bit \( y_i \), simulate \( M_g(x) \) (without actually writing any of its outputs) until its output head would reach index \( i \).

**Corollary 5.** Logspace reductions are transitive, i.e., \( A \leq_l B \) and \( B \leq_l C \) imply \( A \leq_l C \).

**Corollary 6.** If \( A \leq_l C \) and \( C \in L \), then \( A \in L \).

## 3 NL-Completeness

NL-completeness follows the usual template for such definitions, except it is defined in terms of logspace reductions.

**Definition 7.** A language \( B \) is NL-complete if

1. \( B \in \text{NL} \)
2. \( A \leq_l B \) for all \( A \in \text{NL} \).

We have a simple and natural example of an NL-complete problem:

\[
\text{PATH} = \{ \langle G, s, t \rangle \mid \text{Digraph } G \text{ has a path from } s \text{ to } t \}.
\]

You may wonder about the version of this problem for undirected graphs. This problem is in \( L \), and was a major result of Reingold proved circa 2005.

**Theorem 8.** PATH is NL-complete.

**Proof.** As usual, there are two things to show.
PATH ∈ NL. The idea is to nondeterministically guess a path from \( s \) to \( t \). Let \( n \) be the number of vertices in the input graph \( G \). Note that there is a path from \( s \) to \( t \) in the graph iff there is such a path of length at most \( n \).

1. Set \( v = s, \ell = 0 \).
2. While \( \ell \leq n \):
   3. Update \( v \) to a nondeterministically chosen neighbor
   4. If \( v = t \), accept
   5. Increment \( \ell \)
   6. Reject.

The space usage is \( O(\log n) \) to maintain the identity of the current vertex \( v \), plus \( O(\log n) \) to maintain the step counter \( \ell \).

PATH is NL-hard. Let \( A \in \text{NL} \) and let \( M \) be an NTM deciding \( A \) in space \( O(\log n) \). We give a logspace reduction \( f \) from \( A \) to PATH.

Define \( f(x) = (G_{M,x}, s, t) \) where

- \( G_{M,x} \) is the configuration graph of \( M \) on \( x \). That is, vertices are configurations, and there is an edge \( C \rightarrow C' \) whenever the transition function of \( M \) can take configuration \( C \) to \( C' \).
- \( s = C_{\text{start}} \) and \( t = C_{\text{acc}} \).

**Correctness:** We have that
\[
x \in A \iff \exists \text{ an accepting computation path of } M \text{ on } x.
\iff \exists \text{ a path from } C_{\text{start}} \text{ to } C_{\text{acc}} \text{ in } G_{M,x}
\iff (G_{M,x}, s, t) \in \text{PATH}
\]

**Space usage:** Each configuration \( C \) (a vertex of \( G_{M,x} \)) takes \( O(\log |x|) \) space to write down. (Note that the big \( O \) hides a dependence on the machine \( M \), but that is fixed independent of \( x \) once the problem \( A \) is specified.)

Thus, we can write down (in a write-once fashion) the adjacency matrix of \( G_{M,x} \) by just checking if each pair \( (C, C') \) is consistent with \( M \)'s transition function.

\[\square\]

4 The Verifier View of NL

As with NP, we can formulate an alternative view of NL as the class of languages that admit logspace deterministic verifiers. However, we have to be careful about how we define a verifier.

**Theorem 9.** A language \( A \in \text{NL} \) if and only if there exists a TM \( V \) with a read-once “certificate tape” and a polynomial \( p \) such that for all \( x \in \{0,1\}^* \),
\[
x \in A \iff \exists u \in \{0,1\}^{p(|x|)} \quad M(x, u) = 1.
\]

Here, \( x \) is given to \( M \) on its input tape while \( u \) is given on its (again, read once) certificate tape.
Proof. For the “only if” direction: Let $A \in \text{NL}$ be decided by a logspace NTM $N$. Then $x \in A$ iff there exists a sequence of nondeterministic choices leading $N$ to accept on input $x$.

Use a certificate $u$ to encode such a sequence of nondeterministic choices. We construct a verifier $V(x, u)$ that simulates $N$ on $x$ using $u$ as the nondeterministic choices, accepting iff the simulation accepts.

For the “if” direction: Suppose $A$ has a logspace verifier $V$.

First idea: using nondeterminism, guess a certificate $u$ on some work tape, and then verify it. The problem is the same one we encountered with composing logspace reductions: Writing down $u$ requires polynomial space, not logspace.

Better: Guess a certificate $u$ bit-by-bit on the fly as the verifier needs them. This is ok because the verifier only needs read-once access to the certificate, so there’s no need to record previous bits of $u$ anywhere explicitly. □