1 PSPACE completeness and TQBF

We covered Section 4.2, PSPACE Completeness, with the study of TQBF (Totally Quantified Boolean Formulas). I think the text is clear, and I essentially followed it, so I will refer you to it, instead of notes. We covered Section 4.2.1 (but not 4.2.1)

During the previous lecture we looked at $NL$. $L$ and the GAP (Graph Accessability Problem – given directed graph $G = (V, E)$ and vertices $s, t \in V$, is there a directed path from $s$ to $t$?) and showed that GAP is complete for $NL$ under logspace reductions.

We spent much of the lecture proving the surprising (and nontrivial) theorem that NL is closed under complementation.

2 Immerman-Szelepcsényi Theorem: $NL=\text{coNL}$

It seems that nondeterminism and its complement are quite different. For the first, we have to give a proof that something happens, while for the second we need a proof that something does not happen. The latter seems harder:
instead of exhibiting a particular solution, we need to show that a solution
does not exist.

Turns out that for space bounded computation our intuition fails us.

**Theorem 1** (Immerman-Szelepcsényi). $NL = co-NL$

**Proof.** Since GAP is complete for NL, its complement is complete for coNL,
so we need to show that $\overline{\text{GAP}} \in NL$. $\overline{\text{GAP}}$ is the problem ‘Given graph
$G = (V, E)$ and vertices $s, t \in V$ is it the case that there is no directed path
from $s$ to $t$?’.  

The proof of the theorem is to provide a solution for this problem in $NL$.
In other words, show that if there is no such path than there is a proof $z$ that
can be verified by a deterministic logspace bounded machine, that attest to
this fact (completeness), and if there is such a path then no string $z$ will
lead the verifier to (falsely) certify that there is no path (soundness). Recall
that the logspace verifier has only read only, read once access to $z$ ($z$ is on a
special input tape where the head must move from left to right.)

Our textbook provides a clear and complete proof (pages 91-92).

I will not reproduce it here.

However, I will try to sketch how one arrives to such a proof, and make
very clear what information is stored on the worktape of the verifier, and
what is stored on the 1-way proof tape of the verifier.

$\square$

### 2.1 Failed attempts at a proof, and how to correct them

We need to show how to give a proof and a verification algorithm to show
that a specific $t \in V$ is not reachable from $s$. So let’s start by generalizing
the problem to look at the reachability (from now on ‘reachability’ will stand
for ‘reachability from $s$’) for an arbitrary $v \in V$.

It is not hard to give a proof when $v$ is reachable: the proof $z$ is a sequence
of vertices $u_0, u_1, u_2, \ldots u_k$ such that

1. $u_0 = s$
2. for $0 \leq i \leq k - 1$, $(u_i, u_{i+1}) \in E$
3. $u_k = v$
The prover stores (on its worktape—we’ll call it a ‘register’) the target vertex \( v \) (this will take \( O(\log |V|) \) squares), then verify that the first vertex of the proof is \( s \), and that condition (2) above holds for \( i = 0, \ldots k - 1 \). This can be done with 2 registers (plus a constant number to find the representation of the edge on the input tape.)

Finally, we verify that the last vertex is \( v \).

So, in \( NL \), we can certify when \( v \) is reachable. How to certify that it isn’t?

(The obvious problem with complementing a nondeterministic computation is that a string \( z^\prime \) that does not work with the verification algorithm above is *not* a proof that we cannot reach \( v \)—it only shows that \( z^\prime \) is not a proof that \( v \) is reachable.)

A first idea is the following: if we could list *all* reachable vertices, we could certify that \( v \) is not reachable, by discovering that it is not on the list.

The problem is that this list may well be \( O(|V|) \) long, and we cannot maintain it when we have only logarithmic tape. However, we may not need to store the list—since we can reuse the worktape, all we need is to examine all the reachable vertices. The good news is that this could be on the proof tape, and we just saw that we can certify each of them using a constant number of logspace registers. So the strategy should be

the proof tape should be a list of the form
\[ u_1 \text{ proof that } u_1 \text{ is reachable, } u_2 \text{ proof that } u_2 \text{ is reachable, } \ldots \text{ } u_k \text{ proof that } u_k \text{ is reachable} \]
where \( k \) is the number of reachable vertices
as we see each \( u_i \) we compare it with \( v \) and certify that it is not on the list

So what is wrong with this algorithm?

Hint: \( k \).

In other words, we do not know how many vertices are reachable. As a result, the prover can be fooled. If \( z^\prime \) is a string as above, but one where some reachable vertex \( u \) is omitted, the algorithm does not detect that. In other words, \( v \) may be reachable, yet we omit it from our list, and the algorithm will happily declare that \( v \) is not reachable.

Bad.

So let’s imagine for a moment that we do in fact have \( k \).
Can we make the idea above work?
Consider the proof above, with the additional check that every time we prove a new $u$ to be reachable, we add 1 to a counter (we only need an extra logspace register). Now we check that $z$ has exactly $k$ vertices.

Unfortunately, this doesn’t work either. It is not enough to find $k$ vertex names – they must be the names of distinct vertices, and our verifier does not check for that. (In other words we could have a string $z$ where for some $j \neq i$ $u_i = u_j$. This, in effect, reproduces the same possible error as our first attempt.)

What is worse, in order to certify that the $u_i$ are distinct, the natural algorithm would remember the ones we already saw on the proof tape, and when seeing the next vertex, check that it is distinct from all previous ones–but this requires storing possibly $\Omega(|V|)$ vertices, and we only have logarithmic memory.

The good news is that there is a trick for that: if the string $z$ has the vertices $u_i$ in increasing lexicographic order, we can ensure that there are no duplicates in the list, by remembering the last vertex seen, $u_j$, and checking that the following vertex, $u_{j+1}$, is greater. Again, this just take a single register (at this point we do not need to keep the previous vertex $u_{j-1}$ any more).

Let us recapitulate what we achieved so far:

Given an input tape with the graph $G = (V,E)$ (for example by the $|V|^2$ bits of its adjacency matrix), vertices $s$ and $t$ and the cardinality, $k$, of the set of vertices reachable from $s$, we can provide a proof $z$ and a logspace algorithm that verifies that $t$ is not reachable. The string $z$ is $u_1$ proof the $u_1$ is reachable, $u_2$ proof that $u_2$ is reachable, ... $u_k$ proof that $u_k$ is reachable

For each $u_i$ we verify that

$u_i > u_{i-1}$
$u_i$ is reachable
$u_i \neq v$

This verifies that $v$ is not reachable.

Now, we have to somehow figure out how to compute $k$, and we’ll be done.

A natural idea is to get inspired by Breadth First Search. Let $C_0 = \{s\}$, and let $C_i = \{x \in V$ the shortest path from $s$ to $x$ has no more than $i$ edges\}. 4
It should be clear that the ‘new’ vertices in $C_i$ that were not in $C_{i-1}$ use exactly $i$ edges in their shortest paths from $s$, and that these are the ‘levels’ found by the Breadth First Search (we are not using BFS in our algorithm! This is just a characterization).

We can now make a nice observation: if we had a proof that $|C_i| = m$ for some $m$, we could remember this value, and given a proof tape with a sequence of $m$ vertices, each followed by the path leading to it, we could in effect certify that this is a list of the vertices of $C_i$, listed in increasing order. As we saw, this is enough to also certify that a given vertex $v$ is not in $C_i$. Actually, we can do more with this information: we can verify that $v \not\in C_{i+1}$!

To do so we modify slightly the previous algorithm: as we look at each $u_j$ on the list $z$, (remember, we verify that these are in increasing order, and are reachable) we check that $v \neq u_i$ and that for each vertex $w$ such that $(u_j, w) \in E$ (the neighbors of $u_j$) $w \neq v$.

The attentive reader will have noticed that in this process we have in effect listed all vertices of $C_{i+1}$. The very attentive reader will also have noticed that, unfortunately this list is not ordered, and may have repetitions in it—so we cannot extract $|C_{i+1}|$ from it. (If we could, we’d be done, as we could eventually find $|C_{|V|}|$, and solve our problem...)

Still, now we are essentially done. With some more elaborate proof tape we can do as follows:

To get $|C_{i+1}|$ (having computed $|C_i|$)

$\text{counter} = 0$;

For every vertex $w \in V$

if $w \in C_{i+1}$ the oracle tape will have a path from $x$ to $w$, which we verify

$\text{counter} = \text{counter} + 1$; $\text{next}(w)$

if $w \not\in C_{i+1}$ we have an oracle tape segment which allows us to verify this by the algorithm above (we can verify that the tape has a list $C_i$’s vertices in increasing order—which we can do since we have $|C_i|$—look at each of the listed vertices and their neighbors, and verify that none of these is $w$)

$\text{At the end of the algorithm above, counter will have } |C_{i+1}|.$

This allows us to increase $i$, and eventually compute the number of reachable vertices—which allows us to certify each of the nonreachable ones.

Exercise: estimate the length of the proof tape given by this process (and show that it is polynomial in $|V|$)