CS221: Computational Complexity Prof. Salil Vadhan

Lecture 9: Savitch's theorem, NL = co-NL

10/9 *Scribe: Nicholas Shiftan*

Contents

1 Proving Reachability is NL-Complete

First, we define the Reachability problem:

$$\text{RCH} = \{(G, s, t) : \exists \text{ a path from } s \text{ to } t \text{ in } G\}$$

where G is a *directed* graph.

Theorem 1 Reachability *is* **NL***-complete.*

Proof: To prove that Reachability is **NL**-complete, we need prove that RCH $\in$ **NL** and that RCH is **NL**-hard. We proved that RCH $\in$ **NL** in the last lecture. To prove that RCH is **NL**-hard, we will attempt to reduce the problem of deciding any arbitrary language $L \in$ **NL** to the question of Reachability . Given any language $L \in$ **NL**, $\exists$ a logspace NTM which accepts L. Given an instance x, we can construct a 3-tuple (G, s, t) such that $\exists$ a path $s \rightsquigarrow t \Leftrightarrow x \in L$. We construct this tuple as follows.

Let $G = (V, E)$ be the configuration graph of M on x, where V represents all possible configurations of M on x. (Note that G is of size poly($|x|$) because G because M is logspace.) As we could guess, we define E to represent all legal movements from one configuration to another. Formally, $E = \{(u, v) :$ one can get from config u to config v in one timestep$\}$. In other words, this represents the transition table acting on configurations. We let $s =$ the initial configuration of M on x, and we lot $t =$ the accepting configuration. Without loss of generality, we assume that there is a unique accepting configuration; were there more than one, once could easily add 1 additional state and 1 additional transition such that they all ended up in the same final configuration. ■

2 NSPACE vs. TIME

There are a number of corollaries to **Theorem 1**.

Corollary 2 $\mathbf{NL} \subseteq \mathbf{P}$

Proof: RCH $\in \mathbf{P}$ (via a Breadth-First-Search), and $\mathbf{P}$ is closed under reductions. ■

Corollary 3 $\mathbf{NSPACE}(f(n)) \subseteq \bigcup_c \mathbf{TIME}(2^{cf(n)})$ *for all proper* $f(n) \geq \log n$

Proof: We can prove this via padding. Let $g(n) = 2^{f(n)}$. Then $\mathbf{NSPACE}(\log g(n)) \subseteq \mathbf{TIME}(g(n)^c)$. ■

3 NSPACE vs. SPACE

Recall: Breadth-First-Search takes Space $\Theta(n)$.

Theorem 4 (Savitch) RCH $\in \mathbf{SPACE}(\log^2 n) \stackrel{\text{def}}{=} \mathbf{L}^2$

Note: You can find this proof on page 149 of Papadimitriou

Proof: First, some definitions. Given a graph $G = (V, E)$ and nodes $x, y \in V$, we define the predicate PATH as follows:

$$\text{PATH}(G, x, y, i) = \begin{cases} \text{TRUE} & \text{if } \exists \text{ a path of length} \leq 2^i \text{ from } x \text{ to } y \\ \text{FALSE} & \text{otherwise} \end{cases}$$

An immediate result of this definition is that we can now define the Reachability problem in terms of deciding PATH. After all, it follows that

$$(G, s, t) \in \text{RCH} \Leftrightarrow \text{PATH}(G, s, t, \lceil \log n \rceil) = \text{TRUE}.$$

So, how we do decide PATH(G, s, t, i) in Log Space? The answer lies in a simple fact: if $\exists$ a path $s \rightsquigarrow t$, then there are two distinct possiblities. Either $(s, t) \in E$, or $\exists$ some vertex $x \in V$ such that $s \rightsquigarrow x$ and $x \rightsquigarrow t$. It should also be noted that if we're looking for a path $s \rightsquigarrow t$ on length $< 2^i$, then we can also find an x such that both the path $s \rightsquigarrow x$ and the path $x \rightsquigarrow t$ are of length $< 2^{i-1}$.

Thus, a we will use a recursive algorithm to decide PATH. Consider the following algorithm:

```
PATH(G, x, y, i) {
.    if i = 0 {
.        if x = y return TRUE;
.        else return FALSE;
.    }
.    for each vertex z ∈ V {
.        if PATH(x, z, i − 1) ∧ PATH(z, y, i − 1) return TRUE;
.    }
.    return FALSE;
}
```

A brief argument needs to be made to explain why this algorithm is, in fact, $\in \mathbf{L}^2$. The logic is fairly straightforward. Suppose we have a 4-tape TM, with an input tape, an output tape, and

2 work tapes. The first tape serves as a "stack" to implement the recursive data structure. It contains a series of 3-tuples (x, y, i), each of which take $3 \log n$ bits to represent. Since we can only go $\log n$ levels deep, it follows that this tape uses spaec $(\log n) \cdot (3 \log n) = \mathcal{O}(\log^2 n)$. Since all the information needed to the existence of a certain path can be found on the input tape and the first work tape, we need the second tape only for work purposes (i.e. to keep track of cursor locations and such). Thus it only requires $\mathcal{O}(\log n)$ space. And thus our total space requirements are $\mathcal{O}(\log^2 n)$. ■

Note: Of course, this algorithm does require time $\mathcal{O}(\#\text{ of leaves}) = \mathcal{O}((2n)^{\log n}) \neq \mathcal{O}(\text{poly}(n))$; we can do much better using other, more space-greedy algorithms. It is an open question whether there is an inherent tradeoff between time and space.

This proof has a number of immediate corollaries.

Corollary 5 $\mathbf{NL} \subseteq \mathbf{L}^2$.

Corollary 6 $\mathbf{NSPACE}(f(n)) \subseteq \mathbf{SPACE}(f(n)^2)$ *for all proper* $f(n) \geq \log n$

This proof is, as above, via padding. See Papadimitriou, pp. 150 for a more direct proof.

Corollary 7 $\mathbf{NPSPACE} = \mathbf{PSPACE}$

Open Question: Does $\mathbf{NL} = \mathbf{L}$?

4 NSPACE vs. co-NSPACE

Theorem 8 (Immerman-Szelepscényi) REACHABILITY $\in$ **co-NL**.

Note: You can find this proof on page 151 of Papadimitriou

Proof: To prove this, we need to give an **NL** algorithm for deciding UNREACHABILITY. Using a simple construction, we can reduce this problem to a slightly easier one. We will map $(G, s, t) \mapsto (G', s)$, where $G' = G$ but with edges added between t and *every* vertex. It follows that

$$\exists \text{ a path } s \rightsquigarrow t \text{ in } G \Leftrightarrow n \text{ vertices are reachable from } s \text{ in } G'.$$

So it suffices to design an "**NL** algorithm" for counting the number of vertices reachable from s. What does it mean to give a nondeterministic algorithm for computing a function (as opposed to deciding a language)? The algorithm may fail on some computation paths, but on *all* non-failing computation paths it must compute the correct answer. (Naturally, we require that on every input, there is always at least one non-failing computation path.) You can verify that if we construct such an algorithm for counting the number of vertices reachable from s in G' using $O(\log n)$ space, we can obtain an **NL** algorithm for UNREACHABILITY.

Now, on to the actual algorithm for counting the number of vertices reachable from s in G. We will use a procedure called inductive counting. Let $S_k = \{$All vertices reachable from s via a path

of length $\leq k\}$. Let $N_k = |S_k|$. Our goal is clearly to determine N_{n-1}. We know that $S_0 = \{s\}$. Thus, given N_k, we will need to inductively compute N_{k+1}.

```
HowManyReachable(G, s) {
.     N_0 = 0;
.     for k = 1, 2, ..., n − 1 {;
.         N_k = 0;
.         for all vertices u {
.             reply = FALSE;
.             count = 0;
.             for all vertices v {
.                 guess path p of length ≤ k + 1 from s;
.                 if p ends at v, count ← count + 1;
.                 if (v, u) ∈ E, reply = TRUE;
.             }
.             if reply = TRUE N_k ← N_k + 1;
.             if reply = FALSE && count < N_{k−1}, then HALT and "FAIL";
.         }
.     }
.     return N_{n−1};
}
```

Note: I highly recommend reading Papadimitrious's explanation of the algorithm as he breaks the code down into four very simple subroutines.

■