Theorem: REACH is complete for **NL**.

Proof:

$$w \in \mathcal{L}(N) \quad \Leftrightarrow \quad \text{CompGraph}(N, w) \in \text{REACH} \quad \spadesuit$$

Space Hierarchy Theorem: Let $f(n) \geq \log n$ be a space constructible function. If

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

Then, $\textbf{DSPACE}[g(n)] \subsetneq \textbf{DSPACE}[f(n)]$.

Proof: Diagonalize against all machines using space $3f(n)$ and time $2^{3f(n)}$. $\spadesuit$

We stated but did not prove the similar *Time Hierarchy Theorem*.

How well can we simulate *nondeterministic* space with *deterministic* space?

We know one way to do it already:

Proposition 18.1

$$\mathbf{NSPACE}[s(n)] \subseteq \mathbf{NTIME}[2^{O(s(n))}] \subseteq \mathbf{DSPACE}[2^{O(s(n))}]$$

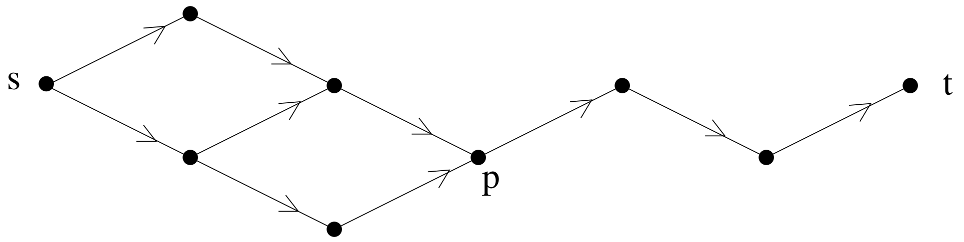
In fact we can do far better, as we'll now show.

We have two "algorithms" now for reachability:

- DFS, etc., which are good for time $(n^{O(1)})$ but bad for space $(O(n))$, and
- Nondeterministic search, which is good for space $(O(\log n))$ but not a real algorithm at all because of the nondeterminism.

How do we attack reachability if space is the main concern?

Theorem 18.2 $\quad \text{REACH} \in \mathbf{DSPACE}[(\log n)^2]$



Proof:

Define $\text{PATH}(x, y, k)$ as "there is a path from vertex x to vertex y of length k or less.

$$G \in \text{REACH} \;\Leftrightarrow\; G \models \text{PATH}(s, t, n)$$

$$\text{PATH}(x, y, 1) \;\equiv\; (x = y) \;\vee\; E(x, y)$$

$$\text{PATH}(x, y, 2d) \;\equiv\; (\exists z)(\text{PATH}(x, z, d) \;\wedge\; \text{PATH}(z, y, d))$$

This gives us a recursive algorithm for PATH. How much space does it use?

We determine the space usage with a recurrence.

$S(n, d)$ = space to check paths of distance d in graphs with n nodes.

$$\begin{aligned} S(n,0) &= O(1) \\ S(n,k) &= O(\log n) + S(n,k/2) \text{ and so,} \end{aligned}$$

$$S(n,n) = O((\log n)^2)$$

♠

This can be thought of as a *middle-first search* algorithm for REACH, efficient for deterministic space but lousy for time.

```
boolean isPath (vertex x,
                vertex y, int dist) {
   if (x == y) return true;
   if (dist == 1) return (edge(x, y));
   else for (vertex u = 0; u < n; u++)
      if (isPath (x, u, dist/2) &&
          isPath (u, y, dist - dist/2))
         return true;
   return false;}
```

The call `path(s,t,n-1)` recurses to a depth of at most $\log n$. Each recursive call needs $O(\log n)$ bits on the stack. But the running time may be as bad as $n^{\log n}$ since there are in effect $\log n$ nested loops.

$$
\begin{aligned}
T(n,0) &= O(1) \\
T(n,k) &= O(n)T(n,k/2) \text{ and so,} \\
T(n,n) &= n^{O(\log n)}
\end{aligned}
$$

Corollary 18.3 (Savitch's Theorem) *For* $s(n) \geq \log n$,

$\mathbf{DSPACE}[s(n)] \;\subseteq\; \mathbf{NSPACE}[s(n)] \;\subseteq\; \mathbf{DSPACE}[(s(n))^2]$

Proof: Let $A \in \mathbf{NSPACE}[s(n)]$; $\quad A = \mathcal{L}(N)$

$$w \in A \qquad \Leftrightarrow \qquad \text{CompGraph}(N, w) \in \text{REACH}$$

$$|w| = n; \qquad |\text{CompGraph}(N, w)| = 2^{O(s(n))}$$

Testing if $\text{CompGraph}(N, w) \in \text{REACH}$ takes space,

$$\begin{aligned}(\log(|\text{CompGraph}(N, w)|))^2 \;&=\; (\log(2^{O(s(n))}))^2 \\ &=\; O((s(n))^2)\end{aligned}$$

From w build $\text{CompGraph}(N, w)$ in **DSPACE**[s(n)]. ♠

Corollary 18.4 PSPACE $=$ NPSPACE.

Along with regular languages and CFL's, there is another old-time complexity class called the *context-sensitive languages* that turns out to be **NSPACE**(n). It was asked whether this class is closed under complement (like the regular languages) or not (like the CFL's). Since the general intuition was that it was *not*, no one looked for a proof that it *was*. The problem remained open for about twenty years.

In 1987 two researchers, Neil Immerman in the US and Richard Szelepcsényi in Slovakia, simultaneously found a proof that nondeterministic space classes *are* closed under complement. Not only that, the proof is easy to present.

The reason for the simultaneity is probably that a series of results just before this began to suggest that the result might be true. Neil reports that he got the basic idea of the proof while walking his dog.

Theorem 18.5

$$\overline{\text{REACH}} \in \mathbf{NL}$$

Proof:

Fix the graph G.

$$N_d \quad = \quad |\{v \mid v \text{ reachable from } s \text{ using} \leq d \text{ edges}\}|$$

Claim: The following predicates are each in **NL**:

1. DIST(x, d): distance$(s, x) \leq d$
2. NDIST$(x, d; m)$: if $m = N_d$ then $\neg$DIST(x, d)

Proof:

1. Guess the path of length $\leq d$ from s to x.
2. Guess m distinct vertices $v_1, \ldots, v_m$, with each $v_i \neq x$, and guess paths showing DIST(v_i, d) for each i.

♠

Claim: We can "compute" N_d in **NL**.

What does "compute" mean? We define a nondeterministic procedure that either (a) outputs the correct value of N_d, or (b) fails to output anything. There must be at least one path on which (a) occurs.

Proof: By induction on d.

Base case: $N_0 = 1$, whatever the graph.

Inductive step: Suppose we are on a path that has reached a value of N_d, which by the IH is correct. The following pseudo-Java code returns the correct value of N_{d+1} if it makes the right guesses, and rejects otherwise:

```
int c = 0;
for (int v=0; v < n; v++) {
   if (guessBoolean())
      if (DIST(v,d+1)) c++;
      else reject;
   else // verify !DIST(v,d+1)
      for (int z=0; z < n; z++) {
         if (NDIST(z,d,Nd)) break;
         if ((z!=v) && (!edge(z,v)))
            break;
         reject;}}
return c;
```

The "right guesses" are `true` for v's that are within distance $d+1$ of s and `false` for the others. Correct guesses can be verified and wrong guesses cause the whole procedure to reject.

$$G \in \overline{\text{REACH}} \quad \Leftrightarrow \quad \text{NDIST}(t, n; N_n) \qquad \spadesuit$$

$\spadesuit$

Corollary 18.6 (Immerman-Szelepcsényi Theorem) *Let $s(n) \geq \log n$. Then,*

$$\mathbf{NSPACE}[s(n)] \quad = \quad co\text{-}\mathbf{NSPACE}[s(n)]$$

Proof: Let $A \in \mathbf{NSPACE}[s(n)]$; $\quad A = \mathcal{L}(N)$

$$w \in A \qquad \Leftrightarrow \qquad \text{CompGraph}(N, w) \in \text{REACH}$$

$$|w| = n; \qquad |\text{CompGraph}(N, w)| = 2^{O(s(n))}$$

Testing whether $\text{CompGraph}(N, w)$ is in $\overline{\text{REACH}}$ with a nondeterministic procedure takes space

$$\begin{aligned} \log(|\text{CompGraph}(N, w)|) &= \log(2^{O(s(n))}) \\ &= O(s(n)) \end{aligned}$$

♠

Definition 18.7 We say that S is *reducible* to T, $S \leq T$, iff $\exists\ f \in F(\mathbf{L})$ such that,

$$(\forall w \in \mathbf{N}) \quad (w \in S) \qquad \Leftrightarrow \qquad (f(w) \in T)$$

♠

$$A_{0,17} \quad = \quad \{n \mid M_n(0) = 17\}$$

Claim: $K \leq A_{0,17}$.

Proof: Define $f(n)$ as follows:

$$M_{f(n)} \;=\; \boxed{\begin{array}{c}\text{erase input}\\ \text{write } n\end{array}}\ \boxed{M_n}\ \boxed{\begin{array}{c}\text{if 1 then write 17}\\ \text{else loop}\end{array}}$$

$$n \in K \quad \Leftrightarrow \quad M_n(n) = 1 \quad \Leftrightarrow \quad M_{f(n)}(0) = 17$$

♠

Why is this reduction in $F(\mathbf{L})$?

Given a number n, we need to write code for a TM that prints n, runs M_n on it, and then either writes "17" or loops. We are limited to space that is logarithmic in the length of the binary number n.

But the "print n" part is going to be mostly a copy of the binary of n, and we have assumed that the code of M_n is easily obtainable from the binary for n. (For example, we could say that the binary either *is* the code for M_n in ASCII, or n isn't the number of a TM.) The last part is only $O(1)$ bits of TM code.

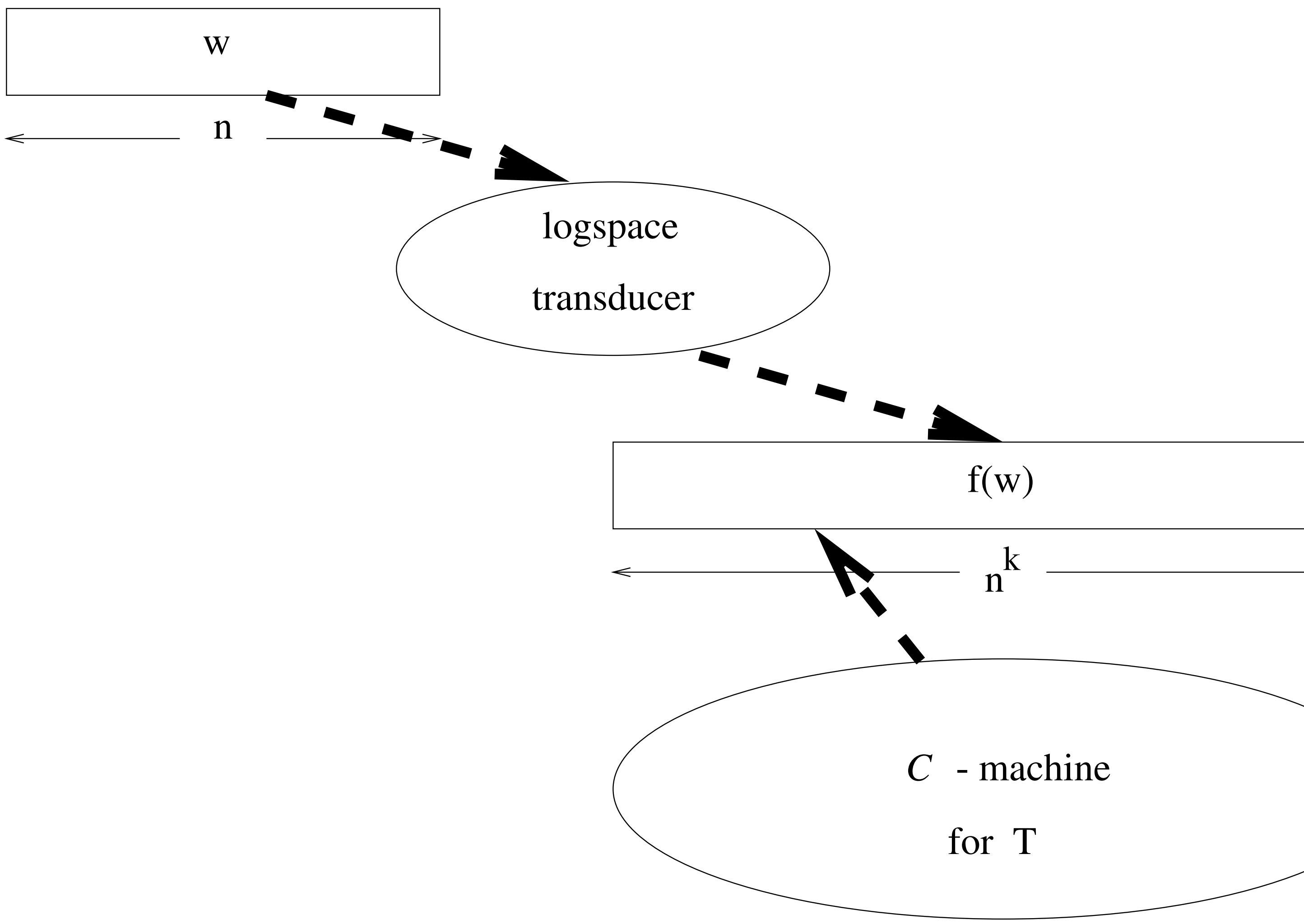
Theorem 18.8 *Let $\mathcal{C}$ be one of the following complexity classes:* L, NL, P, NP, co-NP, PSPACE, EXPTIME, Primitive-Recursive, RECURSIVE, r.e., co-r.e.

$$\textit{Suppose } S \leq T.$$
$$\textit{If } T \in \mathcal{C} \quad \textit{Then} \quad S \in \mathcal{C}$$

That is, each of these classes is closed under reductions.

Proof: Suppose that $S \leq T$ and $T \in \mathcal{C}$.

We build a $\mathcal{C}$ machine for S: $\quad w \in S \quad \Leftrightarrow \quad f(w) \in T$



There's actually a problem with this proof in the case where $\mathcal{C}$ is **L** or **NL**. It's the same issue that came up in the extra credit of HW#2 where we wanted to show that TIMES is in $F(\mathbf{L})$.

A Nontrivial Fact About Logspace Reductions:

If $A \leq B$ and $B \leq C$, then $A \leq C$.

It looks obvious at first, but draw the picture of the two machines! The output tape of the first machine becomes the input tape of the second. In the two original machines neither counts against the space bound, but in the new machine *this becomes a worktape*!

Similarly the former output tape of our logspace transducer is now a worktape, and we *a priori* need a worktape to store the n by n array of partial product bits when computing TIMES.

Proving Transitivity:

Say we know that f reduces A to B and that g reduces B to C.

Clearly the reduction we want from A to C is h, where $h(w) = g(f(w))$. Then $w \in A$ iff $h(w) \in C$. The only problem is to show that h is in $F(\mathbf{L})$.

Here's how to compute $h(w)$, when w is the length-n string on the read-only input tape. Let x be $f(w)$ and start computing $g(x) = h(w)$. When your computation needs a bit x_i of x, however, *suspend* that computation until you have calculated x_i from w, and then continue.

At any given time you have the computation of $g(x)$ from x going, which takes $O(\log |x|) = O(\log n)$ space. You also may be temporarily computing a bit of x from w, which you get by carrying out the computation of $f(w)$ (using $O(\log n)$ space) until x_i is output, at which time you take it and continue with the main computation. (This is a bit time-intensive, but we don't care because we're concerned only with space.)

The total space usage is $O(\log n)$.

Reductions are Useful for:

Lower Bounds:

If A is hard and $A \leq B$ then we may conclude that B is hard.

(**Example:** B is defined to be **NP**-hard if for any $C \in$ **NP**, we have that $C \leq B$. If A is **NP**-hard and $A \leq B$, B is seen to be **NP**-hard by the transitivity of $\leq$.)

Upper Bounds:

If B is easy and $A \leq B$ then we may conclude that A is easy.

(**Example:** Define "easy" to mean "a member of $\mathcal{C}$" for any of the classes in the previous theorem.)