18.405J/6.841J Advanced Complexity Theory	February 4, 1999
Lecture 2	
Lecturer: Daniel A. Spielman	*Scribe: Antonio Ramírez*

1 The Immerman-Szelepcsényi Theorem

Most of this lecture will be devoted to the proof of this theorem, which is stated as follows.

Theorem 1 (Immerman-Szelepcsényi) *For $f(n) \geq \log n$,*

$$\mathrm{NSPACE}(f(n)) = \mathrm{coNSPACE}(f(n)).$$

A consequence of this theorem is that, in some sense, time complexity is more interesting than space complexity, as it is not known whether NP = coNP.

Our proof will use two auxiliary results:

Lemma 2 *Given a digraph G with n nodes, and a node x of G, it is possible to compute the number of nodes reachable from x in* $\mathrm{NSPACE}(\log n)$.

Lemma 3 *Given two nodes x, y of a digraph G,* and *the number of nodes k reachable from x, there is a logspace NTM*[1] *that* rejects *if and only if y is reachable from x.*

1.1 Applying the lemmas

We will apply these results as follows. Consider M, a $\mathrm{coNSPACE}(f(n))$ machine. A configuration of M consists of the contents of the tape (up to the maximum $f(n)$ cells that M is allowed to use at any time), the position of the head, and the finite state. Form a graph G with all the configurations of M as nodes, with a directed edge from config_1 to config_2 if M can go from config_1 to config_2 in one step. We would like to show that there is an $\mathrm{NSPACE}(f(n))$ machine that decides the same language

[1] Nondeterministic Turing machine.

as M. Recall that a co-nondeterministic machine accepts an input if and only if all computations with that input end in an accept state. Reinterpreting this condition using G, M accepts if and only if there is *no* path from the starting node to a node denoting a reject state.

Assume without loss of generality that G has exactly one rejecting configuration. Let x be the starting configuration of M, and y the rejecting configuration. Consider a NTM M' that does the following:

1. Compute the number of nodes of G that are reachable from x.

2. Using this number, accept if y is not reachable from y.

Because of Lemmas 2 and 3, M' works in NSPACE($\log n$), where n is the number of nodes in G. But G has $O(c^{f(n)})$ nodes, where c is the alphabet size: each one of the $f(n)$ possible cells can take one of c values (the other components of a configuration are not significant asymptotically). Hence, M' works in NSPACE($\log O(c^{f(n)})$) = NSPACE($f(n)$), as desired.

This shows that coNSPACE($f(n)$) $\subseteq$ NSPACE($f(n)$). But this implies the reverse inclusion too, since co(coNSPACE) = NSPACE. Then Theorem 1 follows from the lemmas.

1.2 Computing functions nondeterministically

There is a subtle point in Lemma 2: we know what it means for an NTM to accept a language, but it is not immediately clear what it means for an NTM to *compute* a function.

Definition 4 *An NTM M is said to compute a function f on input x if M has at least one accepting computation on input x and, for all accepting computations of M on x, $f(x)$ is left on the tape when done.*

To convince ourselves that this definition is the correct one, consider another plausible definition: M computes $f(x)$ if at least one branch accepts with the right answer. This definition is flawed, however: nondeterminism allows M to compute *every* possible answer, one of which must be the correct one. This is absurd, since it would mean that M computes every function.

1.3 Proving the lemmas

Proof of Lemma 3 Determining whether a node v is reachable from x by *accepting* can be done in NSPACE($\log n$). The idea is to nondeterministically guess a sequence of edges to follow to reach v from x:

Algorithm:

```
    u ← x
    for j = 1...n
        u ← choose from {u} ∪ {neighbors of u}
    if u = v then accept else reject
```

At each step, the algorithm nondeterministically follows an edge or stays at the same node, and after n steps it checks to see if v has been reached. At all times, it only stores the name of a node and the counter, which take space $O(\log n)$.

We want an NTM that *rejects* when there is such a path, knowing the number k of nodes reachable from x. The main idea for the proof is this: If there exist k distinct nodes, all reachable from x and each different from y, then y is not reachable from x. This fact is used by the following algorithm:

Algorithm:

```
    i ← 0
    for all nodes v of G
        nondeterministically goto A or B
A:      if v is reachable from x
            if v = y
                then reject
                else i ← i + 1
B:  end for
    if i < k then reject else accept
```

On each iteration the algorithm is allowed to either ignore the vertex or to test it for connectedness from x, using the first algorithm[2]. If it is connected from x, a counter is incremented. At the end, it accepts only if at least k distinct nodes connected to x were found.

This algorithm proves the lemma. ■

[2]It is possible to call nondeterministic "subroutines," as long as we have no actions that are conditional on the subroutine rejecting–if the subroutine rejects, the whole machine rejects on that execution.

Proof of Lemma 2 Let S_i be the set of nodes reachable from x in at most i steps. The plan is to compute $|S_k|$ using $|S_{k-1}|$. If we succeed in doing this, the answer will be $|S_n|$. Clearly, $|S_0| = 1$. The idea of the algorithm is to go through all nodes, and for each try to test if it is in S_k. A node is in S_k if and only if it is a neighbor of a node in S_{k-1}.

What we would like to do is then:

Algorithm:

```
    l ← 0
    for all nodes w in G
        if w is a neighbor of a node in S_{k-1} then l ← l + 1
    end for
    |S_k| ← l
```

We must figure out how to determine in NSPACE$(\log n)$ if a node is a neighbor of a node in S_{k-1}, knowing only $|S_{k-1}|$. This can be done as follows.

Algorithm:

```
    l ← 0
    for all nodes w in G
        /* if w is a neighbor of a node in S_{k-1}, increment l: */
        i ← 0
        flag ← false
        for all nodes v in G
            nondeterministically goto A or B
A:          if v is reachable from x in at most k − 1 steps then
                if w is a neighbor of v then flag ← true
                i ← i + 1
B:          end if
        end for v
        if i < |S_{k-1}| then reject
        if flag = true then l ← l + 1
    end for w
    |S_k| ← l
```

The inner loop looks at a set of nodes, ignoring those that are not in S_{k-1}, and it sets the flag to true if a node in S_{k-1} is adjacent to w. At the same time, it counts how many nodes in S_{k-1} it saw in total. By the end of this loop, if the flag is set, it means that w is adjacent to some node of S_{k-1}, and hence in S_k. In this case, l is incremented, provided that the inner loop saw all the nodes in S_{k-1}.

More formally, we need to prove that:

(i) for a given w, l is incremented only if $w \in S_k$, and

(ii) for a given $w \in S_k$, l is incremented if the computation does not reject.

(i) is clear: for l to be incremented, $flag$ must have been set to **true**, which happens only if a node v is seen that is both a neighbor of w and contained in S_{k-1}. This means that $w \in S_k$.

To show (ii), consider some $w \in S_k$. Note that the non-rejecting executions of the inner loop are those in which the A branch is taken only for nodes in S_{k-1}. If not all of S_{k-1} is hit, then we reject after verifying that $i < |S_{k-i}|$. Otherwise, we know that some node among the ones seen must be adjacent to w, so the flag is set, and both conditions for incrementing l are met.

Note that the execution that always takes branch A does *not* accept: if the inner loop sees a node that is not reachable from x in at most $k-1$ steps, the whole computation rejects, because the subroutine that computes this rejects.

Again, it is easily seen that this algorithm uses only logarithmic space. This concludes the proof. ■

2 Introduction to Alternation

Alternating Turing machines (ATMs) are a generalization of both nondeterministic and co-nondeterministic TMs. NTMs accept if at least one computation accepts; co-NTMs accept if all computations accept. ATMs mix both possibilities: they have the power of deciding at a nondeterministic branch state whether to accept if all branches accept, or if at least one accepts.

Definition 5 *An alternating Turing machine is a Turing machine in which each non-halting state is labeled with either $\exists$ or $\forall$, and so that any state has at most two transitions originating from it. $\exists$-states are called existential, $\forall$-states are called universal.*

The limitation of two transitions from each node is for practical purposes, and does not affect the power of ATMs.

To determine whether or not an ATM M accepts an input x, consider its computation tree, which has a node for each possible state in an execution and a directed edge for each transition (Figure 1). The root of the tree will be the starting state, and the leaves will be accepting or rejecting states. Working from bottom to top, label each node with “accept” or “reject”: an ∃-state is labelled as “accept” if at least one of its children is accepting; a ∀-state is labelled as “accept” if both of its children accept. The outcome of the computation is the label of the root node.

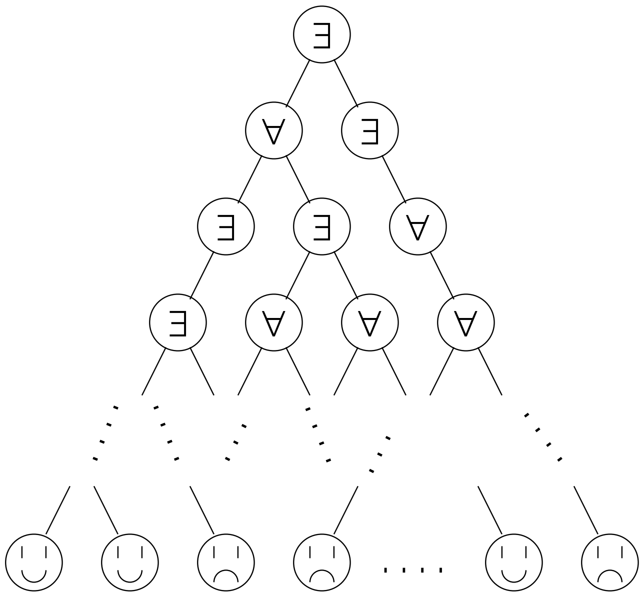

Figure 1: The computation tree of an ATM.

2.1 Alternating complexity classes

The ATM model determines several complexity classes.

Definition 6

$$\begin{aligned}
\mathrm{ATIME}(f(n)) &= \{L \mid L \textit{ is a language decidable in time } O(f(n)) \textit{by an ATM.}\} \\
\mathrm{ASPACE}(f(n)) &= \{L \mid L \textit{ is a language decidable in space } O(f(n)) \textit{by an ATM.}\} \\
\mathrm{AP} &= \bigcup_{i=0}^{\infty} \mathrm{ATIME}(n^k) \\
\mathrm{AL} &= \mathrm{ASPACE}(\log n)
\end{aligned}$$

Some known relations involving these new complexity classes are:

$$\mathrm{NP} \subseteq \mathrm{AP} \quad (1)$$
$$\mathrm{coNP} \subseteq \mathrm{AP} \quad (2)$$

$$\begin{aligned} \text{PSPACE} &= \text{AP} \quad (3) \\ \text{AL} &= \text{P} \quad (4) \end{aligned}$$

The first two relations are straightforward: NTMs and co-NTMs are special cases of ATMs. We will prove the last two in future lectures, but one of the directions of (3) is easy: the PSPACE-complete language TQBF (totally quantified boolean formulas) is easily seen to be in AP; hence PSPACE $\subseteq$ AP.

A more interesting problem in AP is

$$\begin{aligned} \text{MIN-CIRCUIT} \;=\; &\{\text{boolean circuits } c \mid \text{no circuit smaller than } c \\ &\text{computes the same function as c.}\} \end{aligned}$$

The condition for a circuit to be minimal can be written as follows:

$$\text{MIN-CIRCUIT} = \{\text{boolean circuits } c \mid \forall \text{ circuit } c' \text{ where } |c'| < |c|, \exists x : c'(x) \neq c(x)\}$$

Expressing the condition using quantifiers makes it easy to see how it might be solved in AP. This example is interesting because MIN-CIRCUIT is not known to be in NP.