18.405J/6.841J Advanced Complexity Theory	February 5, 1998
Lecture 2	
Lecturer: Daniel A. Spielman	*Scribe: Raj D. Iyer*

In today's lecture I will say more about space and space complexity. Then I will cover alternation, which will lead us to the polynomial hierarchy.

1 Space Complexity

1.1 Definitions and Theorems

In the last class we defined time and time complexity, as well as space and space complexity. Then, using nondeterministic Turing machines (NTM), we defined nondeterministic time and its complexity. Nondeterministic space is defined analogously:

$$\text{NSPACE}(f(n)) = \{ \text{ all languages } A \mid \exists \text{ a NTM that accepts } A \text{ and uses } O(f(n)) \text{ space on all branches } \}$$

Recall that a NTM goes through many branches of the computation and accepts its input if any one of the branches accepts. The first interesting complexity class we come to is nondeterministic log space

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

The next interesting class is

$$\text{NPSPACE} = \bigcup_k \text{NSPACE}\left(n^k\right).$$

Languages in this class can be decided by a NTM using a polynomial amount of space.

Space complexity is a bit weird and we don't get the same things that we do with time complexity. The first thing we see is Savitch's Theorem, with which you should already be familiar:

Theorem 1 (Savitch) $\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n))$.

The proof is relatively simple, and you will see the idea of the proof in today's lecture, although we will not prove it directly. The other critical theorem in space complexity is:

Theorem 2 $\text{NSPACE}(f(n)) = \text{coNSPACE}(f(n))$ for $f(n) \geq \log(n)$.

So you see, things are much less exciting than TIME and coTIME, which we don't know to be equal (we suspect they are unequal!). Theorem 2 was proven independently by N. Immerman and R. Szelepcsényi. These theorems are good to know, and they're the reason why we won't consider much of space complexity in this course.

Since I introduced these complexity classes, I should give you at least one example of a complete problem. I'll give a problem for NL which is complete under log space reductions[1]. Another purpose of this example is show that you can compute something within log space.

[1] A log space reduction is performed by a *log space transducer*, a TM with a read-only input tape, a write-only output tape, and a $O(\log n)$ read/write work tape. See references for more details.

1.2 s-t CONNECTIVITY is NL-complete

The classic NL-complete problem is s-t CONNECTIVITY, abbreviated STCONN.

Problem:	STCONN
Input:	a directed graph with two distinguished nodes: source s and sink t
Question:	Is there a path from s to t?

The language of STCONN is the set of graphs with an s-t path. Of course, a graph need not have an s-t path. Let's first prove STCONN $\in$ NL. Think about how the input graph is represented. For each node we are given a list of the nodes connected to it by an edge. These neighbor lists are concatenated to form a list of lists, indexed by node. Since there are n nodes, each node can be represented by a number in $\log n$ bits. The nondeterministic log space algorithm is remarkably simple. We begin with the source node s and nondeterministically select one of its neighbors. We then choose one of the neighbor's neighbors, and so on. In this way we don't know where we've been; we only know where we're at. If we ever reach t, we accept.

This is a fine idea, but unfortunately the algorithm may not halt. We can fix this by keeping a timer and ensuring that we choose a sequence of at most n nodes. Why is this sufficient? Since there are n nodes total, the longest possible path, which does not contain a cycle, is of length n. Thus if there is a path from s to t, we must be able to traverse it in at most n steps. We will nondeterministically find such a path if and only if it exists. The timer counts up to n and thus requires $\log n$ space. In summary, the algorithm is

1. Initialize counter $c = 0$.
2. Begin with the source node s.
3. Nondeterministically select one of the node's neighbors.
4. Increment c. If $c = n$, reject.
5. If new node is t, accept, else repeat step 3 with new node.

Now let's show that STCONN is NL-complete. We need to show $\forall$ $A \in$ NL such that A is accepted by TM M, $\exists$ a logarithmic space machine N such that for all inputs x,

$$x \in A \Longleftrightarrow N(x) \in \text{STCONN}.$$

That is, N takes an input x for a language A in NL and outputs a graph[2] $N(x)$ with distinguished nodes s and t.

The idea is to describe the computation of M on x by a graph G. Each configuration[3] of M on x corresponds to a node in G. If one configuration yields another in one step, we connect their corresponding nodes by a directed edge. The starting configuration (M at start state with x on its tape) is distinguished as the source s. We can assume without loss of generality that there is a unique accept state[4] which corresponds to the sink t. A path between s and t is a legal sequence of configurations of M on x which begins with the start configuration and ends with the accept configuration. In other words, there exists a path between s and t iff M accepts x.

[2]Note that N computes a function rather than recognizing a language. N can either take a description of M as input and thus work for all problems in NL, or else N can be constructed for each machine M.

[3]A configuration of a TM is the current state, tape contents, and tape symbol under the head.

[4]After M accepts, we can make it erase its tape and move its head to the leftmost tape cell.

Having constructed $\{G, s, t\} = N(x)$, we can feed $N(x)$ to our algorithm for STCONN and accept x iff STCONN accepts $N(x)$. It remains to show that this reduction N uses log space. This is a simple trick. N has to output the graph $N(x)$, so it enumerates over all configurations of M. That is, N iterates over all possible strings C made of the symbols which can appear in a configuration of M. The number of such strings is polynomial[5] in n. For each C (which may or may not be a legal configuration of M), N inspects the description of M and writes on the output tape[6] all the configurations of M that can be reached from C in one (nondeterministic) step. These are the nodes pointed at by C. Since there are $poly(n)$ strings C, and for each C we generate at most $poly(n)$ configurations, the space needed for a counter to execute the loop is $O(\log(poly(n))) = O(\log n)$.

[5]A configuration of a log space transducer with a work tape of size ℓ is a string of $\ell+1$ symbols consisting of one state symbol from a set Q and ℓ tape symbols (including input symbols) from a set Γ. A configuration represents the contents of the work tape and the current state q. Also, the current symbol under the work head is the symbol immediately to the right of the q (thus q can appear at any point in the string). Finally, a count (between 1 and n) at one end of the string indicates the position of the read head on the input tape. The total number of such strings is $|Q|\ell n|\Gamma|^{\ell} = \exp(\ell)$. Since $\ell = O(\log n)$, this is $poly(n)$.

[6]Recall that N's output tape is unbounded. However, since N only has $poly(n)$ possible configuations, its output can use at most $poly(n)$ space (see footnote 1).

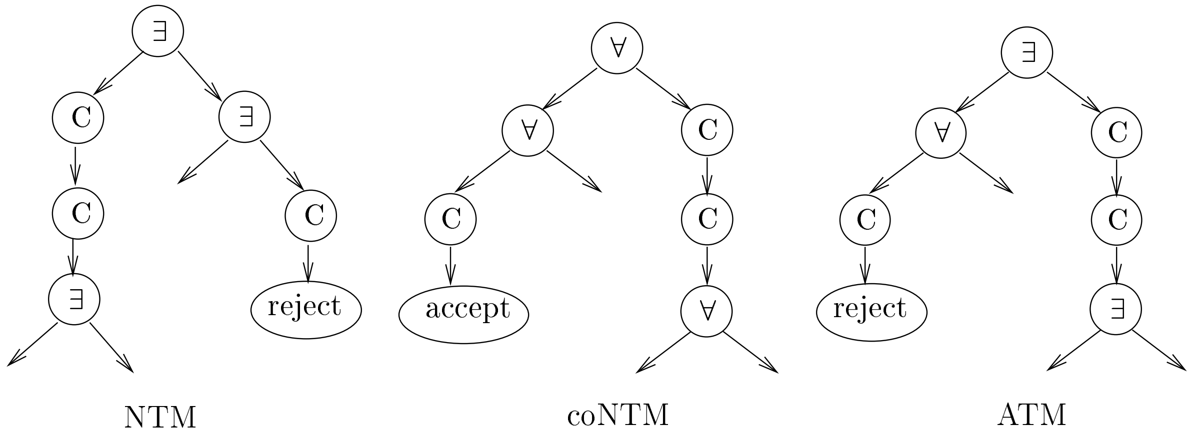

Figure 1: NTM, coNTM, and ATM computation trees.

2 Alternation

2.1 Definitions

The idea of alternation is to combine a nondeterministic machine (NTM) and a co-nondeterministic machine (coNTM). Imagine the computation of a NTM M on input x as a tree consisting of three types of nodes: configurations of M, accept/reject nodes, and *existential* ($\exists$) nodes. An $\exists$ node has two children, representing a branch in the computation. A configuration node (indicated by a C in Figure 1) has one child which can be a configuration reachable in one step, an $\exists$ node, or an accept or reject node. The accept and reject nodes are the leaves of the tree. The root is an $\exists$ node. The idea is that if there exists a path from the root to an accept state, then the machine accepts.

Similarly, a coNTM has *universal* ($\forall$) nodes instead of $\exists$ nodes, and the root is a $\forall$ node. It accepts if all leaves are accept nodes, that is, all paths are accepting (Figure 1).

Now we want to create machines that have both $\exists$ and $\forall$ nodes. We could give a detailed definition of the tree, or we could define "accept" precisely. Let's do the latter.

An *alternating Turing machine* (ATM) is a tree with $\exists$, $\forall$, configuration, accept, and reject nodes (Figure 1). To determine acceptance, work backwards in the tree from the leaves up, marking all nodes in the tree as *accept* or *reject*, and accept if and only if the root is marked *accept*. The marking rules are:

- an accept node is marked *accept*.
- a reject node is marked *reject*.
- a configuration node is marked the same as its child.
- an $\exists$ node is marked *accept* iff either of its children is marked *accept*.
- a $\forall$ node is marked *accept* iff both of its children are marked *accept*.

As we have for other TMs, we define the following complexity classes for ATMs:

$$\text{ATIME}(f(n)) = \{ \text{ all languages } A \mid \exists \text{ an ATM that accepts } A \text{ and uses } O(f(n)) \text{ time on all branches } \}$$

$$\text{ASPACE}(f(n)) = \{ \text{ all languages } A \mid \exists \text{ an ATM that accepts } A \text{ and uses } O(f(n)) \text{ space on all branches } \}$$

and

- $\text{AL} = \text{ASPACE}(\log n)$
- $\text{AP} = \bigcup_k \text{ATIME}\left(n^k\right)$
- $\text{APSPACE} = \bigcup_k \text{ASPACE}\left(n^k\right)$.

2.2 A language in AP

Here is a language of boolean circuits in AP.

$$\text{MIN-CIRCUIT}(m) = \{ \text{ circuit } c \text{ with } m \text{ inputs} \mid \exists \text{ no smaller circuit with } m \text{ inputs that computes the same function } \}.$$

Let's write this using quantifiers: "$\forall$ circuits c' with m inputs that are smaller[7] than c, $\exists$ input x such that $c(x) \neq c'(x)$." Consider an ATM M for this language on a particular input. If it marks one child of an $\exists$ node as *accept*, and if it marks all $\forall$ nodes as accept, then it accepts the input. The $\forall$ node chooses all smaller circuits, and the $\exists$ node chooses over all inputs of size m. Since c' is smaller than c, M can write down c' and evaluate it on an input in linear time (in the size of c).

Let

$$\text{MIN-CIRCUIT} = \bigcup_{k \geq 1} \text{MIN-CIRCUIT}(k)$$

We have shown that MIN-CIRCUIT $\in$ AP. We do not know if MIN-CIRCUIT is in NP or coNP, but we suspect it is in neither. We don't think it's complete for AP.

2.3 Relation of alternating and deterministic time and space

If you're all happy, then I'm going to start proving facts about ATMs. I will prove general facts about the machines and then get to the specifics of the polynomial hierarchy.

The classes AL, AP, and APSPACE turn out to be classes we've already seen.

Theorem 3 $\text{ATIME}(f(n)) \subseteq \text{SPACE}(f(n)) \subseteq \text{ATIME}\left(f^2(n)\right)$.

This implies that AP = PSPACE.

Theorem 4 $\text{ASPACE}(f(n)) = \text{TIME}\left(2^{O(f(n))}\right)$.

This implies that APSPACE = EXP.

Let's begin proving these theorems. We have four statements to prove. All the proofs involve following a computation tree in one way or another, but there are different ways of doing it.

[7] The size of a circuit can be measured as the number of gates, the number of wires, the depth, etc.

2.3.1 $\mathbf{ATIME}(f(n)) \subseteq \mathbf{SPACE}(f(n))$

We have the computation tree for the $f(n)$ time bounded ATM M on an input x. We will do a depth first search (DFS) on the tree in $f(n)$ space. Begin at the root. For each node u, if u is an $\exists$ node, go to the left child and determine (recursively) if it is marked *accept*. If so, mark u as *accept* and return. If not, determine if the right child is marked *accept*. If so, mark u as *accept* and return, if not, mark u as *reject* and return. Do the same thing if u is a $\forall$ node, but mark u as *accept* iff both children are marked *accept*.

To do this DFS we need to push each node onto a stack. This is just like the accept condition for ATMs: write down the list of all nodes and move up the tree, marking nodes as *accept* or *reject*. However, how much space does it take to do this? We write down all the nodes in the path so far. The largest nodes (in description length) are configuration nodes, each of which can be stored in $O(f(n))$ space: a configuration consists of M's state and tape contents, and M cannot use more than $O(f(n))$ tape since it must run in $O(f(n))$ time. The tree has depth at most $O(f(n))$ (by definition of M), so there are at most $O(f(n))$ nodes on the stack. So the total space usage is $O(f^2(n))$.

Ah, but we can only use $O(f(n))$ space! Let's improve the algorithm. We don't need to store the entire node description on the stack. We only need to write down which child was chosen, the left or right, as well as the description of the current node. On a recursive call to a child, we push a symbol L or R to indicate the position of the child relative to its parent. When we return from a call, we need to obtain the description of the node at the top of the stack. We do this by starting at the root and reading the stack bottom up, traversing the tree according to the path specified by the stack symbols. The node that the path ends on is the current node. We then pop its symbol (L or R). Thus we use constant space for each stack item, in addition to the $O(f(n))$ space needed to store the current node description. This proves the theorem.

2.3.2 $\mathbf{SPACE}(f(n)) \subseteq \mathbf{ATIME}\left(f^2(n)\right)$

This proof is modeled after the proof that STCONN $\in$ NL. We make a graph over the configurations of the $f(n)$ space bounded machine N. Each configuration corresponds to a node. Since N is deterministic, each node has at most one outgoing edge. So the graph is a path, possibly containing cycles, beginning at the start configuration and possibly terminating in the unique[8] accept configuration. However, N doesn't have enough time to construct this graph.

Editor's note: At this point the class ended, leaving the proofs unfinished. This proof uses the same idea as the proof of Savitch's theorem, as promised at the beginning of lecture.

3 References

The scribe had previously covered this material in Mike Sipser's book, although the book was not consulted during preparation of these notes.

The material on space complexity, including definitions of log space reductions and configurations of a Turing machine can be found in

[8]See footnote 4.

- Michael Sipser, *Introduction to the Theory of Computation*, PWS Publishing Company, Boston, 1997, pp. 297-302.

- Christos Papadimitriou, *Computational Complexity*, Addison-Wesley, Reading, Massachusetts, 1994, pp. 149-153, 159-160.

Essentially all the material on alternation can be found in Sipser, pp. 348-353. Papadimitriou, pp. 399-401, contains definitions of ATMs, their complexity classes, and a different proof of Theorem 4.