18.405J/6.841J Advanced Complexity Theory	February 8, 2001
Lecture 2	
Lecturer: Daniel A. Spielman	*Scribe: Yu Chen*

1 Alternation

In this lecture we will talk about different kinds of Non-deterministic Turing Machines (NTM) and then how to construct Alternating Turing Machines from them. We'll also examine the relation of ATMs to deterministic classes, then we will prove a few different relations between alternating and deterministic time and space

1.1 Nondeterminstic Turing Machines:

There are two different definitions for NP. One is the short definition by verification, which says that all languages in NP are easily verifiable in polynomial time, given a short (polynomially bounded length) certificate.

$$L \in NP \leftrightarrow \begin{cases} \exists A \in P, p(.) \in POLY \text{ such that} \\ x \in L \leftrightarrow \exists w : |w| \leq p(|x|) \text{ and } (x, w) \in A \end{cases}$$

The second definitions says that $L \in NP$ iff there is some poly-time non-deterministic TM which accepts L. The simplest way to think about a non-deterministic Turing Machine is to consider a normal TM, but it contains a series of Existential ($\exists$) states. When the machine enters such a state, it may continue its computation in one of two different ways, which the machine can represent as a 1 or a 0 on the tape head's current position. In this way, we can think of each node in the configuration tree as a different state of the machine. The root, or START configuration is the initial state; nodes with two children are Existential states; nodes with one child will are regular computational states, and leaves correspond to final accept or reject states. Graphically, we can label each node with an accept or reject; if any leaf terminates at an accept state, the labeling travels up to the root.

(Figure 1) on the next page displays a computation tree of a nondeterministic Turing Machine. When all computational branches terminate, the output of the machine is the logical OR of all leaves. That is to say, the machine accepts if one or more of its brances ends in an accept state.

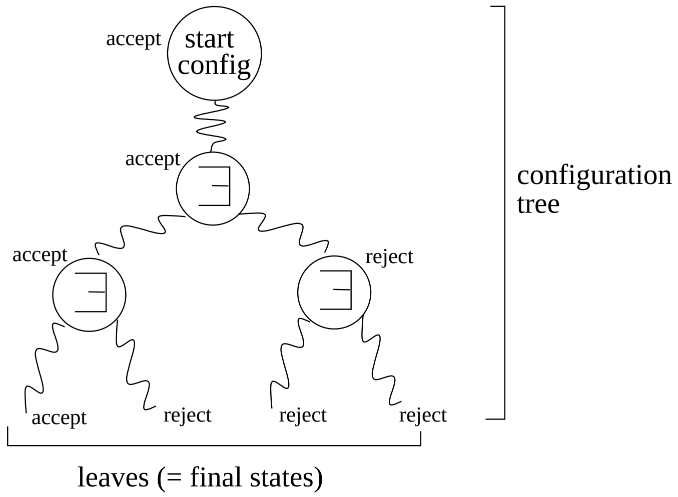

Figure 1: The computation tree of an NTM

1.2 Co-Nondeterministic Turing Machines:

The converse of a Turing Machine with Existential states is one with Universal ($\forall$) states. At each node, the machine can be thought of as spinning off two parallel actions and taking BOTH paths at the same time. The machine accepts if all of its branches end in the accept state. This computation tree would look the same as that of (Figure 1), except with $\forall$ replacing the $\exists$.

1.3 Alternating Turing Machine with $\exists$ and $\forall$ states

We can now consider machines with both Existential and Universal states. They are called Alternating Turing Machines (ATM). We can turn this computation tree into an equivalent circuit by changing accept and reject states into 1 and 0, respectively. $\exists$ and $\forall$ states are changed to OR and AND boolean operators respectively. Figure 2 illustrates. Note : this definition requires all that branches halt.

There are several natural complexity classes related to ATM's.

Definition 1

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

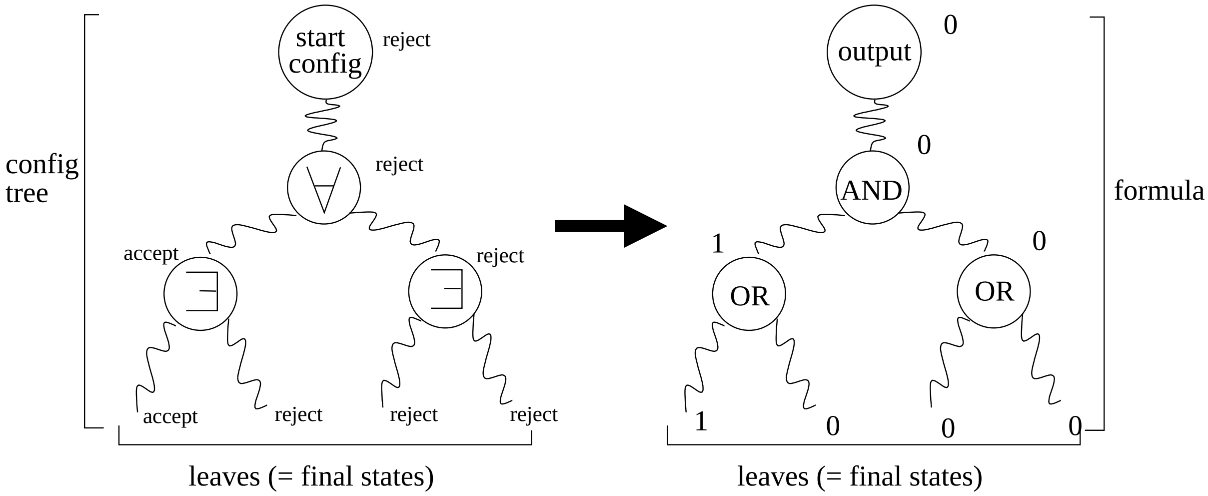

Figure 2: The computation tree of an ATM and corresponding circuit configuration

For example, TQBF fits into AP, so we get that PSPACE $\subseteq$ AP.

The Minimum-Circuit problem is an interesting in AP.

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

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.

2 Relations to deterministic classes

Theorem 2 *For* $f(n) \geq n$ *we have* ATIME(f) $\subseteq$ SPACE(f) $\subseteq$ ATIME(f^2)

Theorem 3 *For* $f(n) \geq \log n$ *we have* ASPACE(f) $=$ TIME($2^{O(f)}$)

Proof : We begin with the easier assersion, SPACE(f) $\subseteq$ ATIME(f^2).

Let L $\in$ SPACE(f), M be the TM that accepts L in SPACE(f). We create a directed graph on the configurations of M. Each node is a configuration (tape state, tape head position, machine state) of the whole machine, represented with O(f) bits, and has an out-degree of 1. Beginning at the START state, how much time does it take the machien to reach the ACCEPT state?

We first define the predicate FromTo(a, b, k).

$$M \in FromTo(a, b, k) \leftrightarrow \begin{cases} \text{1 if M goes from configuration a to b in} \leq \text{k steps.} \\ \text{0 otherwise} \end{cases}$$

Now to prove the theorem we need to show that FromTo(start, accept, $2^{O(f(n))}$). We will determine this in ATIME(f^2). First notice that

FromTo(a, b, k) = $\exists$c (FromTo(a, c, $\lfloor \frac{k}{2} \rfloor$) $\bigwedge$ FromTo(c, b, $\lceil \frac{k}{2} \rceil$))

An alternating machine may recursively decide the formula by existentially guessing c and universally choosing either the left or right subformula to evaluate. The recursion bottoms ont at k=1, where we'll simulate the machine for 1 step. If k=0, then check that a=b. At each iteration, it takes TIME($O(f)$) to guess c, and the total depth of iterations is $\log_2(2^{O(f)}) = O(f(n))$. Thus the total running time is $O(f)$ steps. ■

Proof : Now we prove for $f(n) \geq n$, ATIME(f) $\subseteq$ SPACE(f)

We first prove that ATIME(f) $\subseteq$ SPACE(f^2), then tighten the bounds. Let $L \in$ ATIME(f), and M is an ATM which accepts L in ATIME(f). We create a space $O(f^2)$ machine M' that simulates M. Like we defined computation trees earlier, the computation tree of M is constructed with nodes that have out-degrees of 0, 1, or 2. This tree has depth $\leq O(f)$. On input w, M' performs a depth-first search of the tree to determine which nodes in the tree are accepting, then accept if the root is accepting. As we traverse the tree, for each node M' stores the configuration of the node as well as a bit indicating which child is currently being investigated. This will take space $O(f)$ for each node, and for a possible $O(f)$ nodes in one branch, this means a total of $O(f^2)$ space.

Now we show how to tighten this bound, we can store only the difference between each node's configuration. Alternately, we could store just the configuration of the current node, and a single bit describing the path that we chose at each level. To back up in the tree, we just need to recompute the needed configuration from the root. For these possibilities, we've reduced the storage space to $O(f)$. ■

Proof : For $f(n) \geq \log n$, ASPACE(f) = TIME($2^{O(f)}$)

Since we already know TIME($2^{O(f)}$) = SPACE(f), we only need to prove that ASPACE(f) $\subseteq$ TIME($2^{O(f)}$) through proof by overkill.

We list the configurations of an ASPACE(f) machine, and then write out its computation graph/graph on configurations. Each node uses $O(f)$ space. Nodes of degree 1 have directed transitions. Nodes of degree 2 are either $\exists$ or $\forall$ nodes. Nodes of degree 0 are leaves of either ACCEPT or REJECT. For our algorithm, we will mark nodes in the graph either ACCEPT or REJECT. At each pass, we find an unmarked node whose descendents are all marked, and mark it appropriately with the AND or OR of the children's states. All leaf nodes have already been marked with its state. The algorithm stops when we've marked the START node.

This process takes time polynomial in the size of the graph. Since the graph can have $2^{O(f)}$ nodes, the algorithm is bounded by TIME(2^f).

■

Proof : For $f(n) \geq \log n$, TIME($2^{O(f)}$) $\subseteq$ ASPACE(f)

Let $L \in$ TIME($2^{O(f)}$) and let M be a $2^{O(f)}$ time machine deciding L. WLOG, assume that M accepts by moving its head to the leftmost end of the tape. Now we'll introduce the notion of a computation tableau of M on input w of length n. The tableau can be thought of as a grid of width and height $2^{O(f)}$. A row in the grid represents configurations of the machine. Each entry in the grid contains that configuration. The grid is of height $2^{O(f)}$ for the steps in time. For example, the top row is the starting configuratino of M on w, the ith row contains the configuration at the ith step of the computation. To index into the tableau, we store the coordinates (i, t) for cell i at time t.

We then construct an ATM which verifies the tableau a small portion at a time. Recall that a cell's contents only depends on the three cells immediately above it in the tableau (except for the top row, which is just the START state). The machine starts at the lower left hand corner of the tableau and verifies that the computation has stopped at the ACCEPT state. It then universally chooses one of the three predecessor cells and then existentially guesses their contents to check that their successor's guessed contents follows in a legal transition. The machine performs this operation recursively until reaching the top row, where it will ACCEPT if those cells' contents are consistent with the input.

More rigorously, this algorithm can be demonstrated this way:

Always start at the lower left corner and verify ACCEPT state.

For each step, Verify($cell(i,t), state(i,t)$)

$$\exists guess \begin{cases} \text{cell(i-1, t-1), state same} \\ \text{cell(i, t-1), state same} \\ \text{cell(i+1, t-1), state same} \end{cases}$$

Reject if $cell(i,t)$ doesn't follow from this set.

$\forall$ choose $x \in -1, 0, 1$ Verify cell(i+x, t-1)

Upon reaching time 0, verify its contents.

At each step, this algorithm stores its current index and the contents of 4 cells. This takes SPACE($O(f)$). Therefore, total space for this algorithm is $O(f)$ because the procedure doesn't have to remember the path it took, only its current state.

■