Lecture 4

Complexity Hierarchies & Alternation

January 15, 1998
Lecturer: Richard Ladner
Notes: Jim Fix

In this lecture we will complete our presentation of space and time complexity hierarchies, and introduce a more powerful Turing machine variant.

4.1. Space and Time Hierarchies

In the last lecture, we stated the Space Hierarchy Theorem, which we will now prove.

Proof:[Space-Hierarchy] We use a diagonalization argument: we construct a $s_2(n)$ space bounded Turing machine T such that for any $s_1(n)$ space bounded Turing machine T' there is an input x such that $x \in L(T')$ exactly when $x \notin L(T)$. Thus, we will demonstrate that $L(T) \notin \text{SPACE}(s_1(n))$. Our construction of T relies on the following fact: if T' is a Turing machine that runs in space $s_1(n)$, then there is a binary description e for T' such that the universal Turing machine, given e and an input x, simulates T' using only $cs_1(n)$ space. In addition, we assume that if e is a binary description of T', then so are the strings $e0, e00$ and so on.

Let T be a Turing machine that, for any input string x, does the following:

1. Lays out $s_2(|x|)$ storage cells. T never exceeds this amount of storage.
2. Checks if x is the binary description of a Turing machine. If not, T rejects the input.
3. Simulates the Turing machine with description x on x in the allotted storage without exceeding $2^{s_2(|x|)}$ time. This requires the maintenance of a time step counter that uses an additional $s_2(|x|)$ space.
4. If the simulation accepts x within the allotted space and time, then T rejects x. Otherwise, T accepts x.

Using linear speedup, it is clear that $L(T)$ can be accepted in space $s_2(n)$. We now argue that $L(T)$ cannot be accepted by any $s_1(n)$ space bounded Turing machine.

Let T' be a Turing machine that runs in space $s_1(n)$ and e be a minimal encoding of T'. On input e, T simulates T' in space $cs_1(|e|)$ and time $2^{ds_1(|e|)}$ for some $0 < c < d$. Choose n so that $s_2(n) > ds_1(n)$ and $n \geq |e|$, hence $s_2(n) > cs_1(n)$ and $2^{s_2(n)} > 2^{ds_1(n)}$. Let $x = e0^{n-|e|}$ and consider the result of T on input x. T will have enough storage and time to simulate T', and will accept x if and only if T' would reject x. Hence, $x \in L(T') \Leftrightarrow x \notin L(T)$. $\square$

There is an analogous hierarchy result for time complexity, demonstrating that increasing compute time yields more powerful complexity classes. A function $t(n)$ is *time constructible* if there is a Turing machine that runs in exactly $t(n)$ steps on input a^n.

Theorem 4.1 (Time Hierarchy): If $t_2(n)$ is time constructible and $\lim_{n\to\infty} \frac{t_1(n)\log t_1(n)}{t_2(n)} = 0$, then

$$\mathrm{TIME}(t_1(n)) \subset \mathrm{TIME}(t_2(n)).$$

The proof of this theorem is very similar to Theorem 3.5, and uses the fact that a universal Turing machine can simulate any $t_1(n)$ time bounded machine using only $ct_1(n)\log t_1(n)$ time.

The space and time hierarchy theorems can be quite useful, for example, we have the following applications:

Corollary 4.2:

$$\mathrm{SPACE}(n) \subset \mathrm{SPACE}(n^2) \subset \mathrm{SPACE}(n^3) \subset \cdots,$$
$$\mathrm{TIME}(n) \subset \mathrm{TIME}(n^2) \subset \mathrm{TIME}(n^3) \subset \cdots.$$

There are similar nondeterministic hierarchies, but for "practical" purposes only the deterministic hierarchies are useful.

4.2. Alternation

An *alternating Turing machine* is a generalization of deterministic and nondeterministic Turing machines. Formally, an alternating Turing machine is a 7-tuple $T = (K, K_E, K_U, \Sigma, \Gamma, \Delta, s)$ defined as in nondeterministic Turing machines with the addition that the set of states K is partitioned into the *existential states* K_E and the *universal states* K_U, hence

- $K_E \cup K_U = K$ and $K_E \cap K_U = \emptyset$.

We define machine configurations as before. A configuration that is in an existential (universal) state is called an *existential (universal) configuration.* From Δ, we can derive the one step relation $\xrightarrow{T}$ on configuration pairs as we did for nondeterministic Turing machines.

For alternating Turing machines, we define a new notion for language acceptance. Given an alternating Turing machine T, a *computation tree* of T is a tree whose nodes are configurations of T. The children of a node of configuration C in the tree are all configurations D such that $C \xrightarrow{T} D$. A computation tree is *accepting* if

- it is a single node and its root configuration is accepting, or

- it has more that one node and either
  - the root configuration is existential and at least one of its subtrees is accepting, or
  - the root configuration is universal and all if its subtrees are accepting.

The language accepted by an alternating Turing machine T is the set of strings $x \in L(T)$ such that the computation tree with root labeled $\mathrm{Init}(x)$ is finite and accepting.

4.3. Complexity of Alternation

ATIME($t(n)$) is the class of languages accepted by alternating Turing machines whose computation trees have depth at most $t(n)$ on an input of length n. ASPACE($s(n)$) is the class of languages accepted by alternating Turing machines whose computation trees are finite and whose configurations use at most $s(n)$ storage. It is common to define

$$\mathrm{AP} = \bigcup_k \mathrm{ATIME}(n^k).$$

Space bounded alternating Turing machines are related to deterministic time bounded Turing machines by the remarkable equivalence:

Theorem 4.3 (Alternating Space): If $s(n) \geq \log n$ then

$$\mathrm{ASPACE}(s(n)) = \bigcup_c \mathrm{TIME}(2^{cs(n)}).$$

This theorem provides interesting alternative characterizations for deterministic time complexity classes, most notably:

Corollary 4.4:

$$\mathrm{ASPACE}(\log n) = P$$

$$\mathrm{ASPACE}(n) = \textstyle\bigcup_c \mathrm{TIME}(2^{cn}) = \mathrm{EXPTIME}.$$

Proof:[Alternating Space Theorem]

($\subseteq$) For now, we assume that $s(n)$ is tape constructible. We will correct this later. Let T be an alternating Turing machine that runs in space $s(n)$. T can be in at most $2^{c_1 s(n)}$ configurations. Consider the directed graph G whose nodes are configurations of T and edges are the successor relation. Define a deterministic Turing machine T' which does the following on an input x of length n:

1. write G on the tape using $2^{c_2 s(n)}$ cells
2. mark all accepting configurations with 1 and all rejecting configurations with 0

3. repeat the following until Init(x) is marked

   (a) For each unmarked exitential configuration, if all its successors are marked, then mark it 0 if all its successor are marked 0 and mark it 1 if one of its successors is marked 1.

   (b) For each unmarked universal configuration, if all its successors are marked, then mark it 1 if all its successor are marked 1 and mark it 0 if one of its successors is marked 0.

4. If Init(x) is marked with 1 accept, otherwise reject.

T' will accept x if and only if T would accept x. Steps 1 and 2 each use no more than $2^{O(s(n))}$ time. Step 3 will make at most $2^{O(s(n))}$ iterations, since at least one configuration will be newly marked in step 3a. Step 3a can naively scan through all $2^{O(s(n))}$ configurations, hence step 3 uses no more than $2^{O(s(n))}$ time over all iterations.

To remove the space constructibility requirement on $s(n)$, we have T' guess the amount of space used by T on x. It does this by trying increasing space bounds $s = 1, 2$ and so on. For each value of s, it lays out the graph G as above, however G is modified slightly:

- The nodes of G are the configurations of T that use space s plus an additional error node E.
- The edges are based on the successor relation, however, if a configuration C has a successor configuration that uses more than s space, then the node for C has an edge into E instead.

We initialize the marking by marking the accepting configuration 1 and the rejecting configuration 0, but E remains unmarked. The marking is then done as described above. If $Init(x)$ gets marked, then T' accepts or rejects accordingly, otherwise the next value of s is tested. Note that if $Init(x)$ does not get marked then a configuration reachable from $Init(x)$ attempts to exceed the allotted space s.

($\supseteq$) This will be covered in the next lecture.

$\square$