CS 710: Complexity Theory 10/6/2011

Lecture 9: Nonuniformity

Instructor: Dieter van Melkebeek Scribe: David Guild

This lecture finishes our discussion of Time-Space lower bounds for SAT. We then introduce the concept of nonuniformity, and explore three different models of nonuniform computation: Boolean circuits, branching programs, and uniform models with advice.

1 Time-Space Lower Bounds for SAT, continued

Recall from the previous lecture that we wish to establish some lower bounds for SAT. Although we cannot yet prove a good lower bound in either time or space individually, we *can* show that no algorithm that solves SAT is both fast and space-efficient. In other words, there might be some linear time algorithm for SAT, and there might be some log space algorithm for SAT, but there is no algorithm that is *both* linear time and log space. Our actual theorem is stronger than this, but shows this result for SAT as well.

In order to show a limit for SAT, we will first prove a result about NTIME(n), and then use NP-completeness to extend this limit to SAT.

Theorem 1. *For any positive c, d where $c(c+d) < 2$,*

$$\text{NTIME}(n) \not\subseteq \text{DTISP}(n^c, n^d).$$

We will prove this theorem by contradiction, using the Nondeterministic Time Hierarchy. In particular, we will assume that NTIME(n) $\subseteq$ DTISP(n^c, n^d) for suitable c, d and then show that this implies that there exists some a, b where $a > b$ and NTIME(n^a) $\subseteq$ NTIME(n^b).

Proof of Theorem 1. Assume the theorem's inclusion holds, and choose some $a \geq 1$.

Claim 1. NTIME(n^a) $\subseteq$ DTISP(n^{ac}, n^{ad})

Proof. Given any language $L \in \text{NTIME}(n^a)$, consider

$$L' = \{x10^{|x|^a} | x \in L\}.$$

We will use $y = x10^{|x|^a}$ to denote a string in L', and set $n = |x|$ and $N = |y|$. Note that

$$L' \in \text{NTIME}(N) = \text{NTIME}(n + 1 + n^a)$$

since we can easily check that y is of the correct form and then simply run the algorithm for L on x. By our assumption $L' \in \text{DTISP}(N^c, N^d)$. Then we can decide L in the following manner: given x, implicitly construct y, run this algorithm for L' on the result, and return that answer. So $L \in \text{DTISP}(N^c, N^d)$ as well, and because $N = O(n^a)$, $L \in \text{DTISP}(n^{ac}, n^{ad})$. □

Claim 2. DTISP(t, s) $\subseteq \Sigma_2$ TIME($\sqrt{ts}$)

See previous lecture for a proof of claim 2.

Putting these two pieces together, we see that $\text{NTIME}(n^a) \subseteq \Sigma_2\,\text{TIME}\left(n^{\frac{a(c+d)}{2}}\right)$. For ease of notation, let $e = \frac{a(c+d)}{2}$. Note that since $c(c+d) < 2$, we know that $ec < a$.

Claim 3. $\Sigma_2\,\text{TIME}(n^e) \subseteq \text{NTIME}(n^{ec})$ *for* $e \geq 1$

Proof. Consider some language $L \in \Sigma_2\,\text{TIME}(n^e)$. Recall that

$$x \in L \iff \left(\exists y_1 \in \Sigma^{|x|^e}\right)\left(\forall y_2 \in \Sigma^{|x|^e}\right)\left(\langle x, y_1, y_2\rangle \in V\right).$$

We can break this apart and look at only the sublanguage L':

$$\langle x, y_1\rangle \in L' \iff \left(\forall y_2 \in \Sigma^{|x|^e}\right)\left(\langle\langle x, y_1\rangle, y_2\rangle \in V\right)$$

Notice that $L' \in \text{coNTIME}(N)$, where $N = |\langle x, y_1\rangle| = \Theta(n^e)$. Our assumption can be used here as well; if there is a deterministic way to decide a language in NTIME, we can certainly decide its complement. In particular,

$$L' \in \text{DTISP}(n^{ec}, n^{ed}) \subseteq \text{DTIME}(n^{ec}).$$

But our original language L is just L' with an extra existential, which implies that $L \in \text{NTIME}(n^{ec})$. □

While claim 3 requires that $e \geq 1$, this is not a problem. We originally chose $a \geq 1$, but we can also require that $a \geq \frac{2}{c+d}$, which gives the needed bound on e.

Adding claim 3 to our proof and choosing a suitable value for a, we see that $\text{NTIME}(n^a) \subseteq \text{NTIME}(n^{ec})$. But this violates the Nondeterministic Time Hierarchy, because $ec < a$. So our original assumption must be false. □

Corollary 1. $SAT \in \text{DTISP}(n^c, n^d) \implies \text{NTIME}(n) \subseteq \text{DTISP}(n^c\,\text{poly-log}\,n, n^d\,\text{poly-log}\,n)$

From this corollary we can show that $SAT \notin \text{DTISP}(n^c, n^d)$ for $c(c+d) < 2$.

Corollary 2. $SAT \notin \text{DTISP}(n^c, \log n)$ *where* $c < \sqrt{2}$.

The following theorem does *not* follow from theorem 1, but is known to be true. Its proof is left as an exercise for the reader; the general form is similar to the theorem above, but with claim 2 applied multiple times.

Theorem 2. $SAT \notin \text{L} \cap \text{DTIME}(n)$

2 Nonuniformity

So far, the algorithms we have discussed are able to handle inputs of any length. We call these *uniform* models of computation: for a given problem, a single algorithm can decide any input. This section will introduce *nonuniform* models of computation, where a single algorithm can only decide input of a certain length. Because each one is limited in scope, we will also talk about *families* of nonuniform algorithms: a set of computation models, each designed for a different input length, that recognize strings in a given language.

In general, we will be most interested in nonuniform models that are *not* easily computable from the input length n. The reason should be clear: if it was easy to create such a model, and that model was efficient, we could just design a uniform algorithm that first computed that nonuniform version and then executed it. By focusing on nonuniform algorithms that are hard to create, we allow for the possibility of breaking the resource bounds established for uniform models.

For example, it might be the case that there exist polynomial time nonuniform algorithms for NP-hard problems of limited input size. This would collapse the Polynomial Time Hierarchy to level 2 (that is, $\Sigma_2^p = \Pi_2^p$), but would still allow P $\neq$ NP. These algorithms could be very useful for real-world applications, such as cryptography. We could imagine a nonuniform computation that can quickly break cryptographic keys of a specific length; even if this algorithm was extremely hard to create, it would be very useful to cryptanalysts.

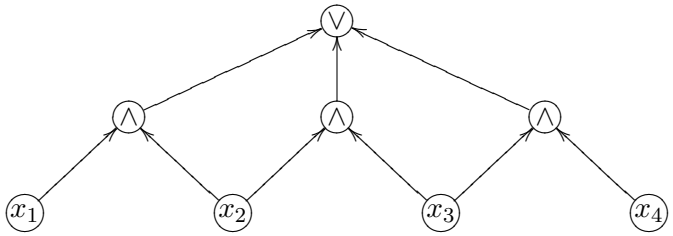
3 Nonuniform Computation Models

This section presents three models of nonuniform computation: Boolean circuits, branching programs, and uniform models with advice. The first two are uniquely nonuniform, and each instance can only handle input of a specific length. The last model is an extension of our standard uniform algorithms with a special *advice* string, which depends only upon the length of the input.

3.1 Boolean Circuits

A Boolean circuit is a directed acyclic graph with a single root node, with out-degree zero, and some number of leaf nodes, with in-degree zero. Each leaf is labeled with an input bit, and each non-leaf node with a logic gate: AND, OR, NOT. The root of the graph is labeled as the output node. This circuit recognizes input in the natural way: setting the value of each leaf according to the bits of the input, values propagate along edges, with each node performing the relevant operation. The result of the computation is the bit produced by the output node.

For example, the following Boolean circuit operates on input of size 4, and accepts any string with two adjacent 1s.



We define the *circuit size* of a Boolean circuit B to be the number of connections (edges) in the graph, and the *circuit depth* to be the length of a longest path from leaf to root. We may also be interested in the *fan-in*, or number of incoming edges for each node. Some proofs may require us to have a bounded fan-in; we will explore this in more detail later on.

In general, circuit size will be the measure of complexity that most closely corresponds to the time complexity in a uniform setting. A Turing machine that runs in time $t(n)$ can be expressed by a Boolean circuit family of size $t(n)^c$. The construction of such a circuit is similar to the construction we used to prove that SAT is NP-complete.

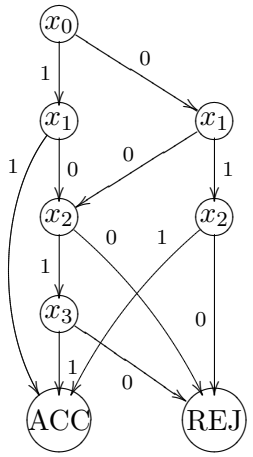
For a language L, we may also consider the circuit complexity $C_L(n)$, which is the size of a smallest Boolean circuit that decides L at length n.

3.2 Branching Programs

A branching program is also a directed acyclic graph. Unlike a Boolean circuit, here the root has in-degree zero and the leaves have out-degree zero. The leaves are labeled with ACCEPT or REJECT, and the other nodes are labeled with input bits. We also require that every non-leaf node has exactly two outgoing edges, which are labeled 0 and 1. The root node is additionally marked as the starting node.

A branching program performs its computation as follows: at each node, look at the bit of input corresponding to this node's label. If it is a 0, move along the outgoing 0 edge to the next node; likewise take the 1 edge if the bit is a 1. Repeat until you arrive at a leaf, then accept or reject according to the leaf's label.

One possible branching program on four input bits, which accepts the same values as the Boolean circuit above, is the following:

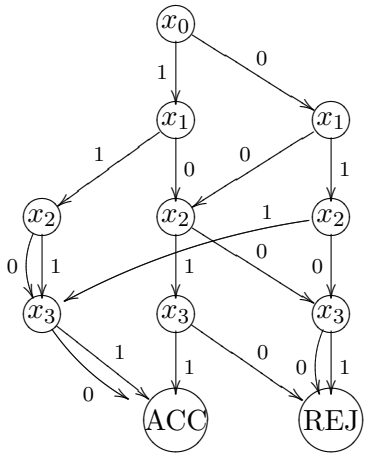


We again define the program's size as the number of connections, and the depth as the maximum path length (although this time from root to leaf). A branching program is *layered* if we can arrange the nodes such that nodes in layer i only have edges to nodes in layer $i+1$. For layered programs, we also define the *width* to be the number of nodes in the largest layer. The program above is not layered, because it has edges that "skip" downward.

A layered program is additionally *oblivious* if all nodes in a layer have the same label. Note that a non-layered program can be transformed into an equivalent layered program by adding dummy nodes (where both edges point to the same node), increasing the program's size by a small constant factor. We can also transform a program into an equivalent oblivious one, but this increases its size by a factor of n.

The branching program above can be transformed into this oblivious layered program. Notice

that as a layered program, its width is 3, not 2.



Similar to circuit complexity, we define the complexity of a branching program $BP_L(n)$ to be the size of the smallest branching program that decides a language L at length n.

For a uniform problem with time bound $t(n)$ and space bound $s(n)$, we can easily construct an equivalent branching program with size $2^{O(s(n))}$, depth $t(n)$, and width $2^{O(s(n))}$. The basic idea is to simulate the computation tableau of a Turing machine. Our branching program will be layered, with each layer corresponding to a single time step of the TM. Within each layer, there will be one node for every possible configuration, and those nodes will be labeled with the input bit being read. The edges are assigned according to the TM's transition function. This branching program starts at the node matching the initial configuration of the TM (starting state, empty work tape). Nodes whose configuration is in an accept or reject state are labeled with ACCEPT or REJECT appropriately.

The depth is clearly $t(n)$: this is a layered program with one layer for every time step of the TM. The width is equal to the number of possible configurations: $2^{s(n)} \cdot |Q| \cdot s(n) \cdot n$. (That is: contents of the work tape, current state, work tape head position, input tape head position.) This is best described as $2^{O(s(n))}$. Since every layer has the same number of nodes, the total size is just depth $\cdot$ width, which is also $2^{O(s(n))}$.

3.3 Uniform Models with Advice

Uniform models with advice are, as the name suggests, an extension of the familiar uniform computation models. While the previous two categories used a different machine for each input length, this type of computation model will use a single machine for all inputs. The *advice* is added as a kind of additional input, much like the certificate or guess for a verifier of a non-deterministic language. However, the advice here may only depend on the length of the input, not on the input value itself.

At first glance it might seem that this is less powerful than simple non-deterministic methods: we are limiting our extra information to only a function of input length. The key difference is that with non-determinism, we still need to verify that the hint is correct. Here, our advice is more like an oracle, and we can use it to "precompute" much harder problems. For example, the language $L = \{\ 0^n \mid$ the TM encoded by n halts on input $n\}$ is undecidable with uniform models, but trivial with advice. We just set the advice to be the correct true/false value for n.

Definition 1. *Given uniform complexity class $\mathcal{C}$ and advice size bound $a : \mathbb{N} \to \mathbb{N}$, define*

$$\mathcal{C}/a = \left\{ L \mid \left(\exists L' \in \mathcal{C}\right) \left(\exists y_0, y_1, y_2, \ldots \in \Sigma^*\right) \text{ such that } x \in L \iff \langle x, y_{|x|} \rangle \in L' \right\}$$

where $|y_n| \leq a(n)$.

For example, P/poly is the class of all languages that can be computed in polynomial time with polynomial-length advice; we could equivalently write $\cup_{c>0}$ P/n^c. Similarly L/poly is the class of languages computable in log space with polynomial advice.

Theorem 3. P*/poly* $= \{L \mid \exists c \; \forall n \; C_L(n) < n^c\}$

Proof. If L has poly-size circuits, then simply let our advice be an encoding of the circuit for input length n, and let L' be a language that runs the encoded circuit with our input x. Then L is also in P/poly.

Recall that we can build a circuit of size $t(n)^c$ from a uniform model. Then for a language $L \in$P/poly, build a circuit that mimics L' with y_n built in. □

Theorem 4. L*/poly* $= \{L \mid \exists c \; \forall n \; BP_L(n) < n^c\}$

Proof. This proof is essentially the same as the previous. Running an advice-encoded BP only requires log space, because we just need to keep track of the current node. Going the other way, $s(n) = O(\log n)$ reduces the size of the BP to $O(n)$, which is polynomial. □

Next Time

In the next lecture, we will show that if SAT has nonuniform models of polynomial size, then the Polynomial Time Hierarchy collapses to level 2 (NP = coNP).

Acknowledgments

In writing the notes for this lecture, I perused the notes by Mushfeq Khan & Chi Man Liu for lecture 7 from the Spring 2010 offering of CS 710.