6.841/18.405J: Advanced Complexity Theory February 6, 2002

Lecture 1

Lecturer: Madhu Sudan *Scribe: April Rasala*

1 Administrivia

The course web page is http://theory.lcs.mit.edu/~madhu/ST02/. If you have not received a class email from Professor Sudan, please email him at madhu@mit.edu to be added to the class email list. As mentioned on the course web page, grades for the course will be based on scribing 1-2 lectures, 4 problem sets and class participation. The only prerequisite for the course is 6.840 or an equivalent course at another institution.

2 Course Contents

Informally, complexity theory examines the effect of limiting resources on the ability to solve a computational problem. Typical resources to consider are time, space, and non-determinism. We'll also consider quantifier alternation and talk about the polynomial time hierarchy. In addition, we will look at randomness as a resource. In determining the power of randomization it is natural to consider related topics like derandomization and pseudorandomness. We will touch on these topics but anyone with a specific interest in derandomization and pseudorandomness might also want to check out a course at Harvard offered by Salil Vadhan (http://www.courses.fas.harvard.edu/~cs225/).

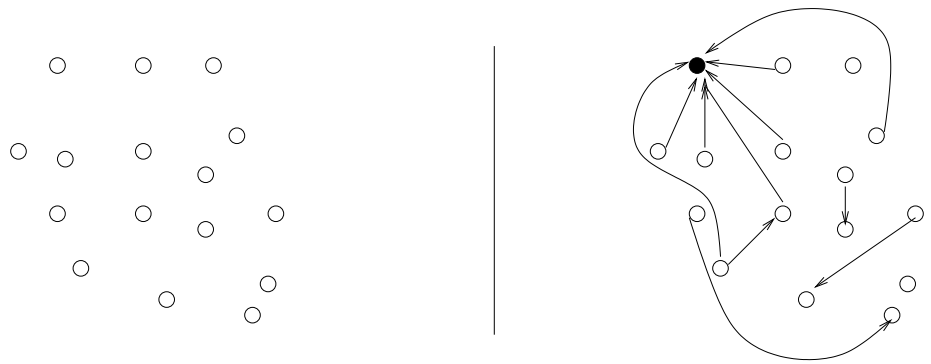
We will consider questions such as "What is a proof", "How does interaction affect computation?", "How does knowledge affect computation"? We will also briefly study topics related to circuit complexity and lower bounds for computation. Finally, if time permits, we will consider other models of computation such as quantum computing.

A more detailed description of possible topics and a lecture schedule is available on the course web page.

3 Classical Computational Complexity

Starting with work by Edmonds, Karp, Cook and Levin, complexity theory has traditionally looked at what sorts of things we can compute when we place a limit on a particular resource. For instance, we might restrict the amount of space used in the computation to be at most logarithmic in the input size. Other classic resources to consider are time and non-determinism.

Given a limit on some resource, such as time or space, we then try to collect a set of interesting problems. We attempt to link these problems together by understanding which problems are computationally no easier than other problems. We can imagine that we have a set of problems which are represented pictorially as a bunch of dots (figure on the left). If we can determine that problem x is no harder than problem y, then we



draw an arrow from the dot representing x to the dot representing y. Interesting problems, under the given resource restriction, are those problems with many arrows pointing toward them (for instance the dark dot in the picture on the right). We then define what aspects of this problem make it interesting and determine other problems that have a similar structure. In this way, we can show that this is not a unique interesting problem. This allows us to define a complexity class of problems.

Another question to ask is "what arrows can we rule out?" Hopefully we can eventually get a map of all computational problems and the complexities involved.

4 Reductions

A formal reduction allows us to turn this notion of the relative hardness of problems into a precise definition. As it turns out, there are already two different types of reductions and both are important for our purposes.

Let O be an oracle (or subroutine) that solves problem y.

- Turing reduction: Given oracle O solving y with the given resource restrictions, there is an algorithm using O that can solve problem x with the same resource restrictions.
- Karp reduction: Given an oracle O solving y, there is an algorithm that can solve problem x by using O once and simply returning the output.

Clearly both types of reductions would allow us to draw some connections between different problems. However, Karp reductions require that x and y be the same "type" of problem. In contrast, a Turing reduction can be used to show that a search problem (say finding a satisfying solution to an instance of SAT) is no harder than a decision problem (such as determining if a given SAT instance is satisfiable). Therefore Turing reductions allow us to focus on questions involving membership in a language.

Since Karp reductions are a restricted version of Turing reductions, the next question is why do we also need Karp reductions? One justification is that Turing reductions might be too powerful to really describe subtle relative difficulties of problems. For instance, suppose we want to determine if a 3-CNF formula ϕ is unsatisfiable. Using a Turing reduction we could show that this was "no harder" than determining if ϕ is satisfiable since we could simply call the oracle for 3SAT on ϕ and then negate the answer provided by the oracle. But this doesn't mesh well with our intuition that seems to say that there is something harder about showing that a boolean formula is unsatisfiable. If we use a Karp reduction then we are not allowed to negate the answer that the oracle produces.

Thus the benefit of Turing reductions is that they allow us to restrict our attention to decision questions about membership in a language. Once we are concerned with only languages,Karp reductions provide more insight.

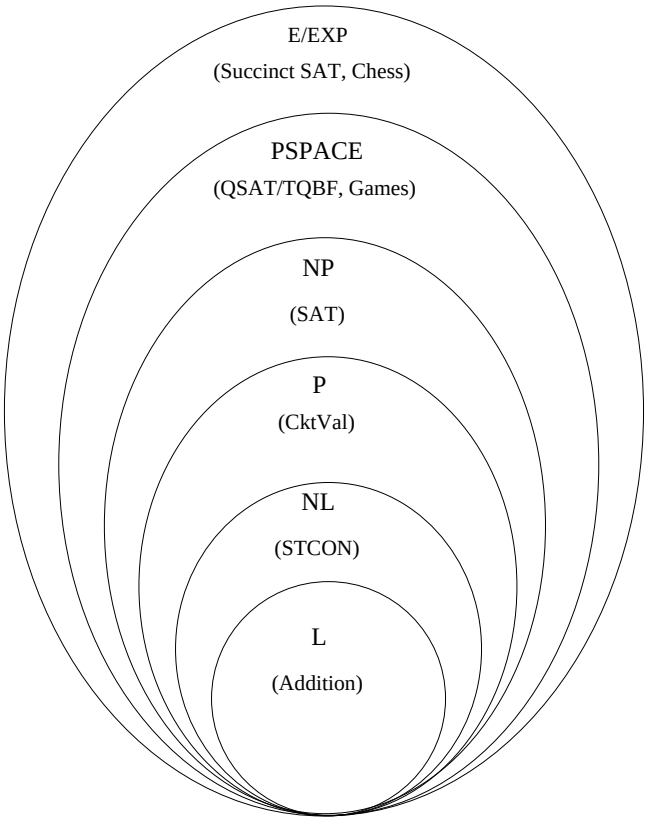
4.1 A Word on Notation

If there exists a polynomial time Turing machine M with access to an oracle for problem L_2 that can solve problem L_1, then we write $L_1 \leq_T^p L_2$. To denote "algorithm A using O" we use A^O.

5 Complexity Classes

The classical complexity classes are depicted in the following picture. Note that `E` refers to the class of languages that can be decided in $\texttt{TIME}(2^{O(n)})$ and `EXP` refers to the class of languages that can be decided in $\texttt{TIME}(2^{n^{O(1)}})$.

What can we say about these classes? First, as shown in the picture, the set of problems that can be solved in $t(n)$ steps on a deterministic Turing machine is contained within the set of problems that can be solved in $t(n)$ time on a non-deterministic Turing machine. This set of problems is in turn contained within the set of problems that can be solved using only $t(n)$ space. Stated formally,



$$\texttt{TIME}(t(n)) \subseteq \texttt{NTIME}(t(n)) \subseteq \texttt{SPACE}(t(n)).$$

Furthermore, $\texttt{SPACE}(t(n)) \subseteq \texttt{TIME}(2^{t(n)})$.

5.1 Hierarchy Theorems

Roughly the Time Hierarchy Theorem says that if $t_1(n) \ll t_2(n)$, then $\texttt{TIME}(t_1(n)) \subset \texttt{TIME}(t_2(n))$. In making this precise, we must consider two cases

Theorem 1 *The set of languages that can be recognized by a Turing machine in* $\texttt{TIME}(t_1(n))$ *is strictly contained within the set of languages that can be recognized by a single-tape Turing machine in* $\texttt{TIME}(t_2(n))$ *if* $t_2(n) = \Omega(t_1^2(n))$.

Theorem 2 *The set of languages that can be recognized by a Turing machine in* $\texttt{TIME}(t_1(n))$ *is strictly contained within the set of languages that can be recognized by a 2-tape Turing machine in* $\texttt{TIME}(t_2(n))$ *if* $t_2(n) = \Omega(t_1(n) \log t_1(n))$.

Like the Time Hierarchy Theorem, the Space Hierarchy Theorem allows us characterize some of the relationships between the complexity classes based on restricted space. However, it requires only that $t_2(n)$ be growing slightly faster than $t_1(n)$.

Theorem 3 *The set of languages that can be recognized by a Turing machine in* $\texttt{SPACE}(t_1(n))$ *is strictly contained within the set of languages that can be recognized by a Turing machine in* $\texttt{SPACE}(t_2(n))$ *if* $t_1(n) = o(t_2(n))$. [1]

[1] Note: We assume that $t(n) = \Omega(\log n)$.

The above theorems apply to deterministic classes. The technique for proving them relies on a simple diagonalization argument. Roughly, suppose we want to show that there is some language L that is in time $t_2(n)$ but not in time $t_1(n)$. We do this by specifically defining L such that it is different from all languages that are decidable in time $t_1(n)$. In particular, suppose that Turing machine M runs in time $t_1(n)$. If M accepts its own encoding, then we define L to reject the encoding of M. On the other hand, if M rejects its own encoding, then we define L to accept the encoding of M. Thus $L = \{\langle M \rangle \mid M \text{ runs in time } t_1(n) \text{ and } \langle M \rangle \notin L(M)\}$

Since every language decidable in time $t_1(n)$ has an associated Turing machine M that recognizes it and since L is designed to specifically differ from M on the encoding of M itself, L must differ from every language decidable in time $t_1(n)$. The only trick is that we need to be sure that the Turing machine deciding L has enough time to simulate M on its own encoding. Thus the need for time $t_1^2(n)$ with a single tape Turing machine and time $t_1(n) \log t_1(n)$ if the machine deciding L has two tapes.

Notice that the diagonalization argument required that we negate the decision of M on its own encoding. This is only possible if M is a deterministic Turing machine. However, a more difficult argument due to Cook shows that $\texttt{NTIME}(t_1(n)) \subset \texttt{NTIME}(t_2(n))$ as long as t_2 is slightly faster growing than t_1.

Finally, Blum's Speedup Theorem says that if we can solve a problem in time cn for some constant c, then we can solve it in time $c'n$ for any constant $0 < c' < c$ by making the Turing machine "bigger" by a constant factor.

5.2 Open Questions

Probably the biggest open question is whether P equals NP. We will generally assume $\texttt{P} \neq \texttt{NP}$. A stronger assertion would be that $\texttt{NP} \neq \texttt{CoNP}$. Weaker assumptions would be $\texttt{P} \neq \texttt{PSPACE}$, $\texttt{SAT} \notin \texttt{L}$ or $\texttt{NP} \neq \texttt{L}$, and $\texttt{SAT} \notin \texttt{TIME}(n \log^c n)$ for any constant c. In addition we can consider these relationships in pairs. For instance we know that either $\texttt{L} \neq \texttt{P}$ or $\texttt{P} \neq \texttt{PSPACE}$ by the Space Hierarchy Theorem.

5.3 Savitch's Theorem

We now consider the relationship between non-deterministic space and deterministic space. We will be proving Savitch's Theorem which says that something computable in non-deterministic space $t(n)$ is computable in deterministic space $t^2(n)$.

We begin by considering the space needed to compute the composition of two functions.

Lemma 4 *If* $g : \{0,1\}^n \to \{0,1\}^n$, $f : \{0,1\}^n \to \{0,1\}^n$, $g \in \texttt{SPACE}(s_1(n))$ *and* $f \in \texttt{SPACE}(s_2(n))$, *then* $f \circ g \in \texttt{SPACE}(s_1(n) + s_2(n))$

Sketch of Proof The obvious algorithm to compute $f(g(n))$ would be to compute $g(n)$ and then compute $f(g(n))$. However, we don't know if we can write down all of $g(n)$ in $\texttt{SPACE}(s_2(n))$ (for example if $s_2(n) = \log n$). We get around this problem by computing the i^{th} bit of $g(n)$ whenever we need it. Thus each time we need a bit of the input, we recompute $g(n)$ and only write down the bit that we need at that step of the computation. Thus we can reuse the same $s_1(n)$ space each time we compute another bit of $g(n)$. At the same time we can be using $s_2(n)$ space to compute $f(g(n))$. Thus the total space used is at most $s_1(n)+s_2(n)$ as claimed. ■

We are now ready to prove Savitch's Theorem.

Theorem 5 *For any* $t(n) \geq \log n$, $\texttt{SPACE}(t(n)) \subseteq \texttt{SPACE}(t^2(n))$.

Sketch of Proof We first simplify the task by showing that we only need to consider $t(n) = \log n$. By a standard padding argument, one can see that if $t(n) = \Omega(\log n)$ then we can always pad the input with enough zeros so that the algorithm that ran in $t(n)$ space, now runs in $\log n'$ space where n' is the length of the input after it has been padded. Thus proving this theorem for $t(n) = \log n$ proves it for all $t(n) \geq \log n$.

To further simplify things we consider an NL-complete problem, namely

$\texttt{STCON} = \{(G, s, t) \mid$ there exists a path from s to t in directed graph $G.\}$

(This problem is also known as PATH.) First, G can be represented by the adjacency matrix A where $A(i,i) = 1$ for all $0 \leq i \leq n-1$ and $A(i,j) = 1$ if and only if there is an edge from i to j in G. Given this representation of G, using boolean multiplication,

$$A^n = \underbrace{AAA \ldots A}_{n \ times}$$

will have the property that $A^n(i,j) = 1$ iff there is path of length at most n from i to j in G and $A^n(i,j) = 0$ otherwise. Thus if we can compute A^n in $O(\log^2 n)$ space, then we will have shown that $\texttt{STCON} \in \texttt{SPACE}(\log^2 n)$ and hence all problems in $\texttt{NL} \in \texttt{SPACE}(\log^2 n)$.

Consider the space needed to compute A^{2^L}. If $L = 1$, then $A^{2^1} = A^2$ which can easily be done in $c \log n$ space for some constant c. Now we assume for the purpose of induction that we can compute $A^{2^{L-1}}$ in $c(L-1)\log n$ space. To show that A^{2^L} can be computed in $cL \log n$ space we first compute $A^{2^{L-1}}$ in $c(L-1)\log n$ space. We can then square $A^{2^{L-1}}$ in $c \log n$ space. Since we used $c(L-1)\log n$ space to compute $A^{2^{L-1}}$ and $c \log n$ space to compute $A^{2^{L-1}} \cdot A^{2^{L-1}}$, by Lemma 4, we can compute A^{2^L} in $c(L-1)\log n + c \log n = cL \log n$ space.

To complete our proof note that $A^n = A^{2^{\log n}}$ and therefore A^n can be computed in $\log^2 n$ space. ■