18.405J/6.841J Advanced Complexity Theory February 3, 1998

Lecture 1

Lecturer: Daniel A. Spielman *Scribe: Zulfikar A. Ramzan*

1 Course Overview

In 18.404/6.840, the prerequisite for this course, Mike Sipser discussed the issue of P vs. NP. In this course, we hope to understand P, NP and much much more. We will discuss topics such as: Randomness and Computation, Counting Classes, the Polynomial Hierarchy, Circuit Complexity, Interactive Proofs, etc.

There will be 4-5 problem sets. If you can get at least 60% of the problems correct, you will get an A. You are allowed to collaborate and use other sources, as long as you cite any help you received.

2 Time Complexity Classes

Define

$$\text{TIME}(t(n)) = \{\text{languages } L : L \text{ is accepted by a Turing machine that halts within } O(t(n)) \text{ steps}\}.$$

We won't specify the sort of Turing machine (e.g., one or two tape, or even a RAM or Kolmogorov machine). As long as we are consistent, it won't matter. Different definitions can change times by a polynomial factor, but that does not change the definitions of classes like P and NP. Also note that we used big O notation in our definition. We won't look at constant factors in too much detail for this course. Blum proved results a while back formally showing that we can more or less ignore constant factors for the time complexity classes in which we are interested.

Define

$$\text{P} = \bigcup_{k \geq 0} \text{TIME}(n^k).$$

Of course, the "P" stands for "polynomial time". We'll treat P as the class of languages that can be computed reasonably/efficiently. Define

$$\text{EXPTIME} = \bigcup_{k \geq 0} \text{TIME}(2^{n^k}).$$

Another interesting class is:

$$E = \bigcup_{k \geq 0} \text{TIME}(k^n).$$

Obviously P $\subset$ EXPTIME. One of the few facts we'll prove in this course is that P $\neq$ EXPTIME. Most of the other statements we deal with will depend on unproved hypotheses. For example, we might prove that P $\neq$ NP implies that a certain problem is hard, but we generally won't be able to prove it unconditionally.

We can in fact prove the following theorem, which is much stronger than P $\neq$ EXPTIME.

Theorem 1 (Time Hierarchy Theorem) *If $f(n) = o(g(n)/\log g(n))$, and f, g are "well behaved" then*

$$\mathit{TIME}(f(n)) \neq \mathit{TIME}(g(n)).$$

This theorem was proved in a prerequisite class. The proof is an application of diagonalization. Also, we won't go into exactly what it means for a function to be "well behaved." We more or less mean that the functions are efficiently computable by a Turing Machine. Most of the functions we will deal with in this course exhibit this property.

Define

$$\begin{aligned} \text{NTIME}(t(n)) = \{L : L \text{ is accepted by a non-deterministic Turing machine} \\ \text{that always halts within } O(t(n)) \text{ steps}\}. \end{aligned}$$

Recall that a non-deterministic Turing machine has non-deterministic states, where it can branch. The machine accepts if at least one of its non-deterministic branches leads to an accepting computation.

Non-deterministic polynomial time is defined as:

$$\text{NP} = \bigcup_{k \geq 0} \text{NTIME}(n^k).$$

The fundamental question we'd like to answer is whether P = NP. Unfortunately, we've got no idea how to answer it. Most people believe, however, that P $\neq$ NP

One useful way to think of languages in NP is as follows. A language L is in NP if and only if one can verify membership efficiently via short witnesses. More formally, that means that there is a language $L' \in$ P and a polynomial $p(n)$ such that $x \in L$ iff there exists w such that $(x, w) \in L'$ and $|w| \leq p(|x|)$. Such a string w is called a witness.

3 Space Complexity Classes

Define

$$\text{SPACE}(t(n)) = \{L : L \text{ is accepted by a non-deterministic Turing machine}$$
$$\text{that uses at most } O(t(n)) \text{ space on inputs of length } n\}.$$

Define

$$\text{PSPACE} = \bigcup_{k \geq 0} \text{TIME}(n^k).$$

These are the set of languages that can be accepted in a polynomial amount of space. In other words you can design a Turing Machine that decides the language, but which only uses polynomially many (in the input size) tape cells. It's obvious that P $\subset$ PSPACE, but we don't know whether P = PSPACE. Many people conjecture that these two classes are indeed different. In fact, if it were the case that P = PSPACE then most of what we learn in this course would become irrelevant! Moreoever, we don't even know whether NP = EXPTIME or whether NP = PSPACE!

Another interesting complexity class we examine in this course is:

$$\text{L} = \text{SPACE}(\log n)$$

For this particular complexity class, we assume we are dealing with a variant of a turing machine in which there is a read-only input tape, and a seperate work tape. It turns out that L $\neq$ PSPACE. In fact, the following theorem gives us a much stronger result:

Theorem 2 (Space Hierarchy Theorem) *If $f(n) = o(g(n))$, and f, g are "well behaved" then*

$$SPACE(f(n)) \neq SPACE(g(n)).$$

The proof of this theorem uses the same techniques as in the proof of the Time Hierachy Theorem. It was also covered in the prerequisite course. One corollary of this theorem is that either L $\neq$ P or P $\neq$ PSPACE, but we don't know which of these facts is true!

4 Reductions, Completeness, and Hardness

Perhaps the most intuitive kind of reductions are the *Polynomial Time Turing Reductions*. These reductions are sometimes refered to as *Cook Reductions* in the literature.

A language A is *polynomial-time Turing reducible* to B if there is a polynomial-time oracle Turing machine $M^?$ that accepts A while using B as an oracle. We write $A \leq_T B$ in this case.

What this definition means is that the Turing machine has access to a "magic" oracle that will answer questions of the form "is $x \in B$" in one step.

More precisely, an *oracle Turing machine* has a special tape on which it can write queries, which are then answered in one step. (We call it an oracle because the oracle is allowed to give answers that could not have been computed efficiently, or even at all.) The reason for using the ? symbol in the notation is that the Oracle Turing Machine itself can be defined independently of the actual oracle you plug in to the machine.

Another type of reduction is the *Polynomial-Time Karp Reduction.* There are many different names for this type of reduction: many-to-one reductions, mapping reductions, or even polynomial time reductions. A language A is *Karp reducible* to a language B if there is a polynomial-time computable function f such that $x \in A$ iff $f(x) \in B$. We write $A \leq_m B$ in that case. The idea is that this condition implies that testing membership in A is no harder than testing membership in B.

A language is said to be *NP-hard* if for all $A \in \text{NP}$, $A \leq_m L$.

A language L is called *NP-complete* if:

1. $L \in \text{NP}$
2. L is NP-hard.

If any NP-complete problem is in P, then P = NP. Examples of NP-complete languages include circuit satisfiability (given a Boolean circuit with one output, can it be made 1?), SAT, and 3SAT. We defined the notions of hardness and completeness with specific reference to Karp reductions. We can define these notions under other kinds of reductions – in some cases we'll need to in order for the defintions to make sense.

Define

$$\text{coNP} = \{L : \bar{L} \in \text{NP}\}.$$

Here $\bar{L}$ is the complement of L. We do not know whether NP = coNP, but most people conjecture that these two complexity classes are inequivalent.

We can define the notion of *co-NP completeness* analagously. *Tautology* (the language consisting of boolean formulae that are true on under all assignments) is the standard example of a co-NP complete problem.

Notice that the Turing reduction gives a perfectly good notion of hardness (in the sense that P = NP if any NP-hard language is in P), but it is not the same notion one gets from mapping reduction. For example, every coNP-complete language is NP-hard under Turing reductions. In particular, $co - SAT \leq_T SAT$ since a SAT oracle can easily be converted to a $co - SAT$ oracle by simply flipping the answer.

5 P-Completeness and the Circuit Value Problem

A language L is said to be *P-complete* under *logspace mapping reductions* if:

1. $L \in P$
2. For all languages $A \in P$, there exists a logspace bounded Turing Machine with a read-only input tape, a write-only output tape, and a read/write work tape, that computes a function f such that: $x \in A \iff f(x) \in L$.

One reason why we are interested in P-complete problems is that we believe they are difficult to compute on a parallel machine. There are many known P-complete problems. We will be interested in the *Circuit Value Problem (CVAL)*. In the Circuit Value Problem, we are given both the description of a circuit and inputs to the circuit, and we are asked to determine the output of the circuit on those particular inputs. It's not hard to see that this can be done in polynomial time. We conjecture, however, that it's hard to come up with an efficient parallel solution.

We will now argue that the CVAL is P-complete under logspace reductions. The following argument is meant to be a proof sketch, and is by no means rigorous. First of all, it's not hard to see that the problem is in P. Now, we must show that for every polynomial time bounded TM M, we can construct a logspace computable function f that takes as input $\bar{x}$ and outputs a circuit C and an assinment to the inputs $\bar{y}$ such that $C(\bar{y})$ accepts if and only if M accepts $\bar{x}$ (in polynomial time). Our construction will be similar to the Cook-Levin construction for the NP-completeness of SAT.

Let $t(n)$ be a bound on the running time of M. Assume without loss of generality, that when M is about to accept (reject), it erases its tape, moves its read head all the way to the left, and enters its accept (reject) state. Moreover, M will continue to stay in the same configuration until it has executed $t(n)$ steps. We will make a $t(n) \times t(n)$ table c, where $c_{i,t}$ consists of the contents of cell i of M's configuration at time t. Now, $c_{i,t}$ only depends on a fixed number of entries in from the $(t-1)$st row. So, we will have for each i and t a circuit segment (of constant size) which computes $c_{i,t}$ from those inputs. We want the output of our main circuit to be $c_{1,t(n)}$. It is easy to construct such a circuit C which computes $c_{1,t(n)}$ by cascading the small circuit segments defined above. This circuit can be produced by a logspace Turing Machine since the structure is repetitive. *QED*