CS151
Complexity Theory

Lecture 1

March 30, 2004

Complexity Theory

Classify problems according to the **computational resources** required

- running time
- storage space
- parallelism
- randomness
- rounds of interaction, communication, others…

Attempt to answer: what is computationally feasible with limited resources?

Complexity Theory

- Contrast with decidability: What is computable?
  - Answer: some things are not
- We care about resources!
  - leads to many more subtle questions
  - fundamental open problems

The central questions

- Is finding a solution as easy as recognizing one?
  P = NP?
- Is *every* sequential algorithm parallelizable?
  P = NC?
- Can *every* efficient algorithm be converted into one that uses a tiny amount of memory?
  P = L?
- Are there small Boolean circuits for *all* problems that *require* exponential running time?
  EXP $\subset$ P/poly?
- Can *every* randomized algorithm be converted into a deterministic algorithm one?
  P = BPP?

Central Questions

We *think* we know the answers to all of these questions …

… **but** no one has been able to prove that even a small part of this "world-view" is correct.

If we're wrong on any one of these then computer science will change dramatically

Introduction

- You already know about two complexity classes
  - **P** = the set of problems decidable in *polynomial time*
  - **NP** = the set of problems with witnesses that can be verified in polynomial time

  … and notion of *NP-completeness*
- Useful **tool**
- Deep **mathematical problem**: P = NP?

Course should be **both** useful and mathematically interesting

A question

- **Given**: multivariate degree r polynomial $f(x_1, x_2, \ldots, x_d)$
  e.g. $f(x_1, x_2, x_3, x_4) =$
  $(x_1^4 - x_3)(x_1 + x_3^2 - 3x_2^5)(4x_1^3 - x_4^2)$
- **Question**: is f identically zero?
- Challenge: devise a deterministic poly-time algorithm for this problem.

A randomized algorithm

- **Given**: multivariate degree r poly. $f(x_1, x_2, \ldots, x_d)$
- **Algorithm**:
  - pick small number of random points
  - if f is zero on all of these points, answer "yes"
  - otherwise answer "no"

(low-degree non-zero polynomial evaluates to zero on only a small fraction of its domain)

- No deterministic algorithm known

Derandomization

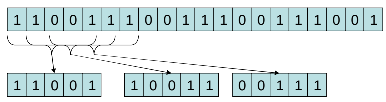
- Here is a deterministic algorithm that works under the assumption that there exist hard problems, say SAT.
- solve SAT on all inputs of length log n

  1 1 0 0 1 1 1 0 0 1

- encode using error-correcting code (variant of a Reed-Muller code)

  1 1 0 0 1 1 1 0 0 1 1 1 0 0 1 1 1 0 0 1

Derandomization



- run randomized alg. using these strings in place of random evaluation points
  - if f is zero on all of these points, answer "yes"
  - otherwise answer "no"
- This works. (proof in this course)

Derandomization

This technique works on any randomized algorithm.

Gives generic "derandomization" of randomized procedures.

A surprising fact

- Is finding a solution as easy as recognizing one?
  P = NP? probably FALSE
- Is *every* sequential algorithm parallelizable?
  P = NC? probably FALSE
- Can *every* efficient algorithm be converted into one that uses a tiny amount of memory?
  P = L? probably FALSE
- Are there small Boolean circuits for *all* problems that *require* exponential running time?
  EXP $\subset$ P/poly? probably FALSE
- Can *every* randomized algorithm be converted into a deterministic algorithm one?
  P = BPP? **probably TRUE**

Outline

Should be mostly review…

1. Problems and Languages
2. Complexity Classes
3. Turing Machines
4. Reductions
5. Completeness

Problems and Languages

- Need formal notion of "computational problem". Examples:
  - Given graph G, vertices s, t, find the shortest path from s to t
  - Given matrices A and B, compute AB
  - Given an integer, find its prime factors
  - Given a Boolean formula, find a satisfying assignment

Problems and Languages

- One possibility: function from strings to strings

$$f:\Sigma^*\rightarrow\Sigma^*$$

- **function problem**:

given x, compute f(x)

- **decision problem**: $f:\Sigma^*\rightarrow$ {yes, no}

given x, accept or reject

Problems and Languages

- simplification doesn't give up much:
  - Given an integer n, find its prime factors
  - Given an integer n and an integer k, is there a factor of n that is < k?
  - Given a Boolean formula, find a satisfying assignment
  - Given a Boolean formula, is it satisfiable?
- solve function problem using related decision problem (how?)
- **We will work mostly with decision problems**

Problems and Languages

- **decision problems**: $f:\Sigma^*\rightarrow$ {yes, no}
- equivalent notion: **language** $L \subset \Sigma^*$

L = set of "yes" instances

- Examples:
  - set of strings encoding satisfiable formulas
  - set of strings that encode pairs (n,k) for which n has factor < k
- decision problem associated with L:
  - Given x, is x in L?

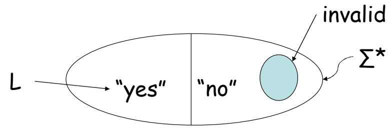
Problems and Languages

An aside: two encoding issues

1. implicitly assume we've agreed on a way to encode inputs (and outputs) as strings
   - sometimes relevant in fine-grained analysis (e.g. adj. matrix vs. adj. list for graphs)
   - almost never an issue in this class
   - avoid silly encodings: e.g. unary

Problems and Languages

2. some strings not valid encodings of any input -- treat as “no”



Complexity Classes

- **complexity class** = class of languages
- set-theoretic definition – no reference to computation (!)
- example:
  - **TALLY** = languages in which every yes instance has form 0^n
  - e.g. L = { 0^n : n prime }

Complexity Classes

- complexity classes you know:
  - **P** = the set of languages decidable in *polynomial time*
  - **NP** = the set of languages L where

    $L = \{ x : \exists\, y, |y| \le |x|^k, (x, y) \in R \}$

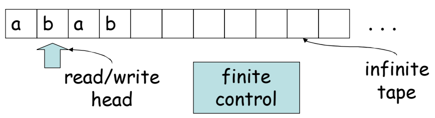
    and R is a language in **P**
- easy to define complexity classes…

Complexity Classes

- …harder to define **meaningful** complexity classes:
  - **capture** genuine computational phenomenon (e.g. parallelism)
  - **contain** natural and relevant problems
  - ideally **characterized** by natural problems (completeness – more soon)
  - **robust** under variations in model of computation
  - possibly **closed** under operations such as AND, OR, COMPLEMENT…

Complexity Classes

- need a **model of computation** to define classes that capture important aspects of computation
- Our model of computation: Turing Machine



Turing Machines

- **Q** finite set of states
- ∑ alphabet including blank: “_”
- $\mathbf{q}_{\mathbf{start}}$, $\mathbf{q}_{\mathbf{accept}}$, $\mathbf{q}_{\mathbf{reject}}$ in Q
- **δ** : Q x ∑ → Q x ∑ x {L, R, -} transition fn.
- input written on tape, head on 1^{st} square, state $\mathbf{q}_{\mathbf{start}}$
- sequence of steps specified by **δ**
- if reach $\mathbf{q}_{\mathbf{accept}}$ or $\mathbf{q}_{\mathbf{reject}}$ halt

Turing Machines

- three notions of computation with Turing machines. In all, input x written on tape…
  - **function computation**: output f(x) is left on the tape when TM halts
  - **language decision**: TM halts in state q_{accept} if $x \in L$; TM halts in state q_{reject} if $x \notin L$.
  - **language acceptance**: TM halts in state q_{accept} if $x \in L$; may loop forever otherwise.

Example:

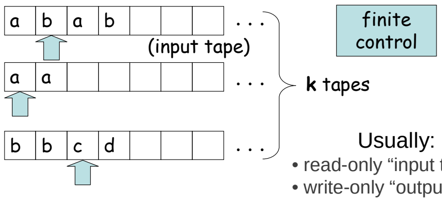
q	σ	δ(q,σ)
start	0	(start, 0, R)
start	1	(start, 1, R)
start	_	(t, _, L)
start	#	(start, #, R)

q	σ	δ(q,σ)
t	0	(accept, 1, -)
t	1	(t, 0, L)
t	#	(accept, #, R)

#	0	1	start
#	0	1	start
#	0	1	start
#	0	1	start
#	0	1	t
#	0	0	t
#	1	0	accept

Turing Machines

- **multi-tape** Turing Machine:

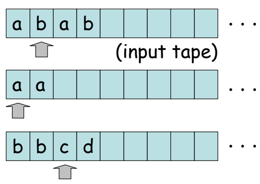


Usually:
- read-only "input tape"
- write-only "output tape"
- k-2 read/write "work tapes"

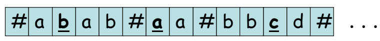
- **δ**:Q x $\Sigma^k \rightarrow$ Q x Σ^k x $\{L,R,-\}^k$

Multitape TMs

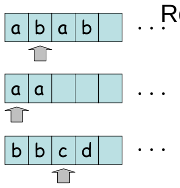
simulation of k-tape TM by single-tape TM:



- add new symbol **x** for each old x
- marks location of "virtual heads"

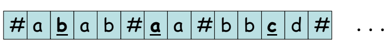


Multitape TMs



Repeat: O(t(n)) times
- scan tape, remembering the symbols under each virtual head in the state O(k t(n)) = O(t(n))
- make changes to reflect 1 step of M; if hit #, shift to right to make room. O(t(n))

when M halts, erase all but 1st string O(t(n))



Extended Church-Turing Thesis

- the belief that TMs formalize our intuitive notion of an efficient algorithm is:

<u>The "extended" Church-Turing Thesis</u>

everything we can compute in time t(n) on a physical computer can be computed on a Turing Machine in time $t^{O(1)}(n)$ (polynomial slowdown)

- quantum computers challenge this belief

Extended Church-Turing Thesis

- consequence of extended Church-Turing Thesis: all reasonable physically realizable models of computation can be *efficiently* simulated by a TM
- e.g. multi-tape vs. single tape TM
- e.g. RAM model

Turing Machines

- Amazing fact: there exist (natural) **undecidable** problems

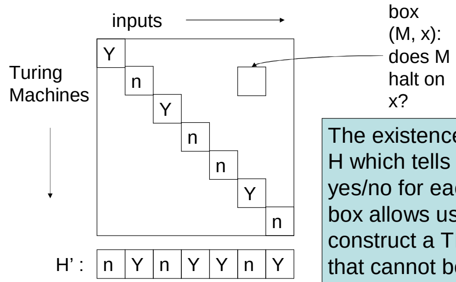
HALT = { (M, x) : M halts on input x }

- Theorem: HALT is undecidable.

Turing Machines

- Proof:
  – Suppose TM H decides HALT
  – Define new TM H’: on input M
    - if H accepts (M, M) then loop
    - if H rejects (M, M) then halt
  – Consider H’ on input H’:
    - if it halts, then H rejects (H’, H’), which implies it cannot halt
    - if it loops, then H accepts (H’, H’) which implies it must halt
  – contradiction.

Diagonalization



The existence of H which tells us yes/no for each box allows us to construct a TM H’ that cannot be in the table.

Turing Machines

- Back to complexity classes:
  – **TIME(f(n))** = languages decidable by a multi-tape TM in at most f(n) steps, where n is the input length, and f : $\mathbf{N} \rightarrow \mathbf{N}$
  – **SPACE(f(n))** = languages decidable by a multi-tape TM that touches at most f(n) squares of its work tapes, where n is the input length, and f : $\mathbf{N} \rightarrow \mathbf{N}$

Note: $\mathbf{P} = \cup_{k >= 1} \mathbf{TIME(n^k)}$

Interlude

- In an ideal world, given language L
  – state an algorithm deciding L
  – **prove** that no algorithm does better
- we are pretty good at part 1
- we are currently **completely helpless** when it comes to part 2, for most problems that we care about

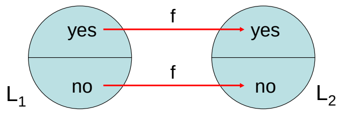
Interlude

- in place of part 2 we can
  - relate the difficulty of problems to each other via **reductions**
  - prove that a problem is a “hardest” problem in a complexity class via **completeness**
- powerful, successful surrogate for lower bounds

Reductions

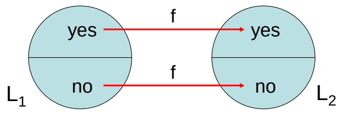
- **reductions** are the main tool for relating problems to each other
- given two languages L_1 and L_2 we say “$\mathbf{L_1}$ **reduces to** $\mathbf{L_2}$” and we write “$\mathbf{L_1 \le L_2}$” to mean:
  - there exists an efficient (for now, poly-time) algorithm that computes a function f s.t.
    - $x \in L_1$ implies $f(x) \in L_2$
    - $x \notin L_1$ implies $f(x) \notin L_2$

Reductions



- positive use: given new problem $\mathbf{L_1}$ reduce it to $\mathbf{L_2}$ that we know to be in **P**. Conclude $\mathbf{L_1}$ in **P** (how?)
  - e.g. bipartite matching **≤** max flow
  - formalizes “$\mathbf{L_1}$ as easy as $\mathbf{L_2}$”

Reductions



- **negative use**: given new problem $\mathbf{L_2}$ reduce $\mathbf{L_1}$ (that we believe not to be in **P**) to it. Conclude $\mathbf{L_2}$ *not* in **P** if $\mathbf{L_1}$ *not* in **P** (how?)
  - e.g. satisfiability **≤** graph 3-coloring
  - formalizes “$\mathbf{L_2}$ as hard as $\mathbf{L_1}$”

Reductions

- Example reduction:
  - 3SAT = { φ : φ is a 3-CNF Boolean formula that has a satisfying assignment }

    (3-CNF = AND of OR of **≤** 3 literals)
  - IS = { (G, k) | G is a graph with an independent set $V' \subset V$ of size $\ge k$ }

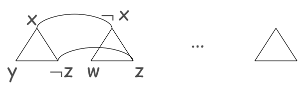
    (ind. set = set of vertices no 2 of which are connected by an edge)

Ind. Set is NP-complete

The reduction f: given

$\varphi = (x \vee y \vee \neg z) \wedge (\neg x \vee w \vee z) \wedge \ldots \wedge (\ldots)$

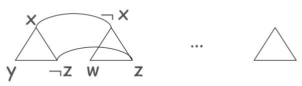
we produce graph G_φ:



- one triangle for each of m clauses
- edge between every pair of contradictory literals
- set k = m

Reductions

$\varphi = (x \vee y \vee \neg z) \wedge (\neg x \vee w \vee z) \wedge \ldots \wedge (\ldots)$



- Claim: φ has a satisfying assignment if and only if G has an independent set of size at least k
- Proof?
- Conclude that 3SAT ≤ IS.

Completeness

- complexity class **C**
- language L is **C-complete** if
  - L is in **C**
  - every language in **C** reduces to L
- very important concept
- formalizes "L is hardest problem in complexity class **C**"

Completeness

- Completeness allows us to reason about the entire class by thinking about a single concrete problem
- related concept: language L is **C-hard** if
  - every language in **C** reduces to L

Completeness

- May ask: how to show *every* language in **C** reduces to L?
  - in practice, shown by reducing **known C**-complete problem to L
  - often not hard to find "1^{st}" C-complete language

Completeness

- Example:

  NP = the set of languages L where

  $L = \{ x : \exists\, y, |y| \leq |x|^k, (x, y) \in R \}$

  and R is a language in **P.**

  one **NP**-complete language "bounded halting":

  $BH = \{ (M, x, 1^k) : \exists\, y$ s.t. M accepts (x, y) in at most k steps $\}$
- challenge is to find **natural** complete problem
- Cook 71 : SAT **NP**-complete

Summary

- problems
  - function, decision
  - language = set of strings
- complexity class = set of languages
- efficient computation identified with efficient computation on Turing Machine
  - single-tape, multi-tape
  - diagonalization technique: HALT undecidable
- TIME and SPACE classes
- reductions
- **C**-completeness, **C**-hardness