Computational Models – Lecture 11[1]

Nachum Dershowitz & Yishay Mansour

Tel Aviv University

June 2017

[1] Based on foils by Iftach Haitner, Benny Chor, Maurice Herlihy.

Talk Outline

- Introduction to time complexity
- The class $\mathcal{P}$
- The class $\mathcal{NP}$
- Verifiability

- Sipser 7.1-7.3

Part I

Introduction to Time Complexity

How long does it take to decide $L = \{0^n 1^n : n \geq 0\}$

Clearly $L \in \mathcal{R}$.

Question 1

How much time does a single-tape TM need to decide L?

Algorithm 2 (Decider M_1 for L)

On input string w:

1. Scan across tape and `Reject` if 0 is found to the right of a 1.
2. While both 0s and 1s appear on tape, repeat the following:

   Scan across tape, crossing off a single 0 and a single 1 in each pass.
3. `Accept` if no 0s and 1s remain; otherwise `Reject`.

We analyze the time it takes to perform each of the three steps separately. In the following let $n = |w|$.

Analyzing Step 1

Scan across tape and `Reject` *if* 0 *is found to the right of a* 1*. If not, return to starting point.*

- Scanning requires n steps.
- Re-positioning head requires n steps.
- Total is $2n = O(n)$ steps.

Analyzing Step 2

While both 0*s and* 1*s appear on tape, repeat the following*

Scan across tape, crossing off a single 0 *and a single* 1 *in each pass.*

- Each scan requires $O(n)$ steps.
- Since each scan crosses off two symbols, the number of scans is at most $n/2$.
- Total number of steps is $(n/2) \cdot O(n) = O(n^2)$.

Analyzing Step 3

If `0`*s still remain after all* `1`*s have been crossed out, or vice-versa,* `Reject`*. Otherwise, if the tape is empty,* `Accept`*.*

- Single scan requires $O(n)$ steps.
- Total is $O(n)$ steps.

Overall cost

Total cost for the three steps

1. $O(n)$
2. $O(n^2)$
3. $O(n)$

which is $O(n^2)$

Deterministic Time

Definition 3 (Deterministic Time)

Let M be a deterministic TM, and let $t : \mathbb{N} \mapsto \mathbb{N}$. We say that M runs in time $t(n)$, if For every input x of length n, the number of steps that $M(x)$ uses is at most $t(n)$.

Question 4

What is a “step"?

Definition 5 (DTIME)

For $t\colon \mathbb{N} \mapsto \mathbb{N}$, let $\mathrm{DTIME}(t(n)) =$
$\{\mathrm{L} \subseteq \Sigma^*\colon$ L is decided by an $O(t(n))$-time single tape TM$\}$

Note that $t(n)$ running time, is also required for strings not in L.

Language Encoding

- Let $L = \{1^n : n \in \mathbb{N}\}$. (over Σ)

  Is L in DTIME(n)? in DTIME($n^{1/2}$)? Proof?

- Let PATH $= \{\langle G, s, t\rangle : G$ has directed path from s to $t\}$

  Is L in DTIME(n)? in DTIME($n^{1/2}$)?

- The questions are not well defined w/o defining the encoding of strings into triplets $\langle G, s, t\rangle$.
- Encoding may matter much – an exponential algorithm with respect to one encoding might turn to linear algorithm with respect to other encoding.

If not defined explicitly, we assume a “reasonable encoding":

- Graph encoding: adjacency list or adjacency matrix
- Integer encoding: binary/decimal encoding (and not unary)

Encoding of integers: binary versus unary

- Factorization algorithm
  1. Do for $i = 2$ to $q - 1$:
     - ⋆ If i divides q output i and halt.
  2. Output "none" and halt.
- What is the running time?
  - $O(q)$ (easily improved to $O(\sqrt{q})$)
  - Yay: poly(nomial) time! ? Yes, if q represented in unary.
  - Uh oh... number of bits to represent q in binary is $\log q$, so $O(q)$ is EXPONENTIAL IN THE INPUT LENGTH
- Today's crypto systems assume that factoring 4,000 bit numbers takes a long time!

Relations among time classes

Let $t_1, t_2 : \mathbb{N} \mapsto \mathbb{N}$ be two functions.

Claim 6

If $t_1(n) = O(t_2(n))$, then $\mathrm{DTIME}(t_1(n)) \subseteq \mathrm{DTIME}(t_2(n))$.

Stated informally, more time does not hurt. But does it actually help?

Would like to say "if $t_1(n) = o(t_2(n))$ then $\mathrm{DTIME}(t_1(n)) \subsetneq \mathrm{DTIME}(t_2(n))$".

But this is what we can get:

Claim 7

(Assume that $t_1(n)$ and $t_2(n)$ are time constructible[a]). If $t_1(n) \in O(t_2(n)/\log(n))$, then $\mathrm{DTIME}(t_1(n)) \subsetneq \mathrm{DTIME}(t_2(n))$.

[a] t is time constructible if the function mapping 1^n to the binary representation of $t(n)$ is computable in time $O(t(n))$).

Informally, sufficiently more time does help

Proof omitted (take complexity course, or search for "time hierarchy theorem")

Back to $L = \{0^n 1^n : n \geq 0\} \in \mathrm{DTIME}(n^2)$

We have seen that $L \in \mathrm{DTIME}(n^2)$. Can we do faster?

Algorithm 8 (Decider M_2 for L)

On input string w

1. Scan across tape and `Reject` if 0 is found to the right of a 1.
2. Repeat the following while both 0s and 1s appear on tape:
   - **2.1** Scan across tape, checking whether total number of 0s plus 1s is even or odd. If odd, `Reject`.
   - **2.2** Scan across tape, crossing off every other 0 (starting with the first), and every other 1 (starting with the first) in each pass.
3. `Accept` if all string is crossed off; Otherwise `Reject`.

Example: $w = 0^{13}1^{13}$

Correctness?

- Let binary representation of number of 0's and 1's be $a_k \dots a_0$ and $b_\ell \dots b_0$ respectively.
- If total number of 0s plus 1s is even, they have same parity $\implies a_0 = b_0$.
- Crossing off every other 0 and 1 has same effect as "shift right" on binary representation.
- So, next iteration checks $a_1 = b_1$...
- When finish, know that $a_i = b_i$ for all i!

Running time analysis of M_2

Algorithm 9 (Decider M_2 for L)

On input string w

1. Scan across tape and `Reject` if 0 is found to the right of a 1.
2. Repeat the following while both 0s and 1s appear on tape:
   - **2.1** Scan across tape, checking whether total number of 0s plus 1s is even or odd. If odd, `Reject`.
   - **2.2** Scan across tape, crossing off every other 0 (starting with the first), and every other 1 (starting with the first) in each pass.
3. `Accept` if all string is crossed off; Otherwise `Reject`.

- One pass in each step (1,2,3) takes $O(n)$ time.
- Steps 1,3: each executed once
- Step 2 executed $1 + \log_2 n$ times
- Total for Step 2 is $(1 + \log_2 n)O(n) = O(n \log n)$.
- Grand total: $O(n) + O(n \log n) = O(n \log n)$.

Further improvements, anybody?

Question 10

Can the running time be made $o(n \log n)$?

Answer: Not on a single tape TM. . .

Claim 11

$L \notin \text{DTIME}(o(n \log n))$ (i.e., $L \notin \text{DTIME}(g(n))$ for any $g(n) \in o(n \log n)$).

Hence, $CFL \not\subset \text{DTIME}(o(n \log n))$.

In contrast, regular languages $\subseteq \text{DTIME}(O(n))$. (?)

The proof of the claim is an immediate corollary of the following lemma.

Lemma 12 (pumping lemma for low-time TMs)

For every TM M of running time $t(n) \in o(n \log n)$, exists $\ell \in \mathbb{N}$ such that: every $w \in \mathcal{L} = \mathcal{L}(M)$ with $|w| \geq \ell$, can be written as $w = xyz$, $|y| > 0$ and $xy^iz \in \mathcal{L}$ for every $i \geq 0$.

A two-tape decider for $L = \{0^n 1^n : n \geq 0\}$

Algorithm 13 (Two-tape Decider M_3 for L)

On input string w:

1. Scan across tape and `Reject` if 0 is found to the right of a 1.
2. Scan across 0s to first 1, copying 0s to tape 2.
3. Scan across 1s on tape 1 until the end. For each 1, cross off a 0. If no 0s left, `Reject`.
4. If any 0s left, `Reject`; otherwise `Accept`.

Question 14

What is M_3 running time?

Complexity of deciding $L = \{0^n 1^n\}$

- Single-tape M_1: $O(n^2)$.
- single-tape M_2: $O(n \log n)$ (fastest possible!).

  Hence $L \in \mathrm{DTIME}O(n \log n)$, but not in $\mathrm{DTIME}(O(f(n)))$ for $f(n) \in o(n \log n)$

- Two-tape M_3: $O(n)$.

Important difference between complexity and computability:

- Computability: all reasonable models equivalent (Church-Turing)
- Complexity: choice of model does affect running time.

Question 15

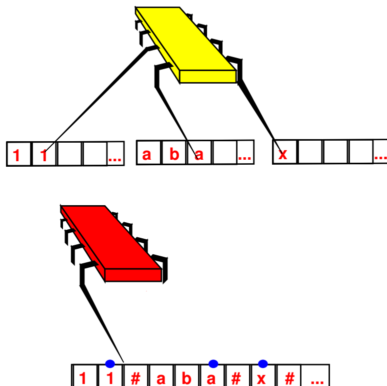
By how much does a model affect complexity?

Multitape speedup

Let $t(n)$ be a function where , and let $L \subseteq \Sigma^*$ be a language.

Claim 16

Assume $\exists$ $t(n)$-time multitape TM that decides L with $t(n) \geq n$, then $\exists$ an $O(t(n)^2)$-time single-tape TM that decides L.



Reminder: Emulating multitape TMs

Algorithm 17 (Single tape emulator S for k-tape M)

On input $w = w_1 \cdots w_n$:

1. Write $\# \dot{w}_1 \; w_2 \cdots w_n \# \dot{\sqcup} \# \dot{\sqcup} \# \cdots \#$ on tape.
2. Scan tape from first $\#$ to $(k+1)$-st $\#$ to read symbols under virtual heads.
3. Rescan tape to write new symbols and move heads.
4. If need to move virtual head onto $\#$, shift tape content to the right.

- For each step of M, the emulator S performs 2 scans and up to k rightward shifts
- On input of length n, M makes $O(t(n))$ many steps, so active portion of each tape is $O(t(n))$ long.
- Total number of steps S makes:
  - $O(k \cdot t(n)) = O(t(n))$ steps to simulate one step of M.
  - Initial tape arrangement $O(n)$.
  - Grand total: $O(n) + O(t(n)^2) = O(t(n)^2)$ steps (recall $t(n) > n$).

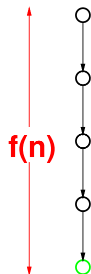
Non-deterministic time

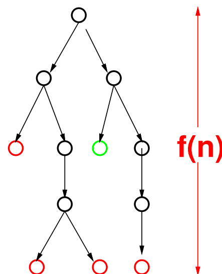
Definition 18 (nondeterministic time)

A non-deterministic TM N runs in time $f(n)$, where $f : \mathbb{N} \mapsto \mathbb{N}$, if for every input x of length n, the maximum number of steps that N uses on any branch of its computation tree on x, is at most $f(n)$.

Notice that also non-accepting branches must reject within $f(n)$ many steps.

deterministic **nondeterministic**





TAKE NOTE: the depth of the tree, not the size of the tree!!!

Non-determinism speedup

Claim 19

Suppose N is a nondeterministic TM that runs in time $t(n)$ and decides the language L. Then there exists an $2^{O(t(n))}$-time deterministic TM D that decided L.

Note contrast with multi-tape result.
Proof's idea: D emulates N. Reminder

- D tries all possible branches (say using BFS).
  - `Accept` if finds any accepting state.
  - `Reject` if all branches reject.
- Since N always stops, exactly one of two possibilities must occur.

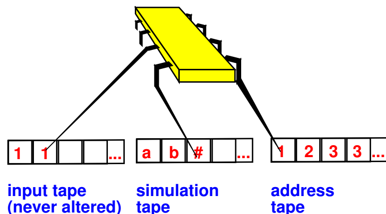
Emulation details

We view the computation of N as a tree, whose nodes are configurations of the TM (i.e., state, head location and tape content).

- Root is starting configuration,
- Fanout is at most some constant b (?),
- Depth at most $\leq t(n)$,
- Total number of nodes $O(b^{t(n)})$,
- Emulation time is $O(b^{t(n)}) = O(2^{O(t(n))})$

Remarks

1. Breadth-first search used in emulation
   - Visit each node.
   - May be improved upon by using depth-first search (is it OK?) or other tree search strategies.
   - Still, doing this may save constants, but nothing substantial (?)
2. Simulation uses a three-tape machine.



Single-tape simulation: $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$.

Polynomial is faster than exponential

Assume one step takes $1/10^6$ fraction of a second....

	10	20	30	40	50	60
n	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2	.0001 second	.00004 second	.00009 second	.00016 second	.00025 second	.00036 second
n^3	.001 second	.00008 second	.027 second	.064 second	.125 second	.216 second
n^5	.1 second	3.2 seconds	24.3 seconds	1.7 minute	5.2 minutes	13.0 minutes
2^n	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	.059 second	58 minutes	6.5 years	3855 centuries	$2 \cdot 10^8$ centuries	$1.3 \cdot 10^{13}$ centuries

Exponential time factoring algorithms

2008 claim on forum:

1 hr for 120 bits
10 hrs for 150 bits
100 hrs for 180 bits
1000 hrs for 210 bits

Gaps between models

- At most polynomial gap in time to perform tasks between different deterministic models (single- vs. multi-tape TMs, TM vs. RAM, etc.)
- Apparently exponential gap in time to perform tasks between deterministic and non-deterministic models.

Claim 20

All “reasonable” classical (not including quantum) models of computation are polynomially equivalent.

Any one can simulate another with only polynomial blowup in running time.

Fact 21

Is a given problem solvable in

- *Linear time? model-specific.*
- *Polynomial time? model-independent.*

Part II

The Class P

The class $\mathcal{P}$

$\mathcal{P}$ is the set of languages decidable in polynomial time on deterministic TMs.

Definition 22 ($\mathcal{P}$)

$\mathcal{P} = \bigcup_{c \geq 0} \text{DTIME}(n^c)$

The class $\mathcal{P}$ is important because:

- Invariant for all (deterministic) models of computation polynomially equivalent to TMs
  - not affected by particulars of model ...
  - go ahead, have another tape, they're pretty small and inexpensive ...
- Roughly corresponds to realistically solvable (tractable) problems.
  - actually depends on context
  - going from exponential to polynomial algorithm usually requires major insight,
  - if you find an inefficient polynomial algorithm, you can often find a more efficient one.

Known problems in $\mathcal{P}$

- Integer arithmetic: Addition, subtraction, multiplication, division with remainder.
- Modular arithmetic: Exponentiation (RSA), inverse.
- Integer Algorithms: Greatest common divisor (gcd).
- Operations research: Maximum network flow, linear programming.
- Algebra: Matrix multiplication, computing determinants, matrix inversion, solving systems of linear equations, factoring polynomials.
- Graph algorithms: BFS and DFS in graphs, minimum spanning trees, finding Eulerian path.

PATH

$$\text{PATH} = \{\langle G, s, t\rangle : G \text{ has directed path from } s \text{ to } t\}$$

Theorem 23

PATH $\in \mathcal{P}$.

Algorithm 24 (M_1 – Naive algorithm for PATH)

Input: an m nodes graph G
For each path p in G of length $\leq m$: check if p goes from s to t.

Question 25

What is the (time) complexity of M_1?

- there are m^m possible paths

  $\implies$ exponential in number of nodes

  $\implies$ exponential in input size
- Oh, oh. Does not sound like $\mathcal{P}$ to me ...

Efficient algorithm for PATH

Algorithm 26 (M_2 – efficient algorithm for PATH)

1. Place mark on s.
2. Repeat until no additional nodes marked:
   2.1 Scan edges of G.
   2.2 If edge (a, b) found from marked node a to unmarked node b, mark node b.
3. Accept if t is marked; otherwise Reject.

Question 27

What is the complexity of M_2?

- Steps 1 and 3 run once.
- Step 2 runs at most m times, because each time (except last) it marks at least one new node.

$\Longrightarrow$ total number of steps is polynomial.

Relative primality

Two numbers are relatively prime if their *greater common divisor (*gcd*)* is 1 (i.e., the largest integer that divides them both).

- $\text{gcd}(10, 21) = 1 \implies 10$ and 21 are relatively prime
- $\text{gcd}(10, 22) = 2 \implies 10$ and 22 are not relatively prime

Definition 28

$\text{RELPRIME} = \{\langle x, y\rangle : \text{gcd}(x, y) = 1\}$.

Theorem 29

$\text{RELPRIME} \in \mathcal{P}$.

Naive algorithm for RELPRIME

Algorithm 30 (Naive algorithm for RELPRIME)

Input: integers x, y:
Search through all possible divisors of x, y and test divisibility.

- If x, y are give in unary:
  - Size of $\langle x \rangle$ is x
  - Testing all potential divisors of x, y is polynomial
- If x, y are give in binary
  - Size of $\langle x \rangle$ is $\log x$
  - Testing all potential divisors of x, y is exponential
- To analyze the running time of an algorithm that decides a language L, one should first say what is the encoding of L.
- Yet, we sometimes ignore that, and assume “reasonable encoding"
- The above algorithm is sometimes called pseudo polynomial.

Euclid's Algorithm for Computing gcd

Algorithm 31 (E)

On input $\langle x, y \rangle$:

1. Repeat until $x = 0$:
   - 1.1 If $y > x$, swap x and y.
   - 1.2 $x \leftarrow x \bmod y$
2. Output y.

Claim 32

E runs in polynomial-time and correctly computes the gcd function.

Algorithm 33 (M for RELPRIME)

On input $\langle x, y \rangle$:
`Accept` iff $E(x, y) = 1$.

To prove that RELPRIME $\in \mathcal{P}$, we only need to prove Claim 32.

Analyzing Euclid's algorithm

We will only prove the running time part.

Algorithm 34 (E)

On input $\langle x, y \rangle$:

1. Repeat until $x = 0$:
   - 1.1 If $y > x$, swap x and y.
   - 1.2 $x \leftarrow x \bmod y$
2. Output y.

- Each execution of Step 1.1 cuts x by at least half (case analysis for $y < x/2$ and $x/2 \leq y < x$)
- After each two executions maximal value is cut in half

  $\implies$ number of stages is $\min(\log_2 x, \log_2 y)$ $\implies$ total running time is polynomial.

Consequently, RELPRIME $\in \mathcal{P}$.

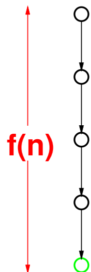
Non-deterministic time

Definition 35 (nondeterministic time)

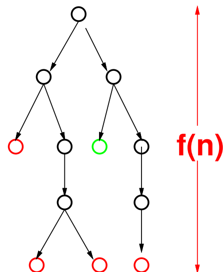
A non-deterministic TM N runs in time $f(n)$, where $f\colon \mathbb{N} \to \mathbb{N}$, if for every input x of length n, the maximum number of steps that N uses on any branch of its computation tree on x, is at most $f(n)$.

Notice that also non-accepting branches must reject within $f(n)$ many steps.

deterministic



nondeterministic



f(n)

TAKE NOTE: the depth of the tree, not the size of the tree!!!

Part III

The Class NP

The Class $\mathcal{NP}$

Definition 36 (NTIME)

For $t\colon \mathbb{N} \to \mathbb{N}$, let $\text{NTIME}(t(n)) =$
$\{L \subseteq \Sigma^*\colon\ L$ is decided by an $O(t(n))$-time single tape NTM$\}$

$\mathcal{NP}$ is the set of languages decidable in polynomial time on non-deterministic TMs.

Definition 37 ($\mathcal{NP}$)

$\mathcal{NP} = \bigcup_{c \geq 0} \text{NTIME}(n^c)$

The class *NP* is important because:

- Insensitive to choice of reasonable non-deterministic computational model.
- Roughly corresponds to problems whose positive solutions are efficiently verified.

Are language in $\mathcal{NP}$ decidable in polynomial time?

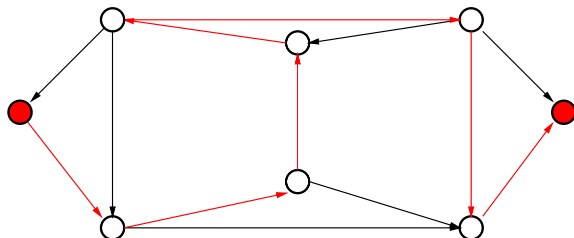
We don't know!

but what we do know is this:

Large class of fundamental languages in $\mathcal{NP}$ that are "the hardest"

(i.e., if ONE is efficiently solvable then ALL are efficiently solvable)

Hamiltonian path



A Hamiltonian path in a directed G visits each node exactly once.

$$\text{HAMPATH} = \{\langle G, s, t\rangle : G \text{ has Hamiltonian path from } s \text{ to } t\}$$

Question 38

How hard is it to decide HAMPATH?

Easy to obtain exponential time algorithm:

- Generate each potential path
- Check whether it is Hamiltonian

HAMPATH $\in \mathcal{NP}$

Here is an NTM that decides HAMPATH in polynomial time.

Algorithm 39 (N)

On input $\langle G = (V, E), s, t \rangle$,

1. Guess a list of numbers $p_1, \ldots, p_m$, where $m = |V|$ and $1 \leq p_i \leq m$.
2. `Accept` if all the following hold (otherwise `Reject`):
   - No repetitions in list
   - $p_1 = s$ and $p_m = t$.
   - $(p_i, p_{i+1}) \in E$ for every $1 \leq i \leq m - 1$

- How does a TM guess a string?

Claim 40

N runs in polynomial time

Verifiability of HAMPATH

This problem has one very interesting feature: polynomial verifiability:

> *We don't know a fast way to find a Hamiltonian path, but we can check whether a given path is Hamiltonian in polynomial time.*

Verifying correctness of a path is much easier than determining whether one exists

Verifiability

Definition 41 (verifier)

A *deterministic* algorithm V is a verifier for a language L, if

- $x \in L \implies \exists c \in \{0,1\}^*$ s.t. $V(x,c) = 1$.
- $x \notin L \implies \nexists c \in \Sigma^*$ s.t. $V(x,c) = 1$.

- The verifier uses the additional information c to verify $x \in L$.
- If V accepts (x, c) (i.e., outputs 1), the string c is called a certificate (also known as, proof or witness) for x.
- A polynomial verifier runs in polynomial time in $|x|$ (i.e., in the length of its left-hand-side input parameter).
- A language L is polynomially verifiable, if it has a polynomial verifier.
- A certificate for $\langle G, s, t\rangle \in$ HAMPATH is simply the Hamiltonian path from s to t.

  Easy to verify in time polynomial in $|\langle G, s, t\rangle|$ whether given path is Hamiltonian.
- Not all languages are known to be polynomially verifiable.

NP and Verifiability

Theorem 42

A language is in $\mathcal{NP}$ iff it has a polynomial time verifier.

Proof's idea:

- The NTM emulates the verifier by guessing the certificate.
- Verifier emulates NTM by using accepting branch as certificate.

Verifiability $\implies \mathcal{NP}$

Claim 43

If L has a poly-time verifier, then it is decided by some polynomial-time NTM.

Proof: Let V be poly-time verifier for L of running time $p(n)$ for some $p \in$ poly.

Algorithm 44 (N)

On input $x \in \{0, 1\}^n$:

1. Guess a string c of length $p(n)$.
2. Emulate V on $\langle x, c \rangle$
3. `Accept` if V accepts; Otherwise `Reject`.

- Why is it suffices to guess a string of length $p(n)$?

$\mathcal{NP} \implies$ Verifiability

Claim 45

If L is decided by a polynomial-time NTM N, then L has a poly-time verifier.

Proof: Assume for simplicity that at each step of N, the number of possible non-deterministic moves is at most two.

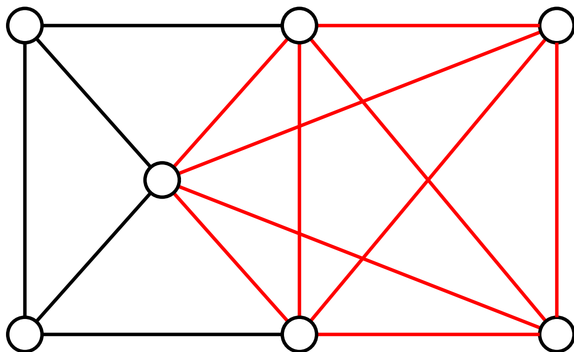
Algorithm 46 (V)

On input (x, c):

1. Emulate $N(x)$, treating each symbol of c as a description of the non-deterministic choice in each step of N.
2. `Accept` if this branch accepts; Otherwise `Reject`.

Without the simplifying assumption?

CLIQUE



A clique in a graph is a subgraph where every two nodes are connected by an edge.

A k-clique is a clique of size k.

Question 47

What is the largest k-clique in the figure?

CLIQUE cont.

$$\text{CLIQUE} = \{\langle G, k\rangle : G \text{ is an undirected graph with a } k\text{-clique}\}$$

Theorem 48

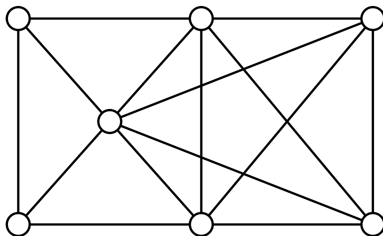
CLIQUE $\in \mathcal{NP}$

Proof's idea: The clique is the certificate.

Algorithm 49 (V)

On input $(\langle G, k\rangle, c)$
`Accept` if c is a k-clique subgraph of G;
Otherwise `Reject`.

Independent set



An independent set in a graph is a set of vertexes, no two of which are linked by an edge.

A k-IS is an independent set of size k.

Question 50

What is the largest k-IS in the figure?

Independent set cont.

$$\text{IND-SET} = \{\langle G, k \rangle : G \text{ contains an independent set of size } k\}$$

Theorem 51

IND-SET $\in \mathcal{NP}$

Proof's idea: The independent set is the certificate.

Algorithm 52 (V)

On input $(\langle G, k \rangle, c)$
`Accept` if c is a k-IS of G (no edges between nodes in c, and $|c| = k$);
Otherwise `Reject`.

The class *co*-$\mathcal{NP}$

$\overline{\text{CLIQUE}} = \{\langle G, k\rangle : G$ is an undirected graph with no k-clique$\}$ seems not to be member of $\mathcal{NP}$.

It seems harder to efficiently verify that something does not exist than to efficiently verify that something does exist.

Definition 53 (*co*-$\mathcal{NP}$)

co-$\mathcal{NP} = \{\mathsf{L} : \overline{\mathsf{L}} \in \mathcal{NP}\}$.

But.. we are not sure...So far, no one knows if *co*-$\mathcal{NP}$ is distinct from $\mathcal{NP}$.

Claim 54

$\mathcal{P} \subseteq$ *co*-$\mathcal{NP}$.

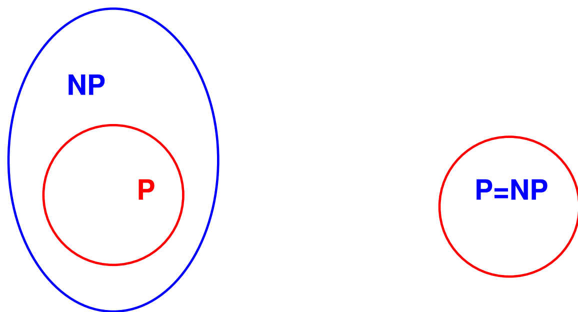
Proof? $\mathsf{L} \in \mathcal{P} \implies \overline{\mathsf{L}} \in \mathcal{P} \implies \overline{\mathsf{L}} \in \mathcal{NP} \implies \mathsf{L} \in$ *co*-$\mathcal{NP}$.

Is Primality in $\mathcal{NP}$? *co*-$\mathcal{NP}$?
How would you prove that a number is prime without trying all divisors?
Actually it is in P! (not obvious at all)

Part IV

P vs. NP

$\mathcal{P}$ vs. $\mathcal{NP}$



The question $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ is one of the great unsolved mysteries in contemporary mathematics.

$\mathcal{P}$ vs. $\mathcal{NP}$

- Most computer scientists believe the two classes are not equal
- Leading computer scientists are not so sure
- Most bogus proofs show them equal (?)
- One of 7 Clay Millenium Prize problems ($1,000,000!)
- “Computer Science’s greatest intellectual export” (Papadimitriou 2007)

$\mathcal{P}$ Vs. $\mathcal{NP}$, cont.

If $\mathcal{P}$ differs from $\mathcal{NP}$, then the distinction between $\mathcal{P}$ and $\mathcal{NP} \setminus \mathcal{P}$ is meaningful and important.

- languages in $\mathcal{P}$ are tractable
- languages in $\mathcal{NP} \setminus \mathcal{P}$ are intractable

Until we can prove that $\mathcal{P} \neq \mathcal{NP}$, there is no hope of proving that a specific language lies in $\mathcal{NP} \setminus \mathcal{P}$.

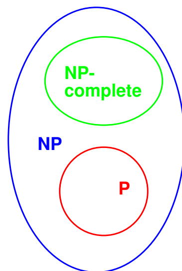
Nevertheless, we can prove statements of the form

If $A \in \mathcal{NP} \setminus \mathcal{P}$, *then* $B \in \mathcal{NP} \setminus \mathcal{P}$.

Part V

NP Completeness

NP Completeness



The class of NP-complete languages are

- "hardest" languages in $\mathcal{NP}$
- If any NP-complete $L \in P$, then $\mathcal{NP} = \mathcal{P}$.

Question 55

Are there NP-complete languages?

Polynomial-time reducibility

Definition 56 (poly-time computable functions)

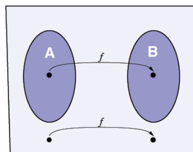
A function $f : \Sigma^* \to \Sigma^*$ is polynomial-time computable, if there is a poly-time deterministic TM that

- starts with input w, and
- halts with $f(w)$ on tape.

Definition 57 (poly-time reduction)

A polynomial-time computable $f : \Sigma^* \to \Sigma^*$ is a poly-time reduction from language A to B, if $x \in A \iff f(x) \in B$ for every $x \in \Sigma^*$.
Is such a reduction from A to B exists, we say that A is poly- time mapping reducible to B, denoted $A \leq_P B$.

The mapping f efficiently converts questions about membership in A to membership in B.



Example: CLIQUE $\leq_P$ IND-SET

Proof:

Definition 58

The complement of a graph $G = (V, E)$ is a graph $G^c = (V, E^c)$, where $E^c = \{(v_1, v_2) \colon v_1, v_2 \in V \text{ and } (v_1, v_2) \notin E\}$.

The reduction $f(G, k)$ from CLIQUE to IND-SET simply computes the complement of the graph and outputs (G^c, k).

f satisfies:

- U is a clique in G $\iff$ U is a independent set in G^c.
- computable in polynomial time!

Remark 59

Same reduction shows that IND-SET $\leq_P$ CLIQUE

Reductions to $\mathcal{P}$

Theorem 60

If $A \leq_P B$ *and* $B \in \mathcal{P}$ *then* $A \in \mathcal{P}$*.*

Proof:

- Let f the reduction from A to B, computed by TM M_f.
  On input x, the TM M_f makes at most $c_f \cdot |x|^{a_f}$ steps.
- Let M_B be the poly-time decider for B.
  On input y, the TM M_B makes at most $c_B \cdot |y|^{a_B}$ steps.

Algorithm 61 (Decider M_A for A)

On input x, return $M_B(f(x))$

- M_A decides A
- Since $|f(x)| \leq c_f|x|^{a_f}$, running time of $M_B(x)$, is at most
  $c_B \cdot (c_f \cdot |x|^{a_f})^{a_B} = (c_B \cdot c_f^{a_B}) \cdot |x|^{a_f \cdot a_B} \in \text{poly}(|x|)$

  Hence, $A \in \mathcal{P}$

What $A \leq_P B$ tells us about B?

Question 62

Assume that $\{0^n1^n \colon n \geq 0\} \leq_P L$. Does it yield that $L \in \mathcal{P}$?

Answer: No. (Reduction in the wrong direction!)

Let $L = H_{TM,\varepsilon}$ and define $f(x) = \begin{cases} M_{stop}, & x \in \{0^n1^n \colon n \in \mathbb{N}\} \\ M_{no-stop}, & \text{otherwise.} \end{cases}$.

$A \leq_P B$ does tell us that B is “at least as hard" as A.

NP completeness, formal definition

Definition 63 ($\mathcal{NP}$-complete)

A language B is NP-complete, if

- B $\in \mathcal{NP}$, and
- Every A $\in \mathcal{NP}$ is poly-time reducible to B (i.e., A $\leq_P$ B)

Let $\mathcal{NPC}$ denote the class of all $\mathcal{NP}$-complete languages.

Compare to

Definition 64 (RE-complete)

A language B is RE-complete, if

- B $\in \mathcal{RE}$, and
- Every A $\in \mathcal{RE}$ is mapping reducible to B.

Why NP completeness?

Theorem 65

If $\mathsf{B} \in \mathcal{NPC}$ *and* $\mathsf{B} \in \mathcal{P}$*, then* $\mathcal{P} = \mathcal{NP}$*.*

Proof: Immediately follows by Thm 63. ♣

To show $\mathcal{P} = \mathcal{NP}$ (and make an instant fortune, see www.claymath.org/millennium/P_vs_NP/), suffices to find a polynomial-time algorithm for any NP-complete problem.

Question 66

Is $\mathcal{NPC}$ empty?

$\mathcal{NPC}$ Is not empty

$A_{NP} = \{\langle M, x, 1^n \rangle : M \text{ is a TM} \wedge \exists c \in \Sigma^* \text{ s.t. } M(x, c) \text{ accepts within } n \text{ steps}\}$.

Theorem 67

$A_{NP} \in \mathcal{NPC}$

Proof:

- Clearly $A_{NP} \in \mathcal{NP}$.
- Let $L \in \mathcal{NP}$, let V be a verifier for L and let $p \in$ poly be a bound on the running time of V (i.e., $V(x, \cdot)$ halts within $p(|x|)$ steps, for every $x \in \Sigma^*$).
- Define $f(x) = \langle V, x, 1^{p(|x|)} \rangle$.
- f is poly-time computable
- $x \in L \iff f(x) \in A_{NP}$.

Finding additional NP-complete languages

Theorem 68

Assume that

1. $B \in \mathcal{NP}$
2. $A \in \mathcal{NPC}$ *and* $A \leq_P B$

then $B \in \mathcal{NPC}$*.*

Proof: Home exercise . . . ♣

We would like to find $L \in \mathcal{NPC}$ that is “natural" and “easy" to reduce to.