Θεωρητική Πληροφορική Ι (ΣΗΜΜΥ)
Αλγόριθμοι & Πολυπλοκότητα ΙΙ (ΜΠΛΑ)

Εργαστήριο Λογικής και Επιστήμης Υπολογισμών
Εθνικό Μετσόβιο Πολυτεχνείο

2013-2014

Πληροφορίες Μαθήματος

Θεωρητική Πληροφορική Ι (ΣΗΜΜΥ)
Αλγόριθμοι & Πολυπλοκότητα ΙΙ (ΜΠλ∀), Λ4-Υπ.

- Διδάσκοντες: Σ. Ζάχος, Ά. Παγουρτζής
- Βοηθός Διδασκαλίας, Επιμέλεια Διαφανειών: Α. Αντωνόπουλος
- Δευτέρα: 17:00 - 19:00 (1.1.31, Παλιά Κτίρια ΗΜΜΥ, ΕΜΠ)
  Πέμπτη: 14:00 - 17:00 (1.1.31, Παλιά Κτίρια ΗΜΜΥ, ΕΜΠ)
- Ώρες Γραφείου: Μετά από κάθε μάθημα, Παρασκευή 10:30-12:30
- Σελίδα: `www.corelab.ntua.gr/courses/grad-algo/`
- Βαθμολόγηση:

Διαγώνισμα:	6 μονάδες
Ασκήσεις:	2 μονάδες
Ομιλία:	2 μονάδες
Quizes :	1 μονάδα

Computation and Reasoning Laboratory
National Technical University of Athens

2013-2014

1st Part

Introduction - Turing Machines - Undecidability - Complexity Classes - Oracles & The Polynomial Hierarchy

Professors:
S. Zachos, Professor
A. Pagourtzis, Ass. Professor

TA-Slides: Antonis Antonopoulos

Bibliography

Textbooks

1. C. Papadimitriou, **Computational Complexity**, Addison Wesley, 1994
2. S. Arora, B. Barak, **Computational Complexity: A Modern Approach**, Cambridge University Press, 2009
3. O. Goldreich, **Computational Complexity: A Conceptual Perspective**, Cambridge University Press, 2008

Lecture Notes

1. L. Trevisan, **Lecture Notes in Computational Complexity**, 2002, UC Berkeley
2. E. Allender, M. Loui, and K. Regan, **Three chapters for the CRC Handbook on Algorithms and Theory of Computation** (M.J. Atallah, ed.), (Boca Raton: CRC Press, 1998).

Contents

- *Computational Complexity*: Quantifying the amount of computational resources required to solve a given task. Classify computational problems according to their inherent difficulty in complexity classes, and prove relations among them.
- *Structural Complexity*: "The study of the relations between various complexity classes and the global properties of individual classes. [...] The goal of structural complexity is a thorough understanding of the relations between the various complexity classes and the internal structure of these complexity classes." [J. Hartmanis]

Decision Problems

- Have answers of the form "*yes*" or "*no*"
- Encoding: each instance x of the problem is represented as a *string* of an alphabet Σ ($|\Sigma| \geq 2$).
- Decision problems have the form "Is x in L?", where L is a *language*, $L \subseteq \Sigma^*$.

- So, for an encoding of the input, using the alphabet Σ, we associate the following language with the decision problem Π:

$L(\Pi) = \{x \in \Sigma^* \mid x \text{ is a representation of a "yes" instance of the problem } \Pi\}$

Example

- Given a number x, is this number prime? ($x \stackrel{?}{\in} \texttt{PRIMES}$)
- Given graph G and a number k, is there a clique with k (or more) nodes in G?

Optimization Problems

- For each instance x there is a **set of Feasible Solutions** $F(x)$.
- To each $s \in F(x)$ we map a positive integer $c(x)$, using **the objective function** $c(s)$.
- We search for the solution $s \in F(x)$ which minimizes (or maximizes) the objective function $c(s)$.

Example

- The **Traveling Salesperson Problem** (TSP):
  Given a finite set $C = \{c_1, \ldots, c_n\}$ of cities and a distance $d(c_i, c_j) \in \mathbb{Z}^+, \forall (c_i, c_j) \in C^2$, we ask for a permutation π of C, that minimizes this quantity:

$$\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)})$$

A Model Discussion

- There are many computational models (RAM, Turing Machines etc).
- The **Church-Turing Thesis** states that all computation models are equivalent. That is, every computation model can be simulated by a Turing Machine.
- In Complexity Theory, we consider **efficiently computable** the problems which are solved (aka the languages that are decided) in **polynomial number of steps** (*Edmonds-Cobham Thesis*).

Efficiently Computable $\equiv$ Polynomial-Time Computable

Contents

Definition

A Turing Machine M is a quintuple $M = (Q, \Sigma, \delta, q_0, F)$:

- $Q = \{q_0, q_1, q_2, q_3, \ldots, q_n, q_{\text{halt}}, q_{\text{yes}}, q_{\text{no}}\}$ is a finite set of states.
- Σ is the alphabet. The tape alphabet is $\Gamma = \Sigma \cup \{\sqcup\}$.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of final states.
- $\delta : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{S, L, R\}$ is the transition function.

- A TM is a "programming language" with a single data structure (a tape), and a cursor, which moves left and right on the tape.
- Function δ is the *program* of the machine.

Turing Machines and Languages

Definition

Let $L \subseteq \Sigma^*$ be a language and M a TM such that, for every string $x \in \Sigma^*$:

- If $x \in L$, then $M(x) =$ "yes"
- If $x \notin L$, then $M(x) =$ "no"

Then we say that M **decides** L.

- We can alternatively say that $M(x) = \chi_L(x)$, where $\chi_L(\cdot)$ is the *characteristic function* of L (if we consider 1 as "yes" and 0 as "no").
- If L is decided by some TM M, then L is called a **recursive language**.

Definition

If for a language L there is a TM M, which if $x \in L$ then $M(x) =$ "yes", and if $x \notin L$ then $M(x) \uparrow$, we call L **recursively enumerable**.

*By $M(x) \uparrow$ we mean that M does not halt on input x (it runs forever).

Theorem

If L is recursive, then it is recursively enumerable.

Proof: *Exercise*

Definition

If for a language L there is a TM M, which if $x \in L$ then $M(x) =$ "yes", and if $x \notin L$ then $M(x) \uparrow$, we call L **recursively enumerable**.

*By $M(x) \uparrow$ we mean that M does not halt on input x (it runs forever).

Theorem

If L is recursive, then it is recursively enumerable.

Proof: *Exercise*

Definition

If f is a function, $f : \Sigma^* \to \Sigma^*$, we say that a TM M computes f if, for any string $x \in \Sigma^*$, $M(x) = f(x)$. If such M exists, f is called a **recursive function**.

- Turing Machines can be thought as algorithms for solving string related problems.

Multitape Turing Machines

- We can extend the previous Turing Machine definition to obtain a Turing Machine with multiple tapes:

Definition

A k-tape Turing Machine M is a quintuple $M = (Q, \Sigma, \delta, q_0, F)$:

- $Q = \{q_0, q_1, q_2, q_3, \ldots, q_n, q_{\text{halt}}, q_{\text{yes}}, q_{\text{no}}\}$ is a finite set of states.
- Σ is the alphabet. The tape alphabet is $\Gamma = \Sigma \cup \{\sqcup\}$.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of final states.
- $\delta : (Q \setminus F) \times \Gamma^k \to Q \times (\Gamma \times \{S, L, R\})^k$ is the transition function.

Bounds on Turing Machines

- We will characterize the "performance" of a Turing Machine by the amount of *time* and *space* required on instances of size n, when these amounts are expressed as a function of n.

Definition

Let $T : \mathbb{N} \to \mathbb{N}$. We say that machine M operates within time $T(n)$ if, for any input string x, the time required by M to reach a final state is at most $T(|x|)$. Function T is a **time bound** for M.

Definition

Let $S : \mathbb{N} \to \mathbb{N}$. We say that machine M operates within space $S(n)$ if, for any input string x, M visits at most $S(|x|)$ locations on its work tapes (excluding the input tape) during its computation. Function S is a **space bound** for M.

Multitape Turing Machines

Theorem

Given any k-tape Turing Machine M operating within time $T(n)$, we can construct a TM M' operating within time $\mathcal{O}\left(T^2(n)\right)$ such that, for any input $x \in \Sigma^*$, $M(x) = M'(x)$.

Proof: See Th.2.1 (p.30) in [1].

- This is a strong evidence of the robustness of our model: *Adding a bounded number of strings does not increase their computational capabilities, and affects their efficiency only polynomially*.

Linear Speedup

Theorem

Let M be a TM that decides $L \subseteq \Sigma^*$, that operates within time $T(n)$. Then, for every $\varepsilon > 0$, there is a TM M' which decides the same language and operates within time $T'(n) = \varepsilon T(n) + n + 2$.

Proof: See Th.2.2 (p.32) in [1].

- If, for example, T is linear, i.e. something like cn, then this theorem states that the constant c can be made arbitrarily close to 1. So, it is fair to start using the $\mathcal{O}(\cdot)$ notation in our time bounds.
- A similar theorem holds for space:

Theorem

Let M be a TM that decides $L \subseteq \Sigma^*$, that operates within space $S(n)$. Then, for every $\varepsilon > 0$, there is a TM M' which decides the same language and operates within space $S'(n) = \varepsilon S(n) + 2$.

Nondeterministic Turing Machines

- We will now introduce an **unrealistic** model of computation:

Definition

A Turing Machine M is a quintuple $M = (Q, \Sigma, \delta, q_0, F)$:

- $Q = \{q_0, q_1, q_2, q_3, \ldots, q_n, q_{\text{halt}}, q_{\text{yes}}, q_{\text{no}}\}$ is a finite set of states.
- Σ is the alphabet. The tape alphabet is $\Gamma = \Sigma \cup \{\sqcup\}$.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of final states.
- $\delta : (Q \setminus F) \times \Gamma \to Pow(Q \times \Gamma \times \{S, L, R\})$ is the transition **relation**.

Nondeterministic Turing Machines

- In this model, an input is accepted if there is *some sequence* of nondeterministic choices that results in "yes".
- An input is rejected if there is *no sequence* of choices that lead to acceptance.
- Observe the similarity with recursively enumerable languages.

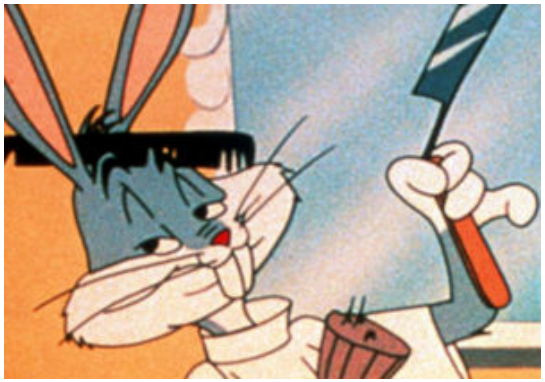
Definition

We say that M operates within bound $T(n)$, if for every input $x \in \Sigma^*$ and every sequence of nondeterministic choices, M reaches a final state within $T(|x|)$ steps.

- The above definition requires that M does not have computation paths longer than $T(n)$, where $n = |x|$ the length of the input.
- The amount of time charged is the *depth* of the **computation tree**.

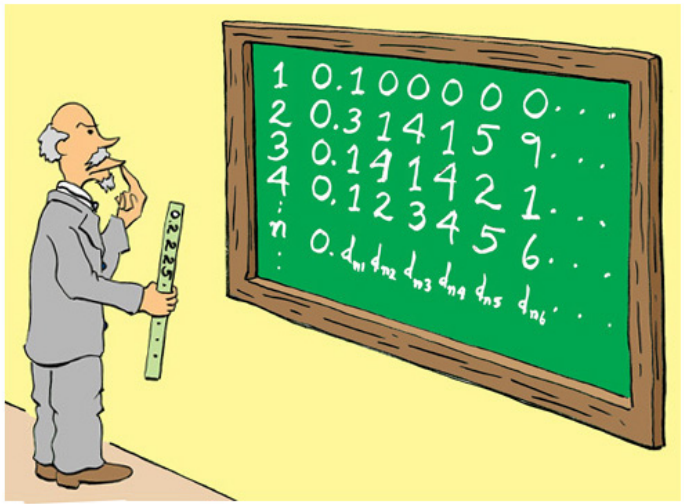
Contents

Diagonalization



Suppose there is a town with just one barber, who is male. In this town, the barber shaves all those, and only those, men in town who do not shave themselves. Who shaves the barber?

Diagonalization is a technique that was used in many different cases:



George showed it wouldn't fit in.

Diagonalization

Theorem

The functions from $\mathbb{N}$ to $\mathbb{N}$ are uncountable.

Proof: Let, for the sake of contradiction that are countable: $\phi_1, \phi_2, \ldots$. Consider the following function: $f(x) = \phi_x(x) + 1$. This function must appear somewhere in this enumeration, so let $\phi_y = f(x)$. Then $\phi_y(x) = \phi_x(x) + 1$, and if we choose y as an argument, then $\phi_y(y) = \phi_y(y) + 1$. $\square$

Machines as strings

- It is obvious that we can represent a Turing Machine as a string: *just write down the description and encode it using an alphabet, e.g.* $\{0, 1\}$.
- We denote by $\llcorner M \lrcorner$ the TM M's representation as a string.
- Also, if $x \in \Sigma^*$, we denote by M_x the TM that x represents.

Keep in mind that:

- **Every string represents some Turing Machine.**
- **Every TM is represented by infinitely many strings.**

The Universal Turing Machine

- So far, our computational models are specified to solve a single problem.
- Turing observed that there is a TM that can simulate any other TM M, given M's description as input.

Theorem

There exists a TM $\mathcal{U}$ such that for every $x, w \in \Sigma^*$, $\mathcal{U}(x, w) = M_w(x)$.
Also, if M_w halts within T steps on input x, then $\mathcal{U}(x, w)$ halts within $CT \log T$ steps, where C is a constant indepedent of x, and depending only on M_w's alphabet size number of tapes and number of states.

Proof: See section 3.1 in [1], and Th. 1.9 and section 1.7 in [2].

The Halting Problem

- Consider the following problem: "*Given the description of a TM M, and a string x, will M halt on input x?* " This is called the HALTING PROBLEM.
- **We want to compute this problem ! ! !** (Given a computer program and an input, will this program enter an infinite loop?)
- In language form: $\mathtt{H} = \{\llcorner M \lrcorner; x \mid M(x) \downarrow\}$, where " $\downarrow$ " means that the machine halts, and " $\uparrow$ " that it runs forever.

Theorem

H *is recursively enumerable.*

Proof: See Th.3.1 (p.59) in [1]

- In fact, H is not just a recursively enumerable language:
  If we had an algorithm for deciding H, then we would be able to derive an algorithm for deciding any r.e. language (**RE**-complete).

The Halting Problem

- But....

Theorem

H *is not recursive.*

Proof: See Th.3.1 (p.60) in [1]

- Suppose, for the sake of contradiction, that there is a TM M_H that decides H.
- Consider the TM D:

$$D(\llcorner M \lrcorner) : \textbf{ if } M_H(\llcorner M \lrcorner; \llcorner M \lrcorner) = \text{“yes”} \textbf{ then } \uparrow \textbf{ else } \text{“yes”}$$

- What is $D(\llcorner D \lrcorner)$?

The Halting Problem

- But....

Theorem

H *is not recursive.*

Proof: See Th.3.1 (p.60) in [1]

- Suppose, for the sake of contradiction, that there is a TM M_H that decides H.
- Consider the TM D:

$$D(\llcorner M \lrcorner) : \textbf{ if } M_H(\llcorner M \lrcorner; \llcorner M \lrcorner) = \text{“yes”} \textbf{ then } \uparrow \textbf{ else } \text{“yes”}$$

- What is $D(\llcorner D \lrcorner)$?
- If $D(\llcorner D \lrcorner) \uparrow$, then M_H accepts the input, so $\llcorner D \lrcorner; \llcorner D \lrcorner \in \mathtt{H}$, so $D(D) \downarrow$.
- If $D(\llcorner D \lrcorner) \downarrow$, then M_H rejects $\llcorner D \lrcorner; \llcorner D \lrcorner$, so $\llcorner D \lrcorner; \llcorner D \lrcorner \notin \mathtt{H}$, so $D(D) \uparrow$. $\square$

- Recursive languages are a *proper* subset of recursive enumerable ones.
- Recall that the complement of a language L is defined as:

$$\overline{L} = \{x \in \Sigma^* \mid x \notin L\} = \Sigma^* \setminus L$$

Theorem

1. *If L is recursive, so is $\overline{L}$.*
2. *L is recursive if and only if L and $\overline{L}$ are recursively enumerable.*

Proof: Exercise

More Undecidability

- The HALTING PROBLEM, our first undecidable problem, was the first, but not the only undecidable problem. Its spawns a wide range of such problems, via *reductions*.
- To show that a problem A is undecidable we establish that, if there is an algorithm for A, then there would be an algorithm for H, which is absurd.

Theorem

The following languages are not recursive:

1. $\{M \mid M \text{ halts on all inputs}\}$
2. $\{M; x \mid \text{There is a } y \text{ such that } M(x) = y\}$
3. $\{M; x \mid \text{The computation of } M \text{ uses all states of } M\}$
4. $\{M; x; y \mid M(x) = y\}$

Rice's Theorem

- The previous problems lead us to a more general conlusion:

Any non-trivial property of Turing Machines is undecidable

- If a TM M accepts a language L, we write $L = L(M)$:

Theorem (Rice's Theorem)

Suppose that $\mathcal{C}$ is a proper, non-empty subset of the set of all recursively enumerable languages. Then, the following problem is undecidable:

Given a Turing Machine M, is $L(M) \in \mathcal{C}$?

Rice's Theorem

Proof:

See Th.3.2 (p.62) in [1]

- We can assume that $\emptyset \notin \mathcal{C}$ (*why?*).
- Since $\mathcal{C}$ is nonempty, $\exists\ L \in \mathcal{C}$, accepted by the TM M_L.
- Let M_H the TM deciding the HALTING PROBLEM for an arbitrary input x. For each $x \in \Sigma^*$, we construct a TM M as follows:

  $M(y)$: **if** $M_H(x) =$ "yes" **then** $M_L(y)$ **else** $\uparrow$

- We claim that: $L(M) \in \mathcal{C}$ if and only if $x \in \mathtt{H}$.

Rice's Theorem

Proof: See Th.3.2 (p.62) in [1]

- We can assume that $\emptyset \notin \mathcal{C}$ (*why?*).
- Since $\mathcal{C}$ is nonempty, $\exists\ L \in \mathcal{C}$, accepted by the TM M_L.
- Let M_H the TM deciding the HALTING PROBLEM for an arbitrary input x. For each $x \in \Sigma^*$, we construct a TM M as follows:

  $M(y)$: **if** $M_H(x) =$ "yes" **then** $M_L(y)$ **else** $\uparrow$

- We claim that: $L(M) \in \mathcal{C}$ if and only if $x \in \mathrm{H}$.
  Proof of the claim:
  - If $x \in \mathrm{H}$, then $M_H(x) =$ "yes", and so M will accept y or never halt, depending on whether $y \in L$. Then the language accepted by M is exactly L, which is in $\mathcal{C}$.
  - If $M_H(x) \uparrow$, M never halts, and thus M accepts the language $\emptyset$, which is not in $\mathcal{C}$. $\square$

Contents

Parameters used to define complexity classes:

- Model of Computation (Turing Machine, RAM, Circuits)
- Mode of Computation (Deterministic, Nondeterministic, Probabilistic)
- Complexity Measures (*Time, Space, Circuit Size-Depth*)
- Other Parameters (Randomization, Interaction)

Our first complexity classes

Definition

Let $L \subseteq \Sigma^*$, and $T, S : \mathbb{N} \to \mathbb{N}$:

- We say that $L \in \mathbf{DTIME}[T(n)]$ if there exists a TM M deciding L, which operates within the *time* bound $\mathcal{O}(T(n))$, where $n = |x|$.
- We say that $L \in \mathbf{DSPACE}[S(n)]$ if there exists a TM M deciding L, which operates within *space* bound $\mathcal{O}(S(n))$, that is, for any input x, requires space at most $S(|x|)$.
- We say that $L \in \mathbf{NTIME}[T(n)]$ if there exists a *nondeterministic* TM M deciding L, which operates within the time bound $\mathcal{O}(T(n))$.
- We say that $L \in \mathbf{NSPACE}[S(n)]$ if there exists a *nondeterministic* TM M deciding L, which operates within space bound $\mathcal{O}(S(n))$.

Our first complexity classes

- The above are **Complexity Classes**, in the sense that they are sets of languages.
- All these classes are parameterized by a function T or S, so they are *families* of classes (for each function we obtain a complexity class).

Definition (Complement of a complexity class)

For any complexity class $\mathcal{C}$, $co\mathcal{C}$ denotes the class: $\{\overline{L} \mid L \in \mathcal{C}\}$, where $\overline{L} = \Sigma^* \setminus L = \{x \in \Sigma^* \mid x \notin L\}$.

- We want to define "reasonable" complexity classes, in the sense that we want to "compute more problems", given more computational resources.

Constructible Functions

- Can we use all computable functions to define Complexity Classes?

Theorem (Gap Theorem)

For any computable functions r and a, there exists a computable function f such that $f(n) \geq a(n)$, and

$$\mathbf{DTIME}[f(n)] = \mathbf{DTIME}[r(f(n))]$$

- That means, for $r(n) = 2^{2^{f(n)}}$, the incementation from $f(n)$ to $2^{2^{f(n)}}$ does not allow the computation of any new function!
- So, we must use some restricted families of functions:

Constructible Functions

Definition (Time-Constructible Function)

A nondecreasing function $T : \mathbb{N} \to \mathbb{N}$ is **time constructible** if $T(n) \geq n$ and there is a TM M that computes the function $x \mapsto \llcorner T(|x|) \lrcorner$ in time $T(n)$.

Definition (Space-Constructible Function)

A nondecreasing function $S : \mathbb{N} \to \mathbb{N}$ is **space-constructible** if $S(n) > \log n$ and there is a TM M that computes $S(|x|)$ using $S(|x|)$ space, given x as input.

- The restriction $T(n) \geq n$ is to allow the machine to read its input.
- The restriction $S(n) > \log n$ is to allow the machine to "remember" the index of the cell of the input tape that it is currently reading.
- Also, if $f_1(n)$, $f_2(n)$ are time/space-constructible functions, so are $f_1 + f_2$, $f_1 \cdot f_2$ and $f_1^{f_2}$.

Constructible Functions

Theorem (Hierarchy Theorems)

Let t_1, t_2 be time-constructible functions, and s_1, s_2 be space-constructible functions. Then:

1. *If $t_1(n) \log t_1(n) = o(t_2(n))$, then* $\mathbf{DTIME}(t_1) \subsetneq \mathbf{DTIME}(t_2)$.
2. *If $t_1(n+1) = o(t_2(n))$, then* $\mathbf{NTIME}(t_1) \subsetneq \mathbf{NTIME}(t_2)$.
3. *If $s_1(n) = o(s_2(n))$, then* $\mathbf{DSPACE}(s_1) \subsetneq \mathbf{DSPACE}(s_2)$.
4. *If $s_1(n) = o(s_2(n))$, then* $\mathbf{NSPACE}(s_1) \subsetneq \mathbf{NSPACE}(s_2)$.

Complexity Classes

Simplified Case of Deterministic Time Hierarchy Theorem

Theorem

$\mathbf{DTIME}[n] \subsetneq \mathbf{DTIME}[n^{1.5}]$

Simplified Case of Deterministic Time Hierarchy Theorem

Theorem

$\mathbf{DTIME}[n] \subsetneq \mathbf{DTIME}[n^{1.5}]$

Proof (*Diagonalization*): See Th.3.1 (p.69) in [2]

Let D be the following machine:

```
On input x, run for |x|^1.4 steps U(M_x, x);
If U(M_x, x) = b, then return 1 − b;
Else return 0;
```

- Clearly, $L = L(D) \in \mathbf{DTIME}[n^{1.5}]$
- We claim that $L \notin \mathbf{DTIME}[n]$:
  Let $L \in \mathbf{DTIME}[n] \Rightarrow \exists\ M :\ M(x) = D(x)\ \forall x \in \Sigma^*$, and M works for $\mathcal{O}(x)$ steps.
  The time to simulate M using $\mathcal{U}$ is $c|x| \log |x|$, for some c.

Simplified Case of Deterministic Time Hierarchy Theorem

Proof (*cont'd*):
$\exists n_0 : \ n^{1.4} > cn \log n \ \forall n \geq n_0$
There exists a x_M, s.t. $x_M = \llcorner M \lrcorner$ and $|x_M| > n_0$ (*why?*) Then,
$D(x_M) = 1 - M(x_M)$ (while we have also that $D(x) = M(x), \ \forall x$)

Simplified Case of Deterministic Time Hierarchy Theorem

Proof (*cont'd*):
$\exists n_0 : \ n^{1.4} > cn \log n \ \forall n \geq n_0$
There exists a x_M, s.t. $x_M = \llcorner M \lrcorner$ and $|x_M| > n_0$ (*why?*) Then,
$\mathbf{D(x_M) = 1 - M(x_M)}$ (while we have also that $D(x) = M(x), \ \forall x$)
Contradiction!! $\square$

Simplified Case of Deterministic Time Hierarchy Theorem

Proof (*cont'd*):
$\exists n_0 : \ n^{1.4} > cn \log n \ \forall n \geq n_0$
There exists a x_M, s.t. $x_M = \llcorner M \lrcorner$ and $|x_M| > n_0$ (*why?*) Then,
$\mathbf{D(x_M) = 1 - M(x_M)}$ (while we have also that $D(x) = M(x), \ \forall x$)
Contradiction!! $\square$

- So, we have the hierachy:

$$\mathbf{DTIME}[n] \subsetneq \mathbf{DTIME}[n^2] \subsetneq \mathbf{DTIME}[n^3] \subsetneq \cdots$$

- We will later see that the class containing the problems we can efficiently solve (recall the Edmonds-Cobham Thesis) is the class $\mathbf{P} = \bigcup_{c \in \mathbb{N}} \mathbf{DTIME}[n^c]$.

- Hierarchy Theorems tell us how classes of the same kind relate to each other, when we vary the complexity bound.
- The most interesting results concern relationships between classes of different kinds:

Theorem

Suppose that $T(n), S(n)$ are time-constructible and space-constructible functions, respectively.Then:

1. $\mathbf{DTIME}[T(n)] \subseteq \mathbf{NTIME}[T(n)]$
2. $\mathbf{DSPACE}[S(n)] \subseteq \mathbf{NSPACE}[S(n)]$
3. $\mathbf{NTIME}[T(n)] \subseteq \mathbf{DSPACE}[T(n)]$
4. $\mathbf{NSPACE}[S(n)] \subseteq \mathbf{DTIME}[k^{\log n + S(n)}]$

Corollary

$$\mathbf{NTIME}[T(n)] \subseteq \bigcup_{c>1} \mathbf{DTIME}[c^{T(n)}]$$

Proof: See Th.7.4 (p.147) in [1]

1. Trivial
2. Trivial
3. We can simulate the machine for each nondeterministic choice, using at most $T(n)$ steps in each simulation. There are *exponentially* many simulations, but we can simulate them one-by-one, reusing the same space.
4. Recall the notion of a configuration of a TM: For a k-tape machine, is a $2k-2$ tuple: $(q, i, w_2, u_2, \ldots, w_{k-1}, u_{k-1})$
   How many configurations are there?
   - $|Q|$ choices for the state
   - $n+1$ choices for i, and
   - Fewer than $|\Sigma|^{(2k-2)S(n)}$ for the remaining strings

   So, the total number of configurations on input size n is at most $nc_1 = c_1^{\log n + S(n)}$

Proof (*cont'd*):

Definition (Configuration Graph of a TM)

The configuration graph of M on input x, denoted $G(M, x)$, has as **vertices** all the possible configurations, and there is an **edge** between two vertices C and C' if and only if C' can be reached from C in one step, according to M's transition function.

- So, we have reduced this simulation to REACHABILITY* problem (also known as S-T CONN), for which we know there is a poly-time ($\mathcal{O}\left(n^2\right)$) algorithm.
- So, the simulation takes $c_2 c_1^{2(\log n + S(n))} \sim k^{\log n + S(n)}$ steps. $\square$

*REACHABILITY: Given a graph G and two nodes $v_1, v_n \in V$, is there a path from v_1 to v_n?

The essential Complexity Hierarchy

Definition

$$\mathbf{L} = \mathbf{DSPACE}[\log n]$$

$$\mathbf{NL} = \mathbf{NSPACE}[\log n]$$

$$\mathbf{P} = \bigcup_{c \in \mathbb{N}} \mathbf{DTIME}[n^c]$$

$$\mathbf{NP} = \bigcup_{c \in \mathbb{N}} \mathbf{NTIME}[n^c]$$

$$\mathbf{PSPACE} = \bigcup_{c \in \mathbb{N}} \mathbf{DSPACE}[n^c]$$

$$\mathbf{NPSPACE} = \bigcup_{c \in \mathbb{N}} \mathbf{NSPACE}[n^c]$$

The essential Complexity Hierarchy

Definition

$$\mathbf{EXP} = \bigcup_{c \in \mathbb{N}} \mathbf{DTIME}[2^{n^c}]$$

$$\mathbf{NEXP} = \bigcup_{c \in \mathbb{N}} \mathbf{NTIME}[2^{n^c}]$$

$$\mathbf{EXPSPACE} = \bigcup_{c \in \mathbb{N}} \mathbf{DSPACE}[2^{n^c}]$$

$$\mathbf{NEXPSPACE} = \bigcup_{c \in \mathbb{N}} \mathbf{NSPACE}[2^{n^c}]$$

The essential Complexity Hierarchy

Definition

$$\mathbf{EXP} = \bigcup_{c \in \mathbb{N}} \mathbf{DTIME}[2^{n^c}]$$

$$\mathbf{NEXP} = \bigcup_{c \in \mathbb{N}} \mathbf{NTIME}[2^{n^c}]$$

$$\mathbf{EXPSPACE} = \bigcup_{c \in \mathbb{N}} \mathbf{DSPACE}[2^{n^c}]$$

$$\mathbf{NEXPSPACE} = \bigcup_{c \in \mathbb{N}} \mathbf{NSPACE}[2^{n^c}]$$

$$\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{NPSPACE} \subseteq \mathbf{EXP} \subseteq \mathbf{NEXP}$$

Certificate Characterization of NP

Definition

Let $R \subseteq \Sigma^* \times \Sigma^*$ a binary relation on strings.

- R is called **polynomially decidable** if there is a DTM deciding the language $\{x; y \mid (x, y) \in R\}$ in polynomial time.
- R is called **polynomially balanced** if $(x, y) \in R$ implies $|y| \leq |x|^k$, for some $k \geq 1$.

Theorem

*Let $L \subseteq \Sigma^*$ be a language. $L \in$ **NP** if and only if there is a polynomially decidable and polynomially balanced relation R, such that:*

$$L = \{x \mid \exists y \ R(x, y)\}$$

- This y is called **succinct certificate**, or **witness**.

Proof: See Pr.9.1 (p.181) in [1]

($\Leftarrow$) If such an R exists, we can construct the following NTM deciding L:
"On input x, *guess* a y, such that $|y| \leq |x|^k$, and then test (in poly-time) if $(x, y) \in R$. If so, accept, else reject." Observe that an accepting computation exists if and only if $x \in L$.

($\Rightarrow$) If $L \in \mathbf{NP}$, then $\exists$ an NTM N that decides L in time $|x|^k$, for some k. Define the following R:
"$(x, y) \in R$ if and only if y is an **encoding** of an accepting computation of $N(x)$."
R is polynomially balanced and decidable (*why?*), so, given by assumption that N decides L, we have our conclusion. $\square$

Can creativity be automated?

As we saw:

- Class **P**: Efficient *Computation*
- Class **NP**: Efficient *Verification*
- So, if we can efficiently verify a mathematical proof, can we create it efficiently?

If $P = NP$...

- For every mathematical statement, and given a page limit, we would (quickly) generate a proof, if one exists.
- Given detailed constraints on an engineering task, we would (quickly) generate a design which meets the given criteria, if one exists.
- Given data on some phenomenon and modeling restrictions, we would (quickly) generate a theory to explain the date, if one exists.

See "A. Wigderson: Knowledge, Creativity and **P** versus **NP**"

Complements of complexity classes

- Deterministic complexity classes are in general closed under complement ($co\mathbf{L} = \mathbf{L}$, $co\mathbf{P} = \mathbf{P}$, $co\mathbf{PSPACE} = \mathbf{PSPACE}$).
- Complements of non-deterministic complexity classes are very interesting:
- The class $co\mathbf{NP}$ contains all the languages that have **succinct disqualifications** (the analogue of *succinct certificate* for the class **NP**). The "no" instance of a problem in $co\mathbf{NP}$ has a short proof of its being a "no" instance.
- So:

$$\mathbf{P} \subseteq \mathbf{NP} \cap co\mathbf{NP}$$

- Note the *similarity* and the *difference* with $\mathbf{R} = \mathbf{RE} \cap co\mathbf{RE}$.

Quantifier Characterization of Complexity Classes

Definition

We denote as $\mathcal{C} = (Q_1/Q_2)$, where $Q_1, Q_2 \in \{\exists, \forall\}$, the class $\mathcal{C}$ of languages L satisfying:

- $x \in L \Rightarrow Q_1 y \ R(x, y)$
- $x \notin L \Rightarrow Q_2 y \ \neg R(x, y)$

- $\mathbf{P} = (\forall/\forall)$
- $\mathbf{NP} = (\exists/\forall)$
- $co\mathbf{NP} = (\forall/\exists)$

Savitch's Theorem

- REACHABILITY $\in$ **NL**. See Ex.2.10 (p.48) in [1]

Theorem (Savitch's Theorem)

REACHABILITY $\in$ **DSPACE**$[\log^2 n]$

Proof: See Th.7.4 (p.149) in [1]

$PATH(x, y, i)$: "*There is a path from* x *to* y*, of length* $\leq 2^i$" .

- We can solve REACHABILITY if we can compute $PATH(x, y, \lceil \log n \rceil)$, for any nodes $x, y \in V$, since any path in G can be at most $n \log n$ long.
- If $i = 0$, we can check whether $PATH(x, y, i)$.
- If $i \geq 1$:

forall nodes z test whether $PATH(x, z, i-1)$ and $PATH(z, y, i-1)$

Savitch's Theorem

Proof (*cont'd*):

- We generate all nodes z one after the other, *reusing* space.
- Once a z is generated, we add $(x, z, i-1)$ to the tape, and start working on this recursively.
- If a negative answer is obtained to $PATH(x, z, i-1)$, we erase this triple and move to the next z.
- If a positive answer is obtained to $PATH(x, z, i-1)$, we erase the triple and move to $PATH(z, y, i-1)$.
- If this is negative, we erase it and move to the next z.
- If it is positive, we compare it to (x, y, i) to check that this is the second recursive call, and then return a positive answer to $PATH(x, y, i)$.
- The work tape contains at any moment at most $\lceil \log n \rceil$, each of length at most $3 \log n$. $\square$

Savitch's Theorem

Corollary

NSPACE$[S(n)] \subseteq$ **DSPACE**$[S^2(n)]$, for any space-constructible function $S(n) \geq \log n$.

Proof:

- Let M be the nondeterministic TM to be simulated.
- We run the algorithm of Savitch's Theorem proof on the configuration graph of M on input x.
- Since the configuration graph has $c^{S(n)}$ nodes, $\mathcal{O}\left(S^2(n)\right)$ space suffices. $\square$

Corollary

PSPACE $=$ **NPSPACE**

NL-Completeness

- In Complexity Theory, we "connect" problems in a complexity class with partial ordering relations, called **reductions**, which formalize the notion of "*a problem that is at least as hard as another*".
- A reduction must be computationally weaker than the class in which we use it.

Definition

A language L_1 is **logspace reducible** to a language L_2, denoted $L_1 \leq_l L_2$, if there is a function $f : \Sigma^* \to \Sigma^*$, computable by a DTM in $\mathcal{O}(\log n)$ space, such that for all $x \in \Sigma^*$:

$$x \in L_1 \Leftrightarrow f(x) \in L_2$$

We say that a language L is **NL**-complete if it is in **NL** and for every $A \in \mathbf{NL}$, $A \leq_l L$.

NL-Completeness

Theorem

REACHABILITY is **NL***-complete.*

NL-Completeness

Theorem

REACHABILITY is **NL***-complete.*

Proof: See Th.4.18 (p.89) in [2]

- We 've argued why REACHABILITY $\in$ **NL**.
- Let $L \in$ **NL**, that is, it is decided by a $\mathcal{O}(\log n)$ NTM N.
- Given input x, we can construct the *configuration graph* of $N(x)$.
- We can assume that this graph has a *single* accepting node.
- We can construct this in logspace: Given configurations C, C' we can in space $\mathcal{O}(|C| + |C'|) = \mathcal{O}(\log |x|)$ check the graph's adjacency matrix if they are connected by an edge.
- It is clear that $x \in L$ if and only if the produced instance of REACHABILITY has a "yes" answer. $\square$

Certificate Definition of NL

- We want to give a characterization of **NL**, similar to the one we gave for **NP**.
- A certificate may be polynomially long, so a logspace machine may not have the space to store it.
- So, we will assume that the certificate is provided to the machine on a separate tape that is **read once**.

Certificate Definition of NL

Definition

A language L is in **NL** if there exists a deterministic TM M with an additional special read-once input tape, such that for every $x \in \Sigma^*$:

$$x \in L \Leftrightarrow \exists y, |y| \in poly(|x|), M(x, y) = 1$$

where by $M(x, y)$ we denote the output of M where x is placed on its input tape, and y is placed on its special read-once tape, and M uses at most $\mathcal{O}(\log |x|)$ space on its read-write tapes for every input x.

- What if remove the read-once restriction and allow the TM's head to move back and forth on the certificate, and read each bit multiple times?

Immerman-Szelepscényi

Theorem (The Immerman-Szelepscényi Theorem)

$\overline{\texttt{REACHABILITY}} \in \mathbf{NL}$

Immerman-Szelepscényi

Theorem (The Immerman-Szelepscényi Theorem)

$\overline{\texttt{REACHABILITY}} \in \mathbf{NL}$

Proof: See Th.4.20 (p.91) in [2]

- It suffices to show a $\mathcal{O}(\log n)$ verification algorithm A such that: $\forall\ (G, s, t)$, $\exists$ a polynomial certificate u such that: $A((G, s, t), u) =$ "yes" iff t is not reachable from s.
- A has read-once access to u.
- G's vertices are identified by numbers in $\{1, \ldots, n\} = [n]$
- C_i: "*The set of vertices reachable from s in $\leq i$ steps.*"
- Membership in C_i is easily certified:
- $\forall i \in [n]$: $v_0, \ldots, v_k$ along the path from s to v, $k \leq i$.
- The certificate is at most polynomial in n.

The Immerman-Szelepscényi Theorem

Proof (*cont'd*):

- We can check the certificate using read-once access:
  1. $v_0 = s$
  2. for $j > 0$, $(v_{j-1}, v_j) \in E(G)$
  3. $v_k = v$
  4. Path ends within at most i steps
- We now construct two types of certificates:
  1. A certificate that a vertex $v \notin C_i$, given $|C_i|$.
  2. A certificate that $|C_i| = c$, for some c, given $|C_{i-1}|$.
- Since $C_0 = \{s\}$, we can provide the 2nd certificate to convince the verifier for the sizes of $C_1, \ldots, C_n$
- C_n is the set of vertices *reachable* from s.

The Immerman-Szelepscényi Theorem

Proof (*cont'd*):

- Since the verifier has been convinced of $|C_n|$, we can use the 1st type of certificate to convince the verifier that $t \notin C_n$.
- **Certifying that $v \notin C_i$, given $|C_i|$**
  The certificate is the list of certificates that $u \in C_i$, for every $u \in C_i$.
  The verifier will check:
  1. Each certificate is valid
  2. Vertex u, given a certificate for u, is larger than the previous.
  3. No certificate is provided for v.
  4. The total number of certificates is exactly $|C_i|$.

The Immerman-Szelepscényi Theorem

Proof (*cont'd*):
Certifying that $v \notin C_i$, given $|C_{i-1}|$
The certificate is the list of certificates that $u \in C_{i-1}$, for every $u \in C_{i-1}$
The verifier will check:

1. Each certificate is valid
2. Vertex u, given a certificate for u, is larger than the previous.
3. No certificate is provided for v *or for a neighbour of* v.
4. The total number of certificates is exactly $|C_{i-1}|$.

Certifying that $|C_i| = c$, given $|C_{i-1}|$
The certificate will consist of n certificates for each vertex.
The verifier will check all certificates, and count the vertices that have been certified to be in C_i. If $|C_i| = c$, it accepts. $\square$

The Immerman-Szelepscényi Theorem

Corollary

For every space constructible $S(n) > \log n$:

$$\mathbf{NSPACE}[S(n)] = co\mathbf{NSPACE}[S(n)]$$

Proof:

- Let $L \in \mathbf{NSPACE}[S(n)]$. We will show that $\exists$ $S(n)$ space-bounded NTM $\overline{M}$ deciding $\overline{L}$:
- $\overline{M}$ on input x uses the above certification procedure on the *configuration graph* of M. $\square$

Corollary

$$\mathbf{NL} = co\mathbf{NL}$$

What about Undirected Reachability?

- `UNDIRECTED REACHABILITY` captures the phenomenon of configuration graphs with both directions.
- H. Lewis and C. Papadimitriou defined the class **SL** (**S**ymmetric **L**ogspace) as the class of languages decided by a **Symmetric Turing Machine** using logarithmic space.
- Obviously,

$$\boxed{\mathbf{L} \subseteq \mathbf{SL} \subseteq \mathbf{NL}}$$

- As in the case of **NL**, `UNDIRECTED REACHABILITY` is **SL**-complete.
- But in 2004, Omer Reingold showed, using expander graphs, a deterministic logspace algorithm for `UNDIRECTED REACHABILITY`, so:

Theorem (Reigold, 2004)

$$\mathbf{L} = \mathbf{SL}$$

Our Complexity Hierarchy Landscape