Computational Complexity; slides 3, HT 2022
Deterministic complexity classes

Prof. Paul W. Goldberg (Dept. of Computer Science,
University of Oxford)

HT 2022

Measuring Complexity

Our general interest: detailed classification of decidable languages.

Goal: Classify languages according to the amount of resources needed to solve them.

Resources: In this lecture we will primarily consider

- **time** – the running time of algorithms (steps on a Turing machine)
- **space** – the amount of additional memory needed (cells on the Turing tapes)

Next: basic complexity classes, polynomial-time reductions

Measuring Complexity

Definition.

Let M be a Turing acceptor and let $S, T : \mathbb{N} \to \mathbb{N}$ be functions.

1. M is *T-time bounded* if it halts on every input $w \in \Sigma^*$ after $\leq T(|w|)$ steps.
2. M is *S-space bounded* if it halts on every input $w \in \Sigma^*$ using $\leq S(|w|)$ cells on its tapes.

   (Here we assume that the Turing machines have a separate input tape that we do not count in measuring space complexity.)

Deterministic Complexity Classes

Definition.

Let $T, S : \mathbb{N} \to \mathbb{N}$ be monotone increasing functions. Define

1. DTIME(T) as the class of languages $\mathcal{L}$ for which there is a T-time bounded k-tape Turing acceptor deciding $\mathcal{L}$, for some $k \geq 1$.
2. DSPACE(S) as the class of languages $\mathcal{L}$ for which there is a S-space bounded k-tape Turing acceptor deciding $\mathcal{L}$, $k \geq 1$.

Deterministic Complexity Classes

Definition.

Let $T, S : \mathbb{N} \to \mathbb{N}$ be monotone increasing functions. Define

1. DTIME(T) as the class of languages $\mathcal{L}$ for which there is a T-time bounded k-tape Turing acceptor deciding $\mathcal{L}$, for some $k \geq 1$.
2. DSPACE(S) as the class of languages $\mathcal{L}$ for which there is a S-space bounded k-tape Turing acceptor deciding $\mathcal{L}$, $k \geq 1$.

Important Complexity Classes:

- Time classes:
  - **P** (or PTIME) $:= \bigcup_{d\in\mathbb{N}} \text{DTIME}(n^d)$ polynomial time
  - EXP $:= \bigcup_{d\in\mathbb{N}} \text{DTIME}(2^{n^d})$ exponential time
  - 2-EXP $:= \bigcup_{d\in\mathbb{N}} \text{DTIME}(2^{2^{n^d}})$ double exp time
- Space classes:
  - LOGSPACE $:= \bigcup_{d\in\mathbb{N}} \text{DSPACE}(d \log n)$
  - PSPACE $:= \bigcup_{d\in\mathbb{N}} \text{DSPACE}(n^d)$
  - EXPSPACE $:= \bigcup_{d\in\mathbb{N}} \text{DSPACE}(2^{n^d})$

But wait...

Do these classes depend on exact def of "Turing machine"?

But wait...

Do these classes depend on exact def of “Turing machine”?

Yes, for DTIME(T), DSPACE(S);
No for the others

Indeed, usually don’t need to refer explicitly to “Turing machine”.
But watch out for nondeterminism (details later)

Important Time Complexity Classes:

- $\mathbf{P} := \bigcup_{d \in \mathbb{N}} \mathrm{DTIME}(n^d)$ — polynomial time
- $\mathrm{EXP} := \bigcup_{d \in \mathbb{N}} \mathrm{DTIME}(2^{n^d})$ — exponential time

Not quite so important:

- $2\text{-EXP} := \bigcup_{d \in \mathbb{N}} \mathrm{DTIME}(2^{2^{n^d}})$ — double exp time

Note: these are all classes of decision problems, i.e. languages.

Observation:

$$\mathbf{P} \subseteq \mathrm{EXP} \subseteq 2\text{-EXP} \subseteq \cdots \subseteq i\text{-EXP} \subseteq \ldots$$

Time Complexity Classes

Important Time Complexity Classes:

- $\mathbf{P} := \bigcup_{d\in\mathbb{N}} \mathsf{DTIME}(n^d)$ polynomial time
- $\mathsf{EXP} := \bigcup_{d\in\mathbb{N}} \mathsf{DTIME}(2^{n^d})$ exponential time

Not quite so important:

- $\text{2-EXP} := \bigcup_{d\in\mathbb{N}} \mathsf{DTIME}(2^{2^{n^d}})$ double exp time

Note: these are all classes of decision problems, i.e. languages.

Observation:

$$\mathbf{P} \subseteq \mathsf{EXP} \subseteq \text{2-EXP} \subseteq \cdots \subseteq i\text{-EXP} \subseteq \dots$$

Alternative definition/notation:

$$\mathbf{P} := \mathsf{DTIME}(n^{O(1)})$$

Linear Speed-Up

Theorem. (Linear Speed-Up Theorem)

Let $k > 1$ and $c > 0$ $\quad$ $T : \mathbb{N} \rightarrow \mathbb{N}$ $\quad$ $\mathcal{L} \subseteq \Sigma^*$ be a language.

If $\mathcal{L}$ can be decided by a $T(n)$ time-bounded k-tape TM

$$M := (Q, \Sigma, \Gamma, q_0, \delta, F)$$

then $\mathcal{L}$ can be decided by a $(\frac{1}{c} \cdot T(n) + n + 2)$ time-bounded k-tape TM

$$M^* := (Q', \Sigma, \Gamma', q_0', \delta', F').$$

Linear Speed-Up

Proof idea. Let $\Gamma' := \Sigma \cup \Gamma^s$ where $s := 6c$. To construct M^*:

Step 1: Compress M's input.

Copy (in $n+2$ steps) the input onto tape 2, compressing s symbols into one (i.e., each symbol corresponds to an s-tuple from Γ^s)

Step 2: Simulate M's computation, s steps at once.

1. Read (in 4 steps) symbols to the left, right and the current position
   and "store" (using $|Q \times \{1, \ldots, s\}^k \times \Gamma^{3sk}|$ extra states).
2. Simulate (in 2 steps) the next s steps of M (as M can only modify the current position and one of its neighbours)
3. M^* accepts (rejects) if M accepts (rejects)

(see Papadimitriou Thm 2.2, page 32)

A Hierarchy of Complexity Classes?

Questions we will study:

- Can we always solve more problems if we have more resources?
- If not, how much more resources do we need to be able to solve strictly more problems?
- How do the complexity classes relate to each other?
- How do we show that some problem is in one of these classes but not in another?
- Are there any other interesting models of computation?
  - Non-deterministic computation
  - Randomised algorithms

Next: robustness of **P**

Robustness of the definition of **P**

If **P** is to be the mathematical model of efficient computation, it should not depend on

- the exact computation-model we are using,
- or how we encode the input (within reason).

Robustness of the definition of **P**

If **P** is to be the mathematical model of efficient computation, it should not depend on

- the exact computation-model we are using,
- or how we encode the input (within reason).

Different Models of Computation:

1. We can simulate t steps of a k-tape Turing machine with an equivalent 1-tape TM in t^2 steps.
2. We can simulate t steps of a two-way infinite k-tape Turing machine with an equivalent standard k-tape TM in $O(t)$ steps.
3. We can simulate t steps of a RAM-machine with a 3-tape TM in $O(t^3)$ steps. Vice-versa in $O(t)$ steps.

Robustness of the definition of **P**

If **P** is to be the mathematical model of efficient computation, it should not depend on

- the exact computation-model we are using,
- or how we encode the input (within reason).

Different Models of Computation:

1. We can simulate t steps of a k-tape Turing machine with an equivalent 1-tape TM in t^2 steps.
2. We can simulate t steps of a two-way infinite k-tape Turing machine with an equivalent standard k-tape TM in $O(t)$ steps.
3. We can simulate t steps of a RAM-machine with a 3-tape TM in $O(t^3)$ steps. Vice-versa in $O(t)$ steps.

Consequence: **P** is the same for all these models (unlike linear time)

Different Encodings

Observation.

1. For any $n \in \mathbb{N}$, the length of the encoding of n in base b_1 and base b_2 are related by a constant factor, for all $b_1, b_2 \geq 2$.
2. For any graph G, the length of its encoding as an
   - adjacency matrix
   - list of edges
   - adjacency list
   - ...

   are all related by a polynomial factor.

Different Encodings

Observation.

1. For any $n \in \mathbb{N}$, the length of the encoding of n in base b_1 and base b_2 are related by a constant factor, for all $b_1, b_2 \geq 2$.
2. For any graph G, the length of its encoding as an
   - adjacency matrix
   - list of edges
   - adjacency list
   - ...

   are all related by a polynomial factor.

Consequence: (for problems on numbers, graphs) **P** is the same for all these encoding (unlike linear time)

Strong Church-Turing Hypothesis

Any function which can be computed by any well-defined procedure can be computed by a Turing machine with only polynomial overhead.

(but doesn't apply to quantum or randomised algorithms)

I also pointed out that "in **P**" corresponds well to existence of a practical algorithm; problem is "tractable"

Growth Rate of Functions (Garey/Johnson '79)

Time complexity function	Size n					
	10	20	30	40	50	60
n	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second	.0036 second
n^3	.001 second	.008 second	.027 second	.064 second	.125 second	.216 second
n^5	.1 second	3.2 seconds	24.3 seconds	1.7 minutes	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	2×10^8 centuries	1.3×10^{13} centuries

Figure 1.2 Comparison of several polynomial and exponential time complexity functions

Proving a problem is in **P**

Good news: proofs of "in **P**" are often cleaner than detailed runtime analysis;
"in **P**" less specific than, e.g. "in DTIME(n^2)"; some technical details are avoided

- The most direct way to show that a problem is in **P** is to exhibit a polynomial time algorithm that solves it.
- Even a naive polynomial-time algorithm often provides a good insight into how the problem can be solved efficiently.
- Because of robustness, we do not generally need to specify all the details of the machine model or the encoding.

  $\rightsquigarrow$ pseudo-code is sufficient.

Example: Satisfiability

Some of the most important problems concern logical formulae

Recall propositional logic

Formulae of propositional logic are built up inductively

- Variables: $X_i \qquad i \in \mathbb{N}$
- Boolean connectives:
  If φ, ψ are propositional formulae then so are
  - $(\psi \vee \varphi)$
  - $(\psi \wedge \varphi)$
  - $\neg\varphi$

Example:

$(X_1 \vee X_2 \vee \neg X_5) \ \wedge \ (\neg X_2 \vee \neg X_4 \vee \neg X_5) \ \wedge \ (X_2 \vee X_3 \vee X_4)$

Conjunctive Normal Form

Formula φ is in conjunctive normal form (CNF) if

$$\varphi := C_1 \wedge \cdots \wedge C_m$$

where each C_i is a clause, that is, a disjunction of literals

$$C_i := (L_{i1} \vee \cdots \vee L_{ik})$$

A literal is a variable X_i or a negated variable $\neg X_i$

k-CNF: CNF φ with at most k literals per clause.

3-CNF example:

$(X_1 \vee X_2 \vee \neg X_5) \ \wedge \ (\neg X_2 \vee \neg X_4) \ \wedge \ (X_2 \vee X_3 \vee X_4) \ \wedge X_6$

Conjunctive Normal Form

Formula φ is in conjunctive normal form (CNF) if

$$\varphi := C_1 \wedge \cdots \wedge C_m$$

where each C_i is a clause, that is, a disjunction of literals

$$C_i := (L_{i1} \vee \cdots \vee L_{ik})$$

A literal is a variable X_i or a negated variable $\neg X_i$

k-CNF: CNF φ with at most k literals per clause.

3-CNF example:

$(X_1 \vee X_2 \vee \neg X_5) \ \wedge \ (\neg X_2 \vee \neg X_4) \ \wedge \ (X_2 \vee X_3 \vee X_4) \ \wedge X_6$

common CNF notation:

$\varphi := \{ \ \{X_1, X_2, \neg X_5\}, \quad \{\neg X_2, \neg X_4\}, \quad \{X_2, X_3, X_4\}, \quad \{X_6\} \ \}$

Satisfiability

Definition. A formula φ is satisfiable if there is a satisfying assignment (a.k.a. model) for φ.

In the case of formulae in CNF:
An assignment β assigning values 0 or 1 to the variables of φ so that every clause contains at least

- one variable to which β assigns 1 or
- one negated variable to which β assigns 0.

Example:

$$(X_1 \vee X_2 \vee \neg X_5) \wedge (\neg X_2 \vee \neg X_4 \vee \neg X_5) \wedge (X_2 \vee X_3 \vee X_4)$$

Satisfying assignment:

$$X_1 \mapsto 1 \qquad X_2 \mapsto 0 \qquad X_3 \mapsto 1 \qquad X_4 \mapsto 0 \qquad X_5 \mapsto 1$$

The Satisfiability Problem

In association with propositional formulae, the following two problems are the most important:

SAT

Input:	Propositional formula φ in CNF
Problem:	Is φ satisfiable?

***k*-SAT**

Input:	Propositional formula φ in k-CNF
Problem:	Is φ satisfiable?

(Let us also note CIRCUIT SAT: given a circuit with n inputs, one output, can we set input values to get output=TRUE?)

2-SAT is in **P**

Proof. The following algorithm solves the problem in poly time.

Let φ be the input formula
Repeat
  If φ contains clauses $\{X\}$ and $\{\neg X\}$, halt and output "no";
  If φ contains clauses $\{X\}$ and $\{\neg X, Y\}$, add clause $\{Y\}$;
  If φ contains clauses $\{X, Y\}$ $\{\neg X, Z\}$, add clause $\{Y, Z\}$;
  Any clause $\{X, X\}$ simplifies to $\{X\}$
Output "yes".

2-SAT is in **P**

Proof. The following algorithm solves the problem in poly time.

Let φ be the input formula
Repeat
  If φ contains clauses $\{X\}$ and $\{\neg X\}$, halt and output "no";
  If φ contains clauses $\{X\}$ and $\{\neg X, Y\}$, add clause $\{Y\}$;
  If φ contains clauses $\{X, Y\}$ $\{\neg X, Z\}$, add clause $\{Y, Z\}$;
  Any clause $\{X, X\}$ simplifies to $\{X\}$
Output "yes".

Poly-time:

- there are $O(n^2)$ iterations.
- Each "if" test searches for $O(n^2)$ items in φ
- Each search is linear in length of φ

above analysis is crude but does the job.

Polynomial-Time Reductions

As for decidability we can use many-one reductions to show membership in **P**.

Definition. A language $\mathcal{L}_1 \subseteq \Sigma^*$ is polynomially reducible to $\mathcal{L}_2 \subseteq \Sigma^*$, denoted $\mathcal{L}_1 \leq_p \mathcal{L}_2$, if there is a polynomial-time computable function f such that for all $w \in \Sigma^*$

$$w \in \mathcal{L}_1 \iff f(w) \in \mathcal{L}_2.$$

Polynomial-Time Reductions

As for decidability we can use many-one reductions to show membership in **P**.

Definition. A language $\mathcal{L}_1 \subseteq \Sigma^*$ is polynomially reducible to $\mathcal{L}_2 \subseteq \Sigma^*$, denoted $\mathcal{L}_1 \leq_p \mathcal{L}_2$, if there is a polynomial-time computable function f such that for all $w \in \Sigma^*$

$$w \in \mathcal{L}_1 \qquad \Longleftrightarrow \qquad f(w) \in \mathcal{L}_2.$$

Lemma. If $\mathcal{L}_1 \leq_p \mathcal{L}_2$ and $\mathcal{L}_2 \in \mathbf{P}$ then $\mathcal{L}_1 \in \mathbf{P}$.

Proof idea. The sum and composition of polynomials is a polynomial.

Generally, members of **P** can be poly-time reduced to each other.

Vertex Colouring:

A vertex colouring of G with k colours is a function

$$c : V(G) \longrightarrow \{1, \ldots, k\}$$

such that adjacent nodes have different colours

i.e. $\{u, v\} \in E(G)$ implies $c(u) \neq c(v)$

***k*-COLOURABILITY**

Input:	Graph G, $k \in \mathbb{N}$
Problem:	Does G have a vertex colouring with k colours?

For $k = 2$ this is the same as BIPARTITE.

A reduction to 3-Sat

Lemma. k-Colourability $\leq_p$ 3-Sat

Proof.

Introduce $X_{v,c}$ to represent "in a solution, v gets colour c".

clauses impose constraints, e.g. $X_{vc} \Rightarrow \neg X_{vc'}$ (or rather, $\neg X_{vc} \vee \neg X_{vc'}$)

$X_{vc} \Rightarrow \neg X_{v'c}$ for (v, v') any edge

$X_{v1} \vee X_{v2} \vee \ldots \vee X_{vk}$ for each v

can replace e.g. $X_{v1} \vee X_{v2} \vee X_{v3} \vee X_{v4}$ with $X_{v1} \vee X_{v2} \vee X_{new}$ and $\neg X_{new} \vee X_{v3} \vee X_{v4}$

Reducible to 2-Sat ??