Complexity Theory

Time Complexity and Polynomial Time

Daniel Borchmann, Markus Krötzsch

Computational Logic

2015-11-10

Time Complexity

Measuring Complexity

Complexity Theory

Study the fine structure of decidable languages.

Goal

Classify languages by the amount of resources needed to solve them.

Resources

When dealing with Turing machines, 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)

Time and Space Bounded Turing Machines

Definition 6.1

Let $\mathcal{M}$ be a Turing machine and let $f : \mathbb{N} \to \mathbb{R}^+$ be a function.

- $\mathcal{M}$ is f-time bounded if it halts on every input $w \in \Sigma^*$ after $\leq f(|w|)$ steps.
- $\mathcal{M}$ is f-space bounded if it halts on every input $w \in \Sigma^*$ using $\leq f(|w|)$ cells on its tapes.

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

Notation

Sometimes notations like "$f(n)$-time bounded" are used, assuming inputs to be of length n
$\leadsto$ we can use this when convenient, e.g., to write "n^3-bounded"

Big-O and Small-o

Recall: Big-O notation

Classify functions by an asymptotic bound that hides linear factors:

$$f(n) = O(g(n)) \quad \text{iff} \quad \exists c > 0\ \exists n_0 \in \mathbb{N}\ \forall n > n_0 : f(n) \leq c \cdot g(n)$$

In words:

f is asymptotically bounded by g up to a constant factor

Small-o notation

Classify functions by a function that dominates them:

$$f(n) = o(g(n)) \quad \text{iff} \quad \forall c > 0\ \exists n_0 \in \mathbb{N}\ \forall n > n_0 : f(n) \leq c \cdot g(n)$$

In words:

f is asymptotically dominated by g

Relaxed Time and Space Bounds

We can use Big-O notation to generalise bounded TMs:

- $\mathcal{M}$ is $O(g(n))$-time bounded if it is f-time bounded for some f with $f(n) = O(g(n))$.
- $\mathcal{M}$ is $O(g(n))$-space bounded if it is f-space bounded for some f with $f(n) = O(g(n))$.

Notation

We generally allow the use of $O(g(n))$ in place of a function $f(n)$ with analogous meaning.

Deterministic Complexity Classes

Definition 6.2

Let $f : \mathbb{N} \to \mathbb{R}^+$ be a function.

- $\textsc{DTime}(f(n))$ is the class of all languages $\mathcal{L}$ for which there is an $O(f(n))$-time bounded Turing machine deciding $\mathcal{L}$.
- $\textsc{DSpace}(f(n))$ is the class of all languages $\mathcal{L}$ for which there is an $O(f(n))$-space bounded Turing machine deciding $\mathcal{L}$.

Notation

Sometimes $\textsc{Time}(f(n))$ is used instead of $\textsc{DTime}(f(n))$.

Some Important Complexity Classes

$$\mathrm{P} = \textsc{PTime} = \bigcup_{d \geq 1} \textsc{DTime}(n^d)$$ polynomial time

$$\textsc{Exp} = \textsc{ExpTime} = \bigcup_{d \geq 1} \textsc{DTime}(2^{n^d})$$ exponential time

$$2\textsc{Exp} = 2\textsc{ExpTime} = \bigcup_{d \geq 1} \textsc{DTime}(2^{2^{n^d}})$$ double-exponential time

$$\mathrm{E} = \textsc{ETime} = \bigcup_{d \geq 1} \textsc{DTime}(2^{dn})$$ exp. time with linear exponent

$$\mathrm{L} = \textsc{LogSpace} = \textsc{DSpace}(\log n)$$ logarithmic space

$$\textsc{PSpace} = \bigcup_{d \geq 1} \textsc{DSpace}(n^d)$$ polynomial space

$$\textsc{ExpSpace} = \bigcup_{d \geq 1} \textsc{DSpace}(2^{n^d})$$ exponential space

Time Complexity Classes

$$\text{P} = \text{PTime} = \bigcup_{d \geq 1} \text{DTime}(n^d)$$ polynomial time

$$\text{Exp} = \text{ExpTime} = \bigcup_{d \geq 1} \text{DTime}(2^{n^d})$$ exponential time

$$2\text{Exp} = 2\text{ExpTime} = \bigcup_{d \geq 1} \text{DTime}(2^{2^{n^d}})$$ double-exponential time

Note

Complexity classes are classes of languages.

Time Complexity

$$\text{P} \subseteq \text{ExpTime} \subseteq 2\text{ExpTime} \subseteq 3\text{ExpTime} \subseteq 4\text{ExpTime} \subseteq \ldots$$

A Hierarchy of Complexity Classes?

- 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?
- Are there any tools by which we can show that a problem is in any of these classes but not in another?

$\rightsquigarrow$ discussed in future lectures

- How do we classify “efficient” in terms of complexity classes?

$\rightsquigarrow$ coming up next

Different Definitions of Complexity Classes?

Other models of computation?

- Is $\mathrm{DT{\scriptsize IME}}(f)$ the same for multi-tape TMs?
- And how about non-deterministic TMs?
- Or TMs with a two-way infinite tape?
- Or random access machines?
- ...

Many complexity classes are robust against many such variations
$\leadsto$ coming up next

Polynomial Time

Polynomial Time

"Intuitive" definition of "efficient":

- Any linear time computation is "efficient".
- Any program that
  - performs "efficient" operations (e.g. linear number of iterations) and
  - only uses "efficient" subprograms

  is "efficient".

This turns out to be equivalent to PTIME.

$$\textsc{PTime} := \bigcup_{d \geq 1} \textsc{DTime}(n^d)$$

PTIME serves as a mathematical model of "efficient" computation.

Robustness of the Definition

If PTime 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).

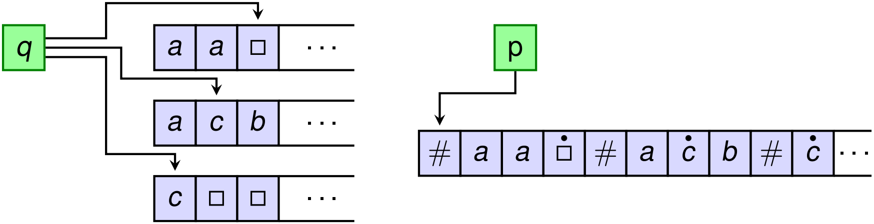
Multi-Tape Turing Machines

Theorem 6.3 (Sipser, Theorem 7.8)

Consider a function f with $f(n) \geq n$. Then, for every $f(n)$-time bounded k-tape Turing machine ($k > 1$), there is an equivalent $O(f^2(n))$-time bounded single-tape Turing machine.

Proof.

Simulate a multi-tape TM with a single-tape TM as shown in Lecture 2:



Multi-Tape Turing Machines

Then analyse how long this simulation really takes:

- Observation: the tapes can never have more than $f(n)$ symbols on them
- The simulation scans the whole tape once to find out what to do: $O(f(n))$ steps
- Then it updates the tapes whole tape in one pass: $O(f(n))$ steps
- Sometimes the whole tape is shifted to make space: at most k times $O(f(n))$ steps
- Overall: one step is simulated in $O(f(n))$ steps
- Simulating $f(n)$ such steps takes $f(n) \cdot O(f(n)) = O(f^2(n))$ steps
- Tape initialisation takes another $O(f(n))$ (irrelevant)

Total simulation possible in $O(f^2(n))$. □

P is Robust for Multi-Tape TMs

Let $\mathrm{DTIME}_k(f(n))$ denote "$\mathrm{DTIME}(f(n))$ for a k-tape TM".

Theorem 6.4

$$\bigcup_{d\in\mathbb{N}} \mathrm{DTIME}(n^d) = \bigcup_{d\in\mathbb{N}} \mathrm{DTIME}_k(n^d) \text{ for every } k \geq 1$$

Proof.

The inclusion $\subseteq$ is clear. The inclusion $\supseteq$ is immediate from the previous theorem. □

Robustness Against Other Models of Computation

P is robust against further models of computation:

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

Consequences:

- $\mathrm{PT{\small IME}}$ is the same for all these models (unlike linear time)
- The exponential time complexity classes are as robust as P

How about non-deterministic TMs?

It is unknown if $\mathrm{PT{\small IME}}$ is robust against this, but most think it is not $\rightsquigarrow$ see next lectures

Linear Speed-Up

The Big-O notation in DTIME hides arbitrary linear factors.
Is it justified to rely on this for defining P?

Yes, it turns out that we can make multi-tape TMs "arbitrarily fast":

Theorem 6.5 (Linear Speed-Up Theorem)

Consider an $f(n)$-time bounded k-tape Turing machine $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ with $k > 1$.

Then, for every constant $c > 0$, there is a $(\frac{1}{c} \cdot f(n) + n + 2)$-time bounded k-tape TM $\mathcal{M}' = (Q', \Sigma, \Gamma', \delta', q'_0, q'_{accept}, q'_{reject})$ that accepts the same language.

Linear Speed-Up

Proof sketch.

Let $\Gamma' := \Sigma \cup \Gamma^m$ where $m := \lceil 6c \rceil$. We construct $\mathcal{M}'$ as follows:

Step 1: Compress $\mathcal{M}$'s input.

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

Step 2: Simulate $\mathcal{M}$'s computation, m steps at once.

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

For details see Papadimitriou, Theorem 2.2. □

Different Encodings

Some simple observations:

- 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 $b_1, b_2 \geq 2$.
- For any graph G, the length of its encoding as an
  - adjacency matrix
  - list of nodes + list of edges
  - adjacency list
  - ...

  are all polynomially related.

Consequence:

PTime is the same for all these encodings (unlike linear time).

PTime = tractable?

The class PTime is a reasonable mathematical model of the class of problems which are tractable or solvable in practice.

However:

This correspondence is not exact.

- When the degree of polynomials is very high, the time grows so quickly that in practice the problem is not solvable.
- The constants may also be very large

And yet:

For many concrete PTime-problems arising in practice, algorithms with moderate exponents and constants have been found.

Growth Rate of Functions

Time Complexity	Size n					
	10	20	30	40	50	60
n	.00001 seconds	.00002 seconds	.00003 seconds	.00004 seconds	.00005 seconds	.00006 seconds
n^2	.0001 seconds	.0004 seconds	.0009 seconds	.0016 seconds	.0025 seconds	.0036 seconds
n^3	.001 seconds	.008 seconds	.027 seconds	.064 seconds	.125 seconds	.216 seconds
n^{10}	166 minutes	119 days	18.7 years	3.3 centuries	31 centuries	192 centuries
2^n	.001 seconds	1.0 seconds	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	.059 seconds	58 minutes	6.5 years	3855 centuries	$2 \cdot 10^8$ centuries	$1.3 \cdot 10^{13}$ centuries

Polynomial Time: Examples

Proving a problem is in PTIME

- The most direct way to show that a problem is in PTIME 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

Definition 6.6 (Propositional Logic Syntax)

Formulae of propositional logic are built up inductively

- (Propositional) 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 6.7

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

Conjunctive Normal Form

Definition 6.8 (Conjunctive Normal Form)

A propositional logic 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

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

and a literal is a variable X_i or a negation $\neg X_i$ thereof.
A CNF φ is in k-CNF is it has at most k literals per clause.

Example 6.9

$(X_1 \vee X_2 \vee \neg X_5) \; \wedge \; (\neg X_2 \vee \neg X_4 \vee \neg X_5) \; \wedge \; (X_2 \vee X_3 \vee X_4)$ is in 3-CNF

Propositional Logic Semantics

Definition 6.10

A formula φ is satisfiable if there is a satisfying assignment for φ.

Specifically for 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 6.11

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

is satisfied by $\{X_1 \mapsto 1,\ X_2 \mapsto 0,\ X_3 \mapsto 1,\ X_4 \mapsto 0,\ X_5 \mapsto 1\}$

The Satisfiability Problem

Related to 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?

2-SAT is in PTIME

Proof.

The following algorithm solves the problem in polynomial time.

Input Γ in CNF

```
bcp(Γ)
if conflict return UNSAT

while Γ ≠ ∅ do
    choose var. X from Γ
    set Γ' := Γ
    assign(Γ, X, 1)
    bcp(Γ)
    if conflict
        Γ := Γ'
        assign(Γ, X, 0)
        bcp(Γ)
        if conflict
            return UNSAT
```

bcp(Γ) (boolean constraint propagation)

```
while Γ contains unit-clause C do
    if C = {X}      assign(Γ, X, 1)
    if C = {¬X}     assign(Γ, X, 0)
if Γ contains empty clause return conflict
```

assign(Γ, X, c)

```
if c = 1
    remove from Γ all clauses C with X ∈ C
    remove ¬X from all remaining clauses
if c = 0
    remove from Γ all clauses C with ¬X ∈ C
    remove X from all remaining clauses
```

□

Polynomial-Time Reductions

As for decidability we can use reductions to show membership in PTime.

Definition 6.12

A language $\mathcal{L}_1 \subseteq \Sigma^*$ is polynomially many-one 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 $w \in \Sigma^*$

$$w \in \mathcal{L}_1 \qquad \text{if and only if} \qquad f(w) \in \mathcal{L}_2.$$

Theorem 6.13

If $\mathcal{L}_1 \leq_p \mathcal{L}_2$ and $\mathcal{L}_2 \in \mathrm{PTime}$ then $\mathcal{L}_1 \in \mathrm{PTime}$.

Proof.

The sum and composition of polynomials is a polynomial. □

Reductions in PTIME

All non-trivial members of PTIME can be reduced to each other:

Theorem 6.14

If $\mathcal{B}$ is any language in P, $\mathcal{B} \neq \emptyset$, $\mathcal{B} \neq \Sigma^*$, then $\mathcal{A} \leq_p \mathcal{B}$ for any $\mathcal{A} \in \mathrm{P}$.

Proof.

Choose $w \in \mathcal{B}$ and $w' \notin \mathcal{B}$

Define the function f by setting

$$\begin{aligned} f(x) &:= w & x \in \mathcal{A} \\ f(x) &:= w' & x \notin \mathcal{A} \end{aligned}$$

Since $\mathcal{A} \in \mathrm{P}$, f is computable in polynomial time, and is a reduction from $\mathcal{A}$ to $\mathcal{B}$. □

Example: Colourability

Definition 6.15 (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, that is:

$$\{u, v\} \in E(G) \text{ implies } c(u) \neq c(v)$$

k-COLOURING

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.

Reducing 2-COLOURABILITY to 2-SAT

Theorem 6.16

2-COLOURABILITY $\leq_p$ 2-SAT, *and therefore* 2-COLOURABILITY $\in \mathrm{P}$.

Proof.

We define a reduction as follows: Given graph G

- For each vertex $v \in V(G)$ of the graph introduce new variable X_v
- For each $\{u, v\} \in E(G)$ add clauses $(X_u \vee X_v)$ and $(\neg X_u \vee \neg X_v)$

This is obviously computable in polynomial time.

We check that it is a reduction:

- If G is 2-colourable, use colouring to assign truth values. (One colour is *true*, the other *false*)
- If the formula is satisfiable, the truth assignment defines valid 2-colouring.

Trivially Tractable Problems

A large class of languages is generally tractable:

Theorem 6.17

If $\mathcal{L}$ is a finite language, then it is decided by an $O(1)$-time bounded TM. In other words, all finite languages are decidable in constant time (and hence also in polynomial time).

Proof.

- As $\mathcal{L}$ is finite, there is a maximum length m of words in $\mathcal{L}$.
- Read the input up to the first m letters.
- The state space contains a table containing the correct result for all such inputs.
- All other inputs are rejected.

□

A Note on Constructiveness

The previous result is an example of a theorem that proves the existence of a P algorithm in cases where we do not know what this algorithm is.

Example 6.18

Let $\mathcal{L}$ be the language that contains all correct sentences from the following set:

$$\{\text{“}\mathrm{P}\text{ is the same as }\mathrm{NP}\text{”}, \text{“}\mathrm{P}\text{ is not the same as }\mathrm{NP}\text{”}\}$$

Then $\mathcal{L}$ is decidable in constant time. However, we don't which constant-time algorithm decides this.

Non-constructiveness:

- We can prove that there is a correct polynomial time algorithm.
- We cannot construct such an algorithm.

Such solutions are called non-constructive.

An Interesting Problem in P

Theorem 6.19

It is decidable in polynomial-time ($O(n^3)$) if a graph can knotlessly be embedded into 3-dimensional space.

Proof sketch.

- Robertson & Seymour proved a general result that implies the existence of a finite set of forbidden structures in knotlessly embeddable graphs.
- For each of these forbidden structures we can test whether a graph contains one of them in time $O(n^3)$.
- Hence, to decide if a graph is knotlessly embeddable, we only need to test for each of the finitely many forbidden structures, whether they occur in the graph.

This yields a cubic time decision procedure. □

However: We do not currently know what these structures are.