

Theory of Computer Science

E1. Complexity Theory: Motivation and Introduction

Gabriele Röger

University of Basel

April 24, 2019

Theory of Computer Science

April 24, 2019 — E1. Complexity Theory: Motivation and Introduction

E1.1 Motivation

E1.2 How to Measure Runtime?

E1.3 Decision Problems

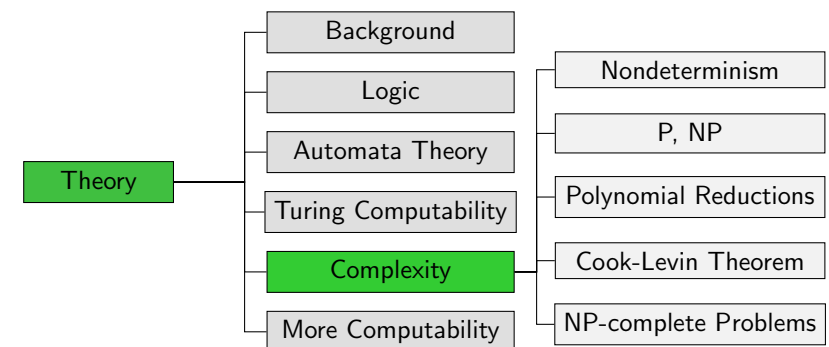
E1.4 Nondeterminism

Overview: Course

contents of this course:

- A. background ✓
 - ▷ mathematical foundations and proof techniques
- B. logic ✓
 - ▷ How can knowledge be represented?
 - How can reasoning be automated?
- C. automata theory and formal languages ✓
 - ▷ What is a computation?
- D. Turing computability ✓
 - ▷ What can be computed at all?
- E. complexity theory
 - ▷ What can be computed efficiently?
- F. more computability theory
 - ▷ Other models of computability

Course Overview



E1.1 Motivation

A Scenario (1)

Example Scenario

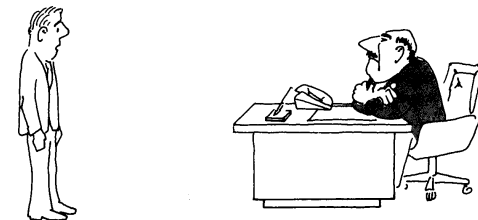
- ▶ You are a programmer at a logistics company.
- ▶ Your boss gives you the task of developing a program to optimize the route of a delivery truck:
 - ▶ The truck begins its route at the company depot.
 - ▶ It has to visit 50 stops.
 - ▶ You know the distances between all relevant locations (stops and depot).
 - ▶ Your program should compute a tour visiting all stops and returning to the depot on a **shortest route**.

A Scenario (2)

Example Scenario (ctd.)

- ▶ You work on the problem for weeks, but you do not manage to complete the task.
- ▶ All of your attempted programs
 - ▶ **compute routes that are possibly suboptimal**, or
 - ▶ **do not terminate in reasonable time** (say: within a month).
- ▶ **What do you say to your boss?**

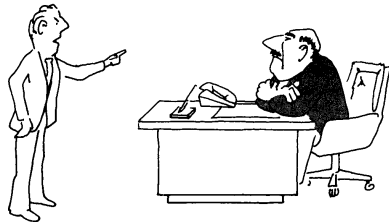
What You Don't Want to Say



**"I can't find an efficient algorithm,
I guess I'm just too dumb."**

Source: M. Garey & D. Johnson, Computers and Intractability, Freeman 1979, p. 2

What You Would Like to Say



"I can't find an efficient algorithm,
because no such algorithm is possible!"

Source: M. Garey & D. Johnson, Computers and Intractability, Freeman 1979, p. 2

What Complexity Theory Allows You to Say



"I can't find an efficient algorithm,
but neither can all these famous people."

Source: M. Garey & D. Johnson, Computers and Intractability, Freeman 1979, p. 3

Why Complexity Theory?

Complexity Theory

Complexity theory tells us which problems can be solved **quickly** ("simple problems") and which ones **cannot** ("hard problems").

German: Komplexitätstheorie

- ▶ This is useful in practice because simple and hard problems require **different techniques** to solve.
- ▶ If we can show that a problem is hard we do not need to waste our time with the (futile) search for a "simple" algorithm.

Why Reductions?

Reductions

An important part of complexity theory are (polynomial) **reductions** that show how a given problem P can be reduced to another problem Q .

German: Reduktionen

- ▶ useful for **theoretical analysis** of P and Q because it allows us to transfer our knowledge between them
- ▶ often also useful for **practical algorithms for P** : reduce P to Q and then use the best known algorithm for Q

Test Your Intuition! (1)

- ▶ The following slide lists some **graph problems**.
- ▶ The input is always a **directed graph** $G = \langle V, E \rangle$.
- ▶ **How difficult** are the problems in your opinion?
- ▶ Sort the problems from **easiest** (= requires least amount of time to solve) to **hardest** (= requires most time to solve)
- ▶ **no justification necessary**, just follow your intuition!
- ▶ **anonymous** and **not graded**

Test Your Intuition! (2)

- 1 Find a **simple path** (= without cycle) from $u \in V$ to $v \in V$ with **minimal length**.
- 2 Find a **simple path** (= without cycle) from $u \in V$ to $v \in V$ with **maximal length**.
- 3 Determine whether G is **strongly connected** (every node is reachable from every other node).
- 4 Find a **cycle** (non-empty path from u to u for any $u \in V$; multiple visits of nodes are allowed).
- 5 Find a **cycle** that visits **all** nodes.
- 6 Find a **cycle** that visits a **given node** u .
- 7 Find a path that **visits all nodes** without repeating a node.
- 8 Find a path that **uses all edges** without repeating an edge.

E1.2 How to Measure Runtime?

- ▶ **Time complexity** is a way to measure **how much time** it takes to solve a problem.
- ▶ How can we **define** such a measure appropriately?

German: Zeitkomplexität/Zeitaufwand

How to Measure Runtime?

- ▶ **Time complexity** is a way to measure **how much time** it takes to solve a problem.
- ▶ How can we **define** such a measure appropriately?

German: Zeitkomplexität/Zeitaufwand

Example Statements about Runtime

Example statements about runtime:

- ▶ “Running `sort /usr/share/dict/words` on the computer `dakar` takes 0.035 seconds.”
- ▶ “With a 1 MiB input file, `sort` takes at most 1 second on a modern computer.”
- ▶ “Quicksort is faster than sorting by insertion.”
- ▶ “Sorting by insertion is slow.”

↔ Very different statements with different **pros and cons**.

Precise Statements vs. General Statements

Example Statement about Runtime

“Running `sort /usr/share/dict/words` on the computer `dakar` takes 0.035 seconds.”

advantage: very **precise**

disadvantage: not **general**

- ▶ **input-specific**:
What if we want to sort other files?
- ▶ **machine-specific**:
What happens on a different computer?
- ▶ even **situation-specific**:
Will we get the same result tomorrow that we got today?

General Statements about Runtime

In this course we want to make **general** statements about runtime. We accomplish this in three ways:

1. General Inputs

Instead of **concrete** inputs, we talk about **general types** of input:

- ▶ **Example**: runtime to sort an input of size n in the **worst case**
- ▶ **Example**: runtime to sort an input of size n in the **average case**

here: runtime for input size n in the **worst case**

General Statements about Runtime

In this course we want to make **general** statements about runtime. We accomplish this in three ways:

2. Ignoring Details

Instead of **exact formulas** for the runtime we specify the **order of magnitude**:

- ▶ **Example**: instead of saying that we need time $[1.2n \log n] - 4n + 100$, we say that we need time $O(n \log n)$.
- ▶ **Example**: instead of saying that we need time $O(n \log n)$, $O(n^2)$ or $O(n^4)$, we say that we need **polynomial** time.

here: What can be computed in **polynomial time**?

General Statements about Runtime

In this course we want to make **general** statements about runtime. We accomplish this in three ways:

3. Abstract Cost Measures

Instead of the **runtime on a concrete computer** we consider a **more abstract** cost measure:

- ▶ **Example:** count the number of executed **machine code statements**
- ▶ **Example:** count the number of executed **Java byte code statements**
- ▶ **Example:** count the number of **element comparisons** of a sorting algorithms

here: count the computation steps of a **Turing machine** (**polynomially equivalent** to other measures)

E1.3 Decision Problems

Decision Problems

- ▶ As before, we simplify our investigation by restricting our attention to **decision problems**.
- ▶ More complex computational problems can be solved with multiple queries for an appropriately defined decision problem (“playing 20 questions”).
- ▶ Formally, decision problems are **languages** (as before), but we use an informal **“given”/“question”** notation where possible.

Example: Decision vs. General Problem (1)

Definition (Hamilton Cycle)

Let $G = \langle V, E \rangle$ be a (directed or undirected) graph.

A **Hamilton cycle** of G is a sequence of vertices in V , $\pi = \langle v_0, \dots, v_n \rangle$, with the following properties:

- ▶ π is a **path**: there is an edge from v_i to v_{i+1} for all $0 \leq i < n$
- ▶ π is a **cycle**: $v_0 = v_n$
- ▶ π is **simple**: $v_i \neq v_j$ for all $i \neq j$ with $i, j < n$
- ▶ π is **Hamiltonian**: all nodes of V are included in π

German: Hamiltonkreis/Hamiltonzyklus

Example: Decision vs. General Problem (2)

Example (Hamilton Cycles in Directed Graphs)

\mathcal{P} : general problem DIRHAMILTONCYCLEGEN

- ▶ **Input:** directed graph $G = \langle V, E \rangle$
- ▶ **Output:** a Hamilton cycle of G or a message that none exists

\mathcal{D} : decision problem DIRHAMILTONCYCLE

- ▶ **Given:** directed graph $G = \langle V, E \rangle$
- ▶ **Question:** Does G contain a Hamilton cycle?

These problems are **polynomially equivalent**:
from a polynomial algorithm for one of the problems
one can construct a polynomial algorithm for the other problem.
(Without proof.)

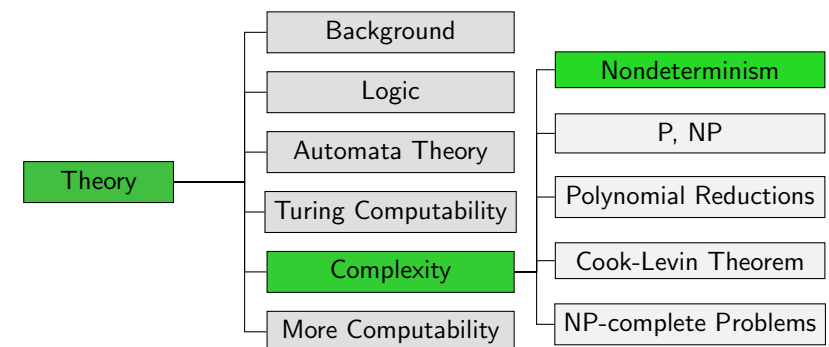
Algorithms for Decision Problems

Algorithms for decision problems:

- ▶ Where possible, we specify algorithms for decision problems in **pseudo-code**.
- ▶ Since they are only yes/no questions, we do not have to return a general result.
- ▶ Instead we use the statements
 - ▶ **ACCEPT** to **accept** the given input (“yes” answer) and
 - ▶ **REJECT** to **reject** it (“no” answer).
- ▶ Where we must be more formal, we use **Turing machines** and the notion of accepting from chapter C7.

E1.4 Nondeterminism

Course Overview



Nondeterminism

- ▶ To develop complexity theory, we need the algorithmic concept of **nondeterminism**.
- ▶ already known for **Turing machines** (\rightsquigarrow chapter C7):
 - ▶ An NTM can have **more than one possible successor configuration** for a given configuration.
 - ▶ Input x is accepted if there is **at least one possible computation** (configuration sequence) that leads to an end state.
- ▶ Here we analogously introduce nondeterminism for pseudo-code.

German: Nichtdeterminismus

Nondeterministic Algorithms

nondeterministic algorithms:

- ▶ All constructs of deterministic algorithms are also allowed in nondeterministic algorithms: **IF**, **WHILE**, etc.
- ▶ Additionally, there is a **nondeterministic assignment**:

GUESS $x_i \in \{0, 1\}$

where x_i is a program variable.

German: nichtdeterministische Zuweisung

Nondeterministic Algorithms: Acceptance

- ▶ Meaning of **GUESS** $x_i \in \{0, 1\}$:
 x_i is assigned **either** the value **0** or the value **1**.
- ▶ This implies that the behavior of the program on a given input is no longer uniquely defined: there are multiple possible execution paths.
- ▶ The program accepts a given input if **at least one execution path** leads to an **ACCEPT** statement.
- ▶ Otherwise, the input is rejected.

Note: **asymmetry** between accepting and rejecting!
(cf. semi-decidability)

More Complex GUESS Statements

- ▶ We will also guess more than one bit at a time:

GUESS $x \in \{1, 2, \dots, n\}$

or more generally

GUESS $x \in S$

for a set S .

- ▶ These are abbreviations and can be split into $\lceil \log_2 n \rceil$ (or $\lceil \log_2 |S| \rceil$) “atomic” **GUESS** statements.

Example: Nondeterministic Algorithms (1)

Example (DIRHAMILTONCYCLE)

input: directed graph $G = \langle V, E \rangle$

$start :=$ an arbitrary node from V

$current := start$

$remaining := V \setminus \{start\}$

WHILE $remaining \neq \emptyset$:

GUESS $next \in remaining$

IF $\langle current, next \rangle \notin E$:

REJECT

$remaining := remaining \setminus \{next\}$

$current := next$

IF $\langle current, start \rangle \in E$:

ACCEPT

ELSE:

REJECT

Example: Nondeterministic Algorithms (2)

- ▶ With appropriate data structures, this algorithm solves the problem in $O(n \log n)$ program steps, where $n = |V| + |E|$ is the size of the input.
- ▶ How many steps would a **deterministic** algorithm need?

Guess and Check

- ▶ The DIRHAMILTONCYCLE example illustrates a general design principle for nondeterministic algorithms:
guess and check
- ▶ In general, nondeterministic algorithms can solve a problem by first guessing a “solution” and then verifying that it is indeed a solution. (In the example, these two steps are interleaved.)
- ▶ If solutions to a problem can be **efficiently verified**, then the problem can also be **efficiently solved** if nondeterminism may be used.

German: Raten und Prüfen

The Power of Nondeterminism

- ▶ Nondeterministic algorithms are very powerful because they can “guess” the “correct” computation step.
- ▶ Or, interpreted differently: they go through many possible computations “in parallel”, and it suffices if **one** of them is successful.
- ▶ Can they solve problems efficiently (in polynomial time) which deterministic algorithms **cannot** solve efficiently?
- ▶ **This is the big question!**

Summary (1)

- ▶ **Complexity theory** deals with the question which problems can be solved **efficiently** and which ones cannot.
- ▶ **here**: focus on what can be computed **in polynomial time**
- ▶ To formalize this, we use Turing machines, but other formalisms are **polynomially equivalent**.
- ▶ We consider **decision problems**, but the results directly transfer to general computational problems.

Summary (2)

important concept: **nondeterminism**

- ▶ **Nondeterministic algorithms** can “guess”, i. e., perform multiple computations “at the same time”.
- ▶ An input receives a “yes” answer if **at least one computation path** accepts it.
- ▶ in NTMs: with **nondeterministic transitions** ($\delta(q, a)$ contains multiple elements)
- ▶ in pseudo-code: with **GUESS statements**