Theoretical Computer Science II (ACS II)

9. Time complexity

Malte Helmert Andreas Karwath

Albert-Ludwigs-Universität Freiburg

January 27th, 2010

Theoretical Computer Science II (ACS II)

January 27th, 2010 — 9. Time complexity

A scenario

Example scenario

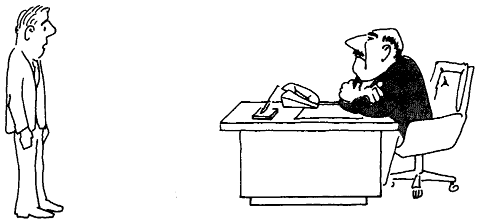
- You are a programmer working for a logistics company.
- Your boss asks you to implement a program that optimizes the travel route of your company's delivery truck:
  - The truck is initially located in your company's depot.
  - There are 50 locations the truck must visit on its route.
  - You know the travel distances between all locations (including the depot).
  - Your job is to write a program that determines a route from the depot via all locations back to the depot that minimizes total travel distance.

A scenario (ctd.)

Example scenario (ctd.)

- You try solving the problem for weeks, but don't manage to come up with a program. All your attempts either
  - cannot guarantee optimality or
  - don't terminate within reasonable time (say, a month of computation).
- What do you tell your boss?

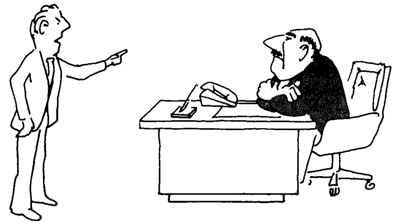
What you don't want to say



"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 ideally 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 ("easy problems") and which ones cannot ("hard problems").

- This is useful because different algorithmic techniques are required for problems for easy and hard problems.
- Moreover, if we can prove a problem to be hard, we should not waste our time looking for "easy" algorithms.

Why reductions?

Reductions

One important part of complexity theory are reductions that show how a new problem P can be expressed in terms of a known problem Q

- This is useful for theoretical analyses of P because it allows us to apply our knowledge about Q.
- It is also often useful for practical algorithms because we can use the best known algorithm for Q and apply it to P.

Complexity pop quiz

- The following slide contains a selection of graph problems.
- In all cases, the input is a directed, weighted graph $G = \langle V, A, w \rangle$ with positive edge weights.
- How hard do you think these graph problems are?
- Sort from easiest (requires least time to solve) to hardest (requires most time to solve).
- No justifications needed, just follow your intuition!

Some graph problems

1. Find a cycle-free path from $u \in V$ to $v \in V$ with minimum cost.
2. Find a cycle-free path from $u \in V$ to $v \in V$ with maximum cost
3. Determine if G is strongly connected (paths exist from everywhere to everywhere).
4. Determine if G is weakly connected (paths exist from everywhere to everywhere, ignoring arc directions).
5. Find a directed cycle.
6. Find a directed cycle involving all vertices.
7. Find a directed cycle involving a given vertex u.
8. Find a path visiting all vertices without repeating a vertex.
9. Find a path using all arcs without repeating an arc.

Overview of this chapter

Chapter overview:

- Refresher: asymptotic growth ("big-O notation")
- models of computation
- P and NP
- polynomial reductions
- NP-hardness and NP-completeness
- some NP-complete problems

Asymptotic growth: motivation

- Often, we are interested in how an algorithm behaves on large inputs, as these tend to be most critical in practice.
- For example, consider the following problem:

Duplicate elimination

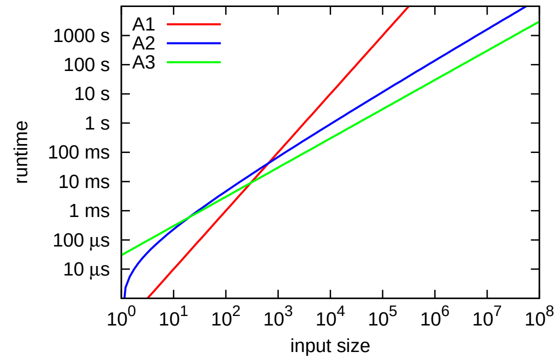
Input: a sequence of words $s_1, \ldots, s_n$ over some alphabet
Output: the same words, in any order, without duplicates

- Here are three algorithms for the problem:
  - A1 The naive algorithm with two nested for loops.
  - A2 Sort input; traverse sorted list and skip duplicates.
  - A3 Hash & report new entries upon insertion.
- Which one is fastest? Let's compare!

Runtimes for duplicate elimination algorithms

Assume that on an input with n words, the algorithms require the following amount of time (in μs):

A1 $f_1(n) = 0.1n^2$
A2 $f_2(n) = 10n \log n + 0.1n$
A3 $f_3(n) = 30n$



Runtime growth in the limit

- For very small inputs, A1 is faster than A2, which is faster than A3.
- However, for very large inputs, the ordering is opposite.
- Big-O notation captures this by considering how runtime grows in the limit of large input sizes.
- It also ignores constant factors, since for large enough inputs, these do not matter compared to differences in growth rate.

Big-O: Definition

Definition ($O(g)$)

Let $g : \mathbb{N}_0 \to \mathbb{R}$ be a function mapping from the natural numbers to the real numbers.

$O(g)$ is the set of all functions $f : \mathbb{N}_0 \to \mathbb{R}$ such that for some $c \in \mathbb{R}^+$ and $M \in \mathbb{N}_0$, we have $f(n) \leq c \cdot g(n)$ for all $n \geq M$.

In words: from a certain point onwards, f is bounded by g multiplied with some constant.

Intuition: If $f \in O(g)$, then f does not grow faster than g (maybe apart from constant factors that we do not care about).

Big-O: Notational conventions

"f is $O(g)$:"

- Formally, $O(g)$ is a set of functions, so to express that function f belongs to this class, we should write $f \in O(g)$.
- However, it is much more common to write $f = O(g)$ instead of $f \in O(g)$.
- In this context, "$=$" is pronounced "is", not "equals": "f is O of g."
- Note that this is not the usual meaning for "$=$".
- For example, it is not symmetric: we write $f = O(g)$, but not $O(g) = f$.

Further abbreviations:

- Notation like $f = O(g)$ where $g(n) = n^2$ is often abbreviated to $f = O(n^2)$.
- Similarly, if for example $f(n) = n \log n$, we can further abbreviate this to $n \log n \in O(n^2)$.

Big-O example (1)

Big-O example

Let $f(n) = 3n^2 + 14n + 7$.
We show that $f = O(n^2)$.

Big-O example (2)

Big-O example

Let $f(n) = 3n^2 + 14n + 7$.
We show that $f = O(n^3)$.

Big-O example (3)

Big-O example

Let $f(n) = n^{100}$.
We show that $f = O(2^n)$.
(We may use that $\log_2(x) \leq \sqrt{x}$ for all $x \geq 25$.)

Big-O for the duplicate elimination example

- In the duplicate elimination example, using big-O notation we can show that
  - $f_1 = O(n^2)$
  - $f_2 = O(n \log n)$
  - $f_3 = O(n)$

  which emphasizes the essential aspects of the different runtime growths for the algorithms.
- Moreover, big-O notation allows us to order the runtimes:
  - $f_3 = O(f_1)$, but not $f_1 = O(f_3)$
  - $f_2 = O(f_1)$, but not $f_1 = O(f_2)$
  - $f_3 = O(f_2)$, but not $f_2 = O(f_3)$

What is runtime complexity?

- Runtime complexity is a measure that tells us how much time we need to solve a problem.
- How do we define this appropriately?

Examples of different statements about runtime:

- "Running `sort /usr/share/dict/words` on computer `alfons` requires 0.242 seconds."
- "On an input file of size 1 MB, `sort` requires at most 1 second on a modern computer."
- "Quicksort is faster than Insertion sort."
- "Insertion sort is slow."

These are very different statements, each with different advantages and disadvantages.

Precise statements vs. general statements

Example statement about runtime

"Running `sort /usr/share/dict/words` on computer `alfons` requires 0.242 seconds."

Advantage: very precise

Disadvantage: not general

- input-specific:
  What if we want to sort other files?
- machine-specific:
  What if we run the program on another machine?
- even situation-specific:
  If we run the program again tomorrow, will we get the same result?

General statements about runtime

In this course, we want to make general statements about runtime. This is accomplished in three ways:

1. Rather than consider runtime for a particular input, we consider general classes of inputs:
   - Example: worst-case runtime to sort any input of size n
   - Example: average-case runtime to sort any input of size n

General statements about runtime

In this course, we want to make general statements about runtime. This is accomplished in three ways:

2. Rather than consider runtime on a particular machine, we consider more abstract cost measures:
   - Example: count executed x86 machine code instructions
   - Example: count executed Java bytecode instructions
   - Example: for sort algorithms, count number of comparisons

General statements about runtime

In this course, we want to make general statements about runtime. This is accomplished in three ways:

3. Rather than consider all implementation details, we ignore "unimportant" aspects:
   - Example: rather than saying that we need $4n - \lceil 1.2n \log n \rceil + 10$ instructions, we say that we need a linear number ($O(n)$) of instructions.

Which computational model do we use?

We know many models of computation:

- programs in some programming language
  - for example Java, C++, Scheme, ...
- Turing machines
  - Variants: single-tape or multi-tape
  - Variants: deterministic or nondeterministic
- push-down automata
- finite automata
  - variants: deterministic or nondeterministic

Here, we use Turing machines because they are the most powerful of our formal computation models.
(Programming languages are equally powerful, but not formal enough, and also too complicated.)

Are Turing machines an adequate model?

- According to the Church-Turing thesis, everything that can be computed can be computed by a Turing machine.
- However, many operations that are easy on an actual computer require a lot of time on a Turing machine.

$\rightsquigarrow$ Runtime on a Turing machine is not necessarily indicative of runtime on an actual machine!

- The main problem of Turing machines is that they do not allow random access.
- Alternative formal models of computation exist:
  - Examples: lambda calculus, register machines, random access machines (RAMs)
- Some of these are closer to how today's computers actually work (in particular, RAMs).

Turing machines are an adequate enough model

- So Turing machines are not the most accurate model for an actual computer.
- However, everything that can be done in a "more realistic model" in n computation steps can be done on a TM with at most polynomial overhead (e. g., in n^2 steps).
- For the big topic of this part of the course, the P vs. NP question, we do not care about polynomial overhead.
- Hence, for this purpose TMs are an adequate model, and they have the advantage of being easy to analyze.
- Hence, we use TMs in the following.

For more fine-grained questions (e. g., linear vs. quadratic algorithms), one should use a different computation model.

Which flavour of Turing machines do we use?

There are many variants of Turing machines:

- deterministic or nondeterministic
- one tape or multiple tapes
- one-way or two-way infinite tapes
- tape alphabet size: 2, 3, 4, ...

Which one do we use?

Deterministic or nondeterministic Turing machines?

- We earlier proved that deterministic TMs (DTMs) and nondeterministic ones (NTMs) have the same power.
- However, there we did not care about speed.
- The DTM simulation of an NTM we presented can cause an exponential slowdown.
- Are NTMs more powerful than DTMs if we care about speed, but don't care about polynomial overhead?
- Actually, that is the big question: it is one of the most famous open problems in mathematics and computer science.
- To get to the core of this question, we will consider both kinds of TM separately.

What about the other variations?

We do not have to consider other TM variations separately:

- Multi-tape TMs can be simulated on single-tape TMs with quadratic overhead.
- TMs with two-way infinite tapes can be simulated on TMs with one-way infinite tapes with constant-factor overhead, and vice versa.
- TMs with tape alphabets of any size K can be simulated on TMs with tape alphabet $\{0, 1, \Box\}$ with constant-factor overhead $\lceil \log_2 K \rceil$.

$\rightsquigarrow$ Whenever we want a simple model, we can limit ourselves to single-tape one-way infinite TMs with $\Sigma = \{0, 1\}$ and $\Gamma = \Sigma \cup \{\Box\}$.

Nondeterministic Turing machines

Definition (nondeterministic Turing machine)

A nondeterministic Turing machine (NTM) is a 6-tuple $\langle \Sigma, \square, Q, q_0, q_{acc}, \delta \rangle$, where

- Σ is the finite, non-empty input alphabet (often $\{0, 1\}$)
- $\square \notin \Sigma$ is the blank symbol
  - $\Sigma_\square := \Sigma \cup \{\square\}$ is the tape alphabet
- Q is the finite set of states
- $q_0 \in Q$ is the initial state, $q_{acc} \in Q$ the accepting state
  - $Q' := Q \setminus \{q_{acc}\}$ is the set of nonterminal states
- $\delta \subseteq (Q' \times \Sigma_\square) \times (Q \times \Sigma_\square \times \{-1, +1\})$ is the transition relation

Deterministic Turing machines

Definition (deterministic Turing machine)

An NTM $\langle \Sigma, \square, Q, q_0, q_{acc}, \delta \rangle$ is called deterministic (a DTM) if for all $q \in Q'$, $a \in \Sigma_\square$ there is exactly one triple $\langle q', a', \Delta \rangle$ with $\langle \langle q, a \rangle, \langle q', a', \Delta \rangle \rangle \in \delta$.

We then denote this triple with $\delta(q, a)$.

Note: In this definition, a DTM is a special case of an NTM, so if we define something for all NTMs, it is automatically defined for DTMs.

Turing machine configurations

Definition (configuration)

Let $M = \langle \Sigma, \square, Q, q_0, q_{acc}, \delta \rangle$ be an NTM.

A configuration of M is a triple $\langle w, q, x \rangle \in \Sigma_\square^* \times Q \times \Sigma_\square^+$.

- w: tape contents before tape head
- q: current state
- x: tape contents after and including tape head

Turing machine transitions

Definition (yields relation)

Let $M = \langle \Sigma, \square, Q, q_0, q_{acc}, \delta \rangle$ be an NTM.

A configuration c of M yields a configuration c' of M, in symbols $c \vdash c'$, as defined by the following rules, where $a, a', b \in \Sigma_\square$, $w, x \in \Sigma_\square^*$, $q, q' \in Q$ and $\langle \langle q, a \rangle, \langle q', a', \Delta \rangle \rangle \in \delta$:

$$
\begin{aligned}
\langle w, q, ax \rangle &\vdash \langle wa', q', x \rangle && \text{if } \Delta = +1, |x| \geq 1 \\
\langle w, q, a \rangle &\vdash \langle wa', q', \square \rangle && \text{if } \Delta = +1 \\
\langle wb, q, ax \rangle &\vdash \langle w, q', ba'x \rangle && \text{if } \Delta = -1 \\
\langle \epsilon, q, ax \rangle &\vdash \langle \epsilon, q', \square a'x \rangle && \text{if } \Delta = -1
\end{aligned}
$$

Acceptance of configurations

Definition (acceptance of configurations within time n)

Let c be a configuration of an NTM M.
Acceptance within time n is inductively defined as follows:

- If $c = \langle w, q_{acc}, x \rangle$ where q_{acc} is the accepting state of M, then M accepts c within time n for all $n \in \mathbb{N}_0$.
- If $c \vdash c'$ and M accepts c' within time $n - 1$, then M accepts c within time n.

Acceptance of words and languages

Definition (acceptance of words within time n)

Let $M = \langle \Sigma, \Box, Q, q_0, q_{acc}, \delta \rangle$ be an NTM.

M accepts the word $w \in \Sigma^*$ within time $n \in \mathbb{N}_0$ iff M accepts $\langle \epsilon, q_0, w \rangle$ within time n.

- Special case: M accepts ϵ within time $n \in \mathbb{N}_0$ iff M accepts $\langle \epsilon, q_0, \Box \rangle$ within time n.

Definition (acceptance of languages within time f)

Let M be an NTM with input alphabet Σ.
Let $f : \mathbb{N}_0 \to \mathbb{N}_0$.

M accepts the language $L \subseteq \Sigma^*$ within time f iff M accepts each word $w \in L$ within time at most $f(|w|)$, and M does not accept any word $w \notin L$.

P and NP

Definition (P and NP)

P is the set of all languages L for which there exists a DTM M and a polynomial p such that M accepts L within time p.

NP is the set of all languages L for which there exists an NTM M and a polynomial p such that M accepts L within time p.

Notes:

- Sets of languages like P and NP that are defined in terms of resource bounds for TMs are called complexity classes.
- We know that $\mathrm{P} \subseteq \mathrm{NP}$. (Why?)
- Whether the converse holds is an open problem: this is the famous P vs. NP question.

General algorithmic problems vs. decision problems

- An important aspect of complexity theory is to compare the difficulty of solving different algorithmic problems.
  - Examples: sorting, finding shortest paths, finding cycles in graphs including all vertices, ...
- Solutions to algorithmic problems take different forms.
  - Examples: a sorted sequence, a path, a cycle, ...
- To simplify the study, it is common in complexity theory to limit attention to decision problems, i. e., problems where the "solution" is an answer of the form Yes or No.
  - Examples: Is this sequence sorted?
    Is there a path from u to v of cost at most K?
    Is there a cycle in this graph that includes all vertices?
- If we pick the decision problems properly, we can usually show that if the decision problem is easy to solve, then the corresponding algorithmic problem is also easy to solve.

Decision problems: example

Using decision problems to solve more general problems

[O] Shortest path optimization problem:

- Input: Directed, weighted graph $G = \langle V, A, w \rangle$ with positive edge weights $w : A \to \mathbb{N}_1$, vertices $u \in V$, $v \in V$.
- Output: A shortest (= minimum-cost) path from u to v

[D] Shortest path decision problem:

- Input: Directed, weighted graph $G = \langle V, A, w \rangle$ with positive edge weights $w : A \to \mathbb{N}_1$, vertices $u \in V$, $v \in V$, cost bound $K \in \mathbb{N}_0$.
- Question: Is there a path from u to v with cost $\leq K$?

- If we can solve [O] in polynomial time, we can solve [D] in polynomial time and vice versa.

Decision problems as languages

Decision problems can be represented as languages:

- For every decision problem, if we want to pose it to a computer (or other computational device), we must express the input as a word over some alphabet Σ.
- The language defined by the decision problem then contains a word $w \in \Sigma^*$ iff
  - w is a well-formed input for the decision problem, and
  - the correct answer for input w is Yes.

  Example (shortest path decision problem): $w \in \mathrm{SP}$ iff
  - the input properly describes G, u, v, K such that G is a graph, arc weights are positive, etc.
  - that graph G has a path of cost at most K from u to v

Decision problems as languages (ctd.)

- Since decision problems can be represented as languages, we do not distinguish between "languages" and (decision) "problems" from now on.
- For example, we can say that P is the set of all decision problems that can be solved in polynomial time by a DTM.
- Similarly, NP is the set of all decision problems that can be solved in polynomial time by an NTM.
  - From the definition of NTM acceptance, "solved" means
    - If w is a Yes instance, then the NTM has some polynomial-time accepting computation for w
    - If w is a No instance (or not a well-formed input), then the NTM never accepts it.

Example: HamiltonianCycle $\in$ NP

HamiltonianCycle $\in$ NP

The HamiltonianCycle problem is defined as follows:

Given: An undirected graph $G = \langle V, E \rangle$

Question: Does G contain a Hamiltonian cycle?

A Hamiltonian cycle is a path $\pi = \langle v_0, v_1, \ldots, v_n \rangle$ such that

- π is a path: for all $i \in \{0, \ldots, n-1\}$, $\{v_i, v_{i+1}\} \in E$
- π is a cycle: $v_0 = v_n$
- π is simple: $v_i \neq v_j$ for all $i, j \in \{1, \ldots, n\}$ with $i \neq j$
- π is Hamiltonian: for all $v \in V$, there exists $i \in \{1, \ldots, n\}$ such that $v = v_i$

We show that HamiltonianCycle $\in$ NP.

Guess and check

- The (nondeterministic) Hamiltonian Cycle algorithm illustrates a general design principle for NTMs: guess and check.
- NTMs can solve decision problems in polynomial time by
  - nondeterministically guessing a "solution" (also called "witness" or "proof") for the instance
  - deterministically verifying that the guessed witness indeed describes a proper solution, and accepting iff it does
- It is possible to prove that all decision problems in NP can be solved by an NTM using such a guess-and-check approach.

Polynomial reductions: idea

- Reductions are a very common and powerful idea in mathematics and computer science.
- The idea is to solve a new problem by reducing (mapping) it to one for which already now how to solve it.
- Polynomial reductions (also called Karp reductions) are an example of this in the context of decision problems.

Polynomial reductions

Definition (Polynomial reductions/Karp reductions)

Let $A \subseteq \Sigma^*$ and $B \subseteq \Sigma^*$ be decision problems for alphabet Σ.
We say that A is polynomially reducible to B, written $A \leq_p B$,
if there exists a DTM M with the following properties:

- M is polynomial-time
  - i. e., there is a polynomial p such that M stops within time $p(|w|)$ on any input $w \in \Sigma^*$.
- M reduces A to B
  - i. e., for all $w \in \Sigma^*$: $(w \in A \text{ iff } f_M(w) \in B)$,
  - where $f_M : \Sigma^* \to \Sigma^*$ is the function computed by M, i. e., when M is run on input $w \in \Sigma^*$, then $f_M(w)$ is the tape content of M after stopping, ignoring blanks

M is called a polynomial reduction from A to B.

Polynomial reductions are also called Karp reductions.

Polynomial reduction: example

HamiltonianCycle $\leq_p$ TSP

The TSP (Travelling Salesperson) problem is defined as follows:

Given: A finite nonempty set of locations L, a symmetric travel cost function $cost : L \times L \to \mathbb{N}_0$, a cost bound $K \in \mathbb{N}_0$

Question: Is there a tour of total cost at most K, i. e., a permutation $\langle l_1, \dots, l_n \rangle$ of the locations such that $\sum_{i=1}^{n-1} cost(l_i, l_{i+1}) + cost(l_n, l_1) \leq K$?

We show that HamiltonianCycle $\leq_p$ TSP.

Properties of polynomial reductions

Theorem (properties of polynomial reductions)

Let A, B, C be decision problems over alphabet Σ.

1. *If $A \leq_p B$ and $B \in P$, then $A \in P$.*
2. *If $A \leq_p B$ and $B \in NP$, then $A \in NP$.*
3. *If $A \leq_p B$ and $A \notin P$, then $B \notin P$.*
4. *If $A \leq_p B$ and $A \notin NP$, then $B \notin NP$.*
5. *If $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$.*

NP-hardness and NP-completeness

Definition (NP-hard, NP-complete)

Let B be a decision problem.
B is called NP-hard if $A \leq_p B$ for all problems $A \in$ NP.
B is called NP-complete if $B \in$ NP and B is NP-hard.

- NP-hard problems are "at least as hard" as all problems in NP.
- NP-complete problems are "the hardest" problems in NP.
- Do NP-complete problems exist?
- If $A \in$ P for any NP-complete problem A, then P = NP. Why?

SAT is NP-complete

Definition (SAT)

The SAT (satisfiability) problem is defined as follows:

Given: A propositional logic formula φ

Question: Is φ satisfiable?

Theorem (Cook, 1971)

SAT *is NP-complete.*

Proof.

SAT $\in$ NP: Guess and check.
SAT is NP-hard: This is more involved...
(Continued on next slide.)

NP-hardness proof for SAT

Proof (ctd.)

We must show that $A \leq_p \mathrm{SAT}$ for all $A \in$ NP.

Let $A \in$ NP. This means that there exists a polynomial p and an NTM M s.t. M accepts A within time p.

Let $w \in \Sigma^*$ be the input for A.

We must, in polynomial time, construct a propositional logic formula $f(w)$ s.t. $w \in A$ iff $f(w) \in \mathrm{SAT}$ (i. e., is satisfiable).

Idea: Construct a logical formula that encodes the possible configurations that M can reach from input w and which is satisfiable iff an accepting configuration is reached.

NP-hardness proof for SAT (ctd.)

Proof (ctd.)

Let $M = \langle \Sigma, \Box, Q, q_0, q_{acc}, \delta \rangle$ be the NTM for A. We assume (w.l.o.g.) that it never moves to the left of the initial position.
Let $w = w_1 \dots w_n \in \Sigma^*$ be the input for M.
Let p be the run-time bounding polynomial for M.
Let $N = p(n) + 1$ (w.l.o.g. $N \geq n$).

$\rightsquigarrow$ During any computation that takes time $p(n)$, M can only visit the first N tape cells.

$\rightsquigarrow$ We can encode any configuration of M that can possibly be part of an accepting configuration by denoting:

- what the current state of M is
- which of the tape cells $\{1, \dots, N\}$ is the current location of the tape head
- which of the symbols in $\Sigma_\Box$ is contained in each of the tape cells $\{1, \dots, N\}$

NP-hardness proof for SAT (ctd.)

Proof (ctd.)

Use these propositional variables in formula $f(w)$:

- $state_{t,q}$ ($t \in \{0, \dots, N\}$, $q \in Q$)
  $\rightsquigarrow$ encode Turing Machine state in t-th configuration
- $head_{t,i}$ ($t \in \{0, \dots, N\}$, $i \in \{1, \dots, N\}$)
  $\rightsquigarrow$ encode tape head location in t-th configuration
- $content_{t,i,a}$ ($t \in \{0, \dots, N\}$, $i \in \{1, \dots, N\}$, $a \in \Sigma_\Box$)
  $\rightsquigarrow$ encode tape contents in t-th configuration

Construct $f(w)$ in such a way that every satisfying assignment

- describes a sequence of configurations of the TM
- that starts from the initial configuration
- and reaches an accepting configuration
- and follows the transition rules in δ

NP-hardness proof for SAT (ctd.)

Proof (ctd.)

$oneof\ X := (\bigvee_{x \in X} x) \land \neg(\bigvee_{x \in X} \bigvee_{y \in X \setminus \{x\}} (x \land y))$

1. Describe a sequence of configurations of the TM:

$$\begin{aligned} Valid := \bigwedge_{t=0}^{N} \Big(& oneof\ \{state_{t,q} \mid q \in Q\} \land \\ & oneof\ \{head_{t,i} \mid i \in \{1, \dots, N\}\} \land \\ & \bigwedge_{i=1}^{N} oneof\ \{content_{t,i,a} \mid a \in \Sigma_\Box\} \Big) \end{aligned}$$

NP-hardness proof for SAT (ctd.)

Proof (ctd.)

2. Start from the initial configuration:

$$\begin{aligned} Init := {} & state_{0,q_0} \land \mathrm{head}_{0,1} \land \\ & \bigwedge_{i=1}^{n} content_{0,i,w_i} \land \bigwedge_{i=n+1}^{N} content_{0,i,\Box} \end{aligned}$$

NP-hardness proof for SAT (ctd.)

Proof (ctd.)

3. Reach an accepting configuration:

$$Accept := \bigvee_{t=0}^{N} state_{t,q_{acc}}$$

NP-hardness proof for SAT (ctd.)

Proof (ctd.)

4. Follow the transition rules in δ:

$$Trans := \bigwedge_{t=0}^{N-1} ((state_{t,q_{acc}} \rightarrow Noop_t) \wedge (\neg state_{t,q_{acc}} \rightarrow \bigvee_{R \in \delta} \bigvee_{i=1}^{N} Rule_{t,i,R}))$$

where ...

NP-hardness proof for SAT (ctd.)

Proof (ctd.)

4. Follow the transition rules in δ (ctd.):

$$\begin{aligned} Noop_t := & \bigwedge_{q \in Q} (state_{t,q} \rightarrow state_{t+1,q}) \wedge \\ & \bigwedge_{i=1}^{N} (head_{t,i} \rightarrow head_{t+1,i}) \wedge \\ & \bigwedge_{i=1}^{N} \bigwedge_{a \in \Sigma_\square} (content_{t,i,a} \rightarrow content_{t+1,i,a}) \end{aligned}$$

NP-hardness proof for SAT (ctd.)

Proof (ctd.)

4. Follow the transition rules in δ (ctd.):

$$\begin{aligned} & Rule_{t,i,\langle\langle q,a\rangle,\langle q',a',\Delta\rangle\rangle} := \\ & \quad (state_{t,q} \wedge state_{t+1,q'}) \wedge \\ & \quad (head_{t,i} \wedge head_{t+1,i+\Delta}) \wedge \\ & \quad (content_{t,i,a} \wedge content_{t+1,i,a'}) \wedge \\ & \quad \bigwedge_{j \in \{1,\dots,N\} \setminus \{i\}} \bigwedge_{a \in \Sigma_\square} (content_{t,j,a} \rightarrow content_{t+1,j,a}) \end{aligned}$$

(Replace by $\perp$ if $i + \Delta = 0$ or $i + \Delta = N + 1$: these correspond to situations where M leaves the "allowed" part of the tape.)

NP-hardness proof for SAT (ctd.)

Proof (ctd.)

Putting it all together:

Define $f(w) := \mathit{Valid} \wedge \mathit{Init} \wedge \mathit{Accept} \wedge \mathit{Trans}$.

- $f(w)$ can be computed in polynomial time in $|w|$.
- $w \in A$ iff M accepts w within time $p(|w|)$
  iff $f(w)$ is satisfiable
  iff $f(w) \in \mathrm{SAT}$

$\rightsquigarrow$ $A \leq_p \mathrm{SAT}$

Since $A \in \mathrm{NP}$ was chosen arbitrarily, we can conclude that SAT is NP-hard and hence NP-complete. $\square$

More NP-complete problems

- The proof of NP-hardness of SAT was rather involved.
- However, now that we have it, we can prove other problems NP-hard much more easily.
- Simply prove $A \leq_p B$ for some known NP-hard problem A (such as SAT). This immediately proves that B is NP-hard. Why?
- A huge number of problems are known to be NP-complete.
- Garey & Johnson's textbook "Computers and Intractability — A Guide to the Theory of NP-Completeness" (1979) lists several hundred such problems, with references to proofs.

3SAT is NP-complete

Definition (3SAT)

The $\mathrm{3SAT}$ problem is defined as follows:

Given: A propositional logic formula φ in CNF with at most three literals per clause.

Question: Is φ satisfiable?

Theorem

$\mathrm{3SAT}$ *is NP-complete.*

Proof.

$\mathrm{3SAT} \in \mathrm{NP}$: Guess and check.

$\mathrm{3SAT}$ is NP-hard: $\mathrm{SAT} \leq_p \mathrm{3SAT}$ ($\rightsquigarrow$ whiteboard) $\square$

CLIQUE is NP-complete

Definition (CLIQUE)

The CLIQUE problem is defined as follows:

Given: An undirected graph $G = \langle V, E\rangle$ and a number $K \in \mathbb{N}_0$

Question: Does G contain a clique of size at least K,
i. e., a vertex set $C \subseteq V$ with $|C| \geq K$
such that $\langle u, v\rangle \in E$ for all $u, v \in C$ with $u \neq v$?

Theorem

CLIQUE *is NP-complete.*

Proof.

CLIQUE $\in \mathrm{NP}$: Guess and check.

CLIQUE is NP-hard: $\mathrm{3SAT} \leq_p$ CLIQUE ($\rightsquigarrow$ whiteboard)

$\square$

IndSet is NP-complete

Definition (IndSet)

The IndSet problem is defined as follows:

Given: An undirected graph $G = \langle V, E \rangle$ and a number $K \in \mathbb{N}_0$

Question: Does G contain an independent set of size at least K, i. e., a vertex set $I \subseteq V$ with $|I| \geq K$ such that for all $u, v \in I$, $\langle u, v \rangle \notin E$?

Theorem

IndSet *is NP-complete.*

Proof.

IndSet $\in$ NP: Guess and check.

IndSet is NP-hard: Clique $\leq_p$ IndSet ($\rightsquigarrow$ exercises) □

VertexCover is NP-complete

Definition (VertexCover)

The VertexCover problem is defined as follows:

Given: An undirected graph $G = \langle V, E \rangle$ and a number $K \in \mathbb{N}_0$

Question: Does G contain an vertex cover of size at most K, i. e., a vertex set $C \subseteq V$ with $|C| \leq K$ s. t. for all $\langle u, v \rangle \in E$, we have $u \in C$ or $v \in C$?

Theorem

VertexCover *is NP-complete.*

Proof.

VertexCover $\in$ NP: Guess and check.

VertexCover is NP-hard: IndSet $\leq_p$ VertexCover ($\rightsquigarrow$ exercises) □

DirHamiltonianCycle is NP-complete

Definition (DirHamiltonianCycle)

The DirHamiltonianCycle problem is defined as follows:

Given: A directed graph $G = \langle V, A \rangle$

Question: Does G contain a directed Hamiltonian cycle (i. e., a cyclic path visiting each vertex exactly once)?

Theorem

DirHamiltonianCycle *is NP-complete.*

Proof sketch.

DirHamiltonianCycle $\in$ NP: Guess and check.

DirHamiltonianCycle is NP-hard:

3SAT $\leq_p$ DirHamiltonianCycle ($\rightsquigarrow$ next slides)

DirHamiltonianCycle is NP-complete (ctd.)

Proof sketch (ctd.)

3SAT $\leq_p$ DirHamiltonianCycle:

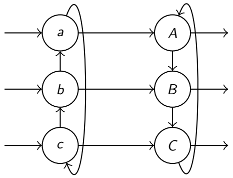
- A 3SAT instance φ is given.
- W.l.o.g. each clause has exactly three literals, and there are no repetitions within a clause.
- Let $v_1, \dots, v_n$ be the propositional variables in φ.
- Let $c_1, \dots, c_m$ be the clauses of φ, where each c_i is of the form $l_{i1} \vee l_{i2} \vee l_{i3}$.
- The reduction generates a graph $f(\varphi)$ with $6m + n$ vertices, described in the following.

DirHamiltonianCycle is NP-complete (ctd.)

Proof sketch (ctd.)

- Introduce vertex x_i with indegree 2 and outdegree 2 for each variable v_i:

- Introduce subgraph C_j with six vertices for each clause c_j:



DirHamiltonianCycle is NP-complete (ctd.)

Proof sketch (ctd.)

Let π be a directed Hamiltonian cycle of the overall graph.

- Whenever π traverses C_j, it must leave it at the corresponding "exit" for the given "entrance"
  (i. e., $a \longrightarrow A$, $b \longrightarrow B$, $c \longrightarrow C$).
  Otherwise π cannot be a Hamiltonian cycle.
- The following are all valid possibilities for Hamiltonian cycles in graphs containing C_j:
  - π crosses C_j once, entering at any entrance
  - π crosses C_j twice, entering at any two different entrances
  - π crosses C_j three times, entering once at each entrance

DirHamiltonianCycle is NP-complete (ctd.)

Proof sketch (ctd.)

Connect the "open ends" of the graph as follows:

- Identify the entrances and exits of the C_j graphs with the three literals of clause c_j.
- One exit of x_i is positive, one negative.
- For the positive exit, determine the clauses in which the positive literal v_i occurs
  - Connect the positive x_i exit to the v_i entrance of the C_j graph for the first such clause.
  - Connect the v_i exit of that graph to the x_i entrance of the second such clause, and so on.
  - Connect the v_i exit of the last such clause to the positive entrance of x_{i+1} (or x_1 if $n = 1$).
- Similarly for the negative exit of x_i and literal $\neg v_i$.

DirHamiltonianCycle is NP-complete (ctd.)

Proof sketch (ctd.)

This is a reduction (which is clearly polynomial):

- ($\Rightarrow$):
  - Given a satisfying truth assignment $\alpha(v_i)$, we can construct a Hamiltonian cycle by leaving x_i through the positive exit if $\alpha(v_i) = \mathbf{T}$; the negative exit if $\alpha(v_i) = \mathbf{F}$.
  - We can then visit all C_j graphs for clauses made true by that literal.
  - Overall, we visit each C_j graph 1–3 times.
- ($\Leftarrow$):
  - A Hamiltonian cycle visits each vertex x_i and leaves it through the positive or negative exit.
  - Set v_i to true or false according to which exit is chosen.
  - This gives a satisfying truth assignment.

□

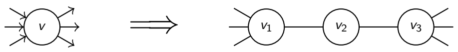
HamiltonianCycle is NP-complete

Theorem

HamiltonianCycle *is NP-complete.*

Proof sketch.

- HamiltonianCycle $\in$ NP : Guess and check.
- HamiltonianCycle is NP-hard:
  DirHamiltonianCycle $\leq_p$ HamiltonianCycle
- Basic gadget of the reduction:



□

TSP is NP-complete

Theorem

TSP *is NP-complete.*

Proof.

- TSP $\in$ NP : Guess and check.
- TSP is NP-hard:
  HamiltonianCycle $\leq_p$ TSP was already shown earlier.

□

And many, many more...

More NP-complete problems:

- SubsetSum: Given natural numbers $a_1, \dots, a_n$ and a target K, is there a subsequence with sum exactly K?
- BinPacking: Given objects of size $a_1, \dots, a_n$, can the objects fit into K bins with capacity B each?
- MineSweeperConsistency: In a given Minesweeper position, is a given cell safe?
- GeneralizedFreeCell: Does a given generalized FreeCell deal (i. e., one that may have more than 52 cards) have a solution?
- ...

Summary

- Complexity theory is about proving which problems are “easy” to solve and which ones are “hard”.
- Two important classes of problems are
  - P (problems that can be solved in polynomial time by a regular computing mechanism) and
  - NP (problems that can be solved in polynomial time using nondeterminism).
- We know P $\subseteq$ NP, but we do not know whether P $=$ NP.
- Many practically relevant problems are NP-complete, i. e., as hard as any other problem in NP.
- If there exists an efficient algorithm for one NP-complete problem, then there exists an efficient algorithm for all problems in NP.