COMP1600, week 12:

Problem Complexities

convenors: Dirk Pattinson, Pascal Bercher

lecturer: Pascal Bercher

slides based on those by: Dirk Pattinson

(with contributions by Victor Rivera and previous colleagues)

Semester 2, 2024

Australian National University

Overview of Week 12

- Motivation
- Non-Deterministic Turing Machines
- Big-$\mathcal{O}$ Notation
- Complexity Classes
- Reductions
- NP-Completeness

Motivation

Motivation

- So far, we only knew that what problems can be decided, semi-decided, or not even expressed.
- Now, we want to know "how quickly" problems can be solved.

Motivation

- So far, we only knew that what problems can be decided, semi-decided, or not even expressed.
- Now, we want to know "how quickly" problems can be solved.
- We investigate the computational hardness of *decision problems*:
  - What's the "performance" of the best-known algorithm for solving the respective problem?
  - Which problems are equally hard? Which ones are harder than others?
  - We look at how problems can be "turned into each other".

Motivation

- So far, we only knew that what problems can be decided, semi-decided, or not even expressed.
- Now, we want to know "how quickly" problems can be solved.
- We investigate the computational hardness of *decision problems*:
  - What's the "performance" of the best-known algorithm for solving the respective problem?
  - Which problems are equally hard? Which ones are harder than others?
  - We look at how problems can be "turned into each other".
- We measure "performance" in terms of a Turing Machine's:
  - Time requirement (number of operations/transitions)
  - Space requirement (number of cells that can be read/written)

Why should we care for Problem Complexities?

Given a new problem to solve, we:

- ... can use existing solvers instead of designing new ones,
  - → *Which software do you think is better? The one you design from scratch in a few weeks, or one that entire research communities (few or dozens to thousands of PhD students, post-docs, Professors) created over decades?*

Why should we care for Problem Complexities?

Given a new problem to solve, we:

- ... can use existing solvers instead of designing new ones,
  - → *Which software do you think is better? The one you design from scratch in a few weeks, or one that entire research communities (few or dozens to thousands of PhD students, post-docs, Professors) created over decades?*
- ... know performance bounds for our yet-to-be-designed solver
  - → *No need to look for an "efficient" algorithm if not a single genius so far was able to do that! (Well, don't let that stop you necessarily, see last point!) But it makes a great excuse. :)*

Why should we care for Problem Complexities?

Given a new problem to solve, we:

- ... can use existing solvers instead of designing new ones,
  - → *Which software do you think is better? The one you design from scratch in a few weeks, or one that entire research communities (few or dozens to thousands of PhD students, post-docs, Professors) created over decades?*
- ... know performance bounds for our yet-to-be-designed solver
  - → *No need to look for an "efficient" algorithm if not a single genius so far was able to do that! (Well, don't let that stop you necessarily, see last point!) But it makes a great excuse. :)*
- ... understand the problems we solve much better.
  - → *If you know that your (new) problem is equivalent to an existing (established) one, that surely helps... (Imagine, you take a course twice! The second time it's much easier...)*

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_4) \land (x_2 \lor \neg x_3) \land (\neg x_2 \lor x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:
  1. Enumerate all possible choices, e.g., using truth tables.
     Runtime?

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_4) \land (x_2 \lor \neg x_3) \land (\neg x_2 \lor x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:
  1. Enumerate all possible choices, e.g., using truth tables.
     Runtime? 2^n, i.e., exponential! Table doubles with each further variable

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:
  1. Enumerate all possible choices, e.g., using truth tables.
     Runtime? 2^n, i.e., exponential! Table doubles with each further variable
  2. Guess a valuation, then verify.
     What about Backtracking?

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:
  1. Enumerate all possible choices, e.g., using truth tables.
     Runtime? 2^n, i.e., exponential! Table doubles with each further variable
  2. Guess a valuation, then verify.
     What about Backtracking? Not required! Non-Determinism is always right!

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_4) \land (x_2 \lor \neg x_3) \land (\neg x_2 \lor x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:
  1. Enumerate all possible choices, e.g., using truth tables.
     Runtime? 2^n, i.e., exponential! Table doubles with each further variable
  2. Guess a valuation, then verify.
     What about Backtracking? Not required! Non-Determinism is always right!
     Runtime?

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:
  1. Enumerate all possible choices, e.g., using truth tables.
     Runtime? 2^n, i.e., exponential! Table doubles with each further variable
  2. Guess a valuation, then verify.
     What about Backtracking? Not required! Non-Determinism is always right!
     Runtime? Polytime for the verification, plus the guessing.

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:
  1. Enumerate all possible choices, e.g., using truth tables.
     Runtime? 2^n, i.e., exponential! Table doubles with each further variable
  2. Guess a valuation, then verify.
     What about Backtracking? Not required! Non-Determinism is always right!
     Runtime? Polytime for the verification, plus the guessing.
- So, what is the best runtime for deciding SAT?
  - The first "runtime class" will be called **EXPTIME** (membership)
  - The second will be called **NP** (**N** is for **n**on-deterministic)
  - $\rightarrow$ SAT is in ...

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_4) \land (x_2 \lor \neg x_3) \land (\neg x_2 \lor x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:
  1. Enumerate all possible choices, e.g., using truth tables.
     Runtime? 2^n, i.e., exponential! Table doubles with each further variable
  2. Guess a valuation, then verify.
     What about Backtracking? Not required! Non-Determinism is always right!
     Runtime? Polytime for the verification, plus the guessing.
- So, what is the best runtime for deciding SAT?
  - The first "runtime class" will be called **EXPTIME** (membership)
  - The second will be called **NP** (**N** is for **n**on-deterministic)
  - → SAT is in both **EXPTIME** and in **NP**, but **NP** is lower!
    Any idea *why* **NP** is lower?

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:
  1. Enumerate all possible choices, e.g., using truth tables.
     Runtime? 2^n, i.e., exponential! Table doubles with each further variable
  2. Guess a valuation, then verify.
     What about Backtracking? Not required! Non-Determinism is always right!
     Runtime? Polytime for the verification, plus the guessing.
- So, what is the best runtime for deciding SAT?
  - The first "runtime class" will be called **EXPTIME** (membership)
  - The second will be called **NP** (**N** is for **n**on-deterministic)
  - → SAT is in both **EXPTIME** and in **NP**, but **NP** is lower!
    Any idea *why* **NP** is lower? The guessing can be compiled away in exponential time! (Just try all options.)

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:
  1. Enumerate all possible choices, e.g., using truth tables.
     Runtime? 2^n, i.e., exponential! Table doubles with each further variable
  2. Guess a valuation, then verify.
     What about Backtracking? Not required! Non-Determinism is always right!
     Runtime? Polytime for the verification, plus the guessing.
- So, what is the best runtime for deciding SAT?
  - The first "runtime class" will be called **EXPTIME** (membership)
  - The second will be called **NP** (**N** is for **n**on-deterministic)
  - → SAT is in both **EXPTIME** and in **NP**, but **NP** is lower!
    Any idea *why* **NP** is lower? The guessing can be compiled away in exponential time! (Just try all options.)
- But is the problem also in **P**? (I.e., can we **P**-solve it without guessing?)

Non-Deterministic Turing Machines

Deterministic Turing Machines, Recap

A **Turing Machine** has the form $(S, s_0, F, \Gamma, \Sigma, B, \delta)$, where

- S is the set of **states** with $s_0 \in S$ the **initial state**;
- $F \subseteq S$ are the **final states**;
- Γ is the set of **tape symbols** (everything that might ever be on the tape);
- $B \in \Gamma \setminus \Sigma$ is the **blank symbol**;
- $\Sigma \subseteq \Gamma$ is the set of **input symbols**;
- δ is a (partial) **transition function**

$$\begin{array}{rcl} \delta \ : \ S \times \Gamma & \rightharpoonup & S \times \Gamma \times \{L, R, S\} \\ (\text{state}, \text{tape symbol}) & \mapsto & (\text{new state}, \text{new tape symbol}, \text{direction}) \end{array}$$

The direction tells the read/write head which way to go next:
Left, Right, or Stay/Stop. (Stopping the head is different from halting.)

Non-Deterministic Turing Machines

A **Turing Machine** has the form $(S, s_0, F, \Gamma, \Sigma, B, \delta)$, where

- S is the set of **states** with $s_0 \in S$ the **initial state**;
- $F \subseteq S$ are the **final states**;
- Γ is the set of **tape symbols** (everything that might ever be on the tape);
- $B \in \Gamma \setminus \Sigma$ is the **blank symbol**;
- $\Sigma \subseteq \Gamma$ is the set of **input symbols**; // so far: all the same!

Non-Deterministic Turing Machines

A **Turing Machine** has the form $(S, s_0, F, \Gamma, \Sigma, B, \delta)$, where

- S is the set of **states** with $s_0 \in S$ the **initial state**;
- $F \subseteq S$ are the **final states**;
- Γ is the set of **tape symbols** (everything that might ever be on the tape);
- $B \in \Gamma \setminus \Sigma$ is the **blank symbol**;
- $\Sigma \subseteq \Gamma$ is the set of **input symbols**; // so far: all the same!
- δ is a (partial) **transition function**

$$\begin{array}{rcl} \delta \;:\; S \times \Gamma & \rightharpoonup & 2^{S \times \Gamma \times \{L,R,S\}} \\ (\text{state}, \text{tape symbol}) & \mapsto & (\text{new state}, \text{new tape symbol}, \text{direction}) \end{array}$$

(We could also have formalized this with a transition relation.) The direction tells the read/write head which way to go next: Left, Right, or Stay/Stop. (Stopping the head is different from halting.)

Fundamental Properties of Non-Det. TMs

Basic Properties/Definitions

- If a TM reaches a *final* state, it acceptes the input word.
  (Same as for deterministic TMs, but now we have many branches/traces!)

Fundamental Properties of Non-Det. TMs

Basic Properties/Definitions

- If a TM reaches a *final* state, it acceptes the input word. (Same as for deterministic TMs, but now we have many branches/traces!)
- A word is rejected iff it is not accepted. (Again the same as for deterministic TMs!)

Fundamental Properties of Non-Det. TMs

Basic Properties/Definitions

- If a TM reaches a *final* state, it acceptes the input word. (Same as for deterministic TMs, but now we have many branches/traces!)
- A word is rejected iff it is not accepted. (Again the same as for deterministic TMs!)
- In complexity theory, we consider only decidable problems, where we can assume that all TMs halt (on all computation traces).

Fundamental Properties of Non-Det. TMs

Basic Properties/Definitions

- If a TM reaches a *final* state, it acceptes the input word. (Same as for deterministic TMs, but now we have many branches/traces!)
- A word is rejected iff it is not accepted. (Again the same as for deterministic TMs!)
- In complexity theory, we consider only decidable problems, where we can assume that all TMs halt (on all computation traces).

Language Definitions.

- The language of a Non-deterministic Turing Machine is the words accepted by it. (Just like for deterministic TMs.)
- Note how this now implies search: there might be many computation traces for an input; maybe just one accepts!

Fundamental Properties of Non-Det. TMs

Basic Properties/Definitions

- If a TM reaches a *final* state, it acceptes the input word. (Same as for deterministic TMs, but now we have many branches/traces!)
- A word is rejected iff it is not accepted. (Again the same as for deterministic TMs!)
- In complexity theory, we consider only decidable problems, where we can assume that all TMs halt (on all computation traces).

Language Definitions.

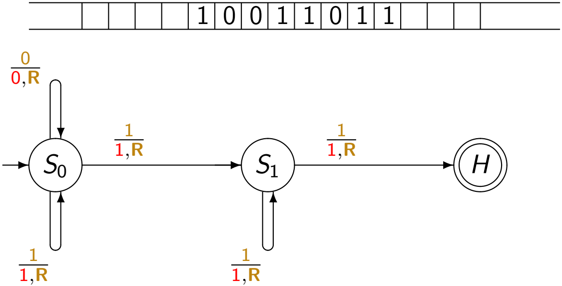
- The language of a Non-deterministic Turing Machine is the words accepted by it. (Just like for deterministic TMs.)
- Note how this now implies search: there might be many computation traces for an input; maybe just one accepts!

Relationship to Deterministic TMs.

- Non-det. TMs can't do *more* than deterministic ones.
- Non-det. TMs could be *quicker* than deterministic ones. (Unknown!)

Example (for a Non-Det. TM)

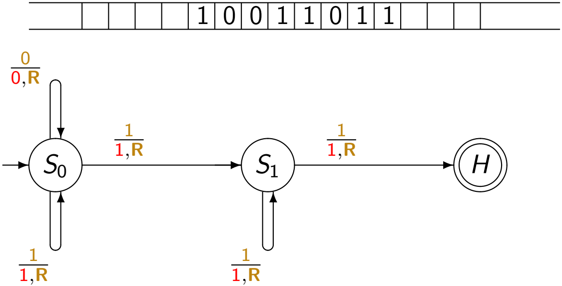
Consider the following TM, defined over an initial string over $\Sigma = \{0, 1\}$:



1. What does this TM do?
2. What language does it accept?

Example (for a Non-Det. TM)

Consider the following TM, defined over an initial string over $\Sigma = \{0, 1\}$:

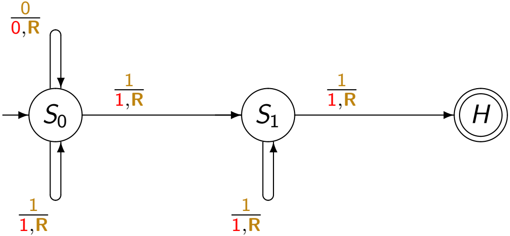


1. What does this TM do?
2. What language does it accept?

Checking for the "right" input.

Example (for a Non-Det. TM)

Consider the following TM, defined over an initial string over $\Sigma = \{0, 1\}$:



1. What does this TM do?
2. What language does it accept?

Checking for the "right" input.

$\{w \mid w \text{ contains} \geq 2 \text{ consecutive 1s}\}$
$= \{w11w' \mid w, w' \in \Sigma^*\}$

Big-$\mathcal{O}$ Notation

Introduction

- In the following we will define complexity classes based on whether some TM with specific properties exists.
- Example SAT: There is a non-det. TM that runs in "polynomial time" – what does that mean?
- We formalize this using the Big-$\mathcal{O}$ notation.
- That way we will know whether some function (the runtime or space consumption of a TM) is in polytime or exponential time etc.

Poll. Who of you knows the big-$\mathcal{O}$ notation already?

Example

Let's decide $L = \{0^i 1^i \mid i \in \mathbb{N}\}$ by TM M, i.e., check whether an arbitrary input has the form $0^i 1^i$ for some i. M does:

- Scan word w and reject if 10 is found.
- Repeat as long as there are 0s and 1s on the tape:
  - Replace both the leftmost 0 and the rightmost 1 with blanks.
- If either only 0s or 1s are left: reject, otherwise accept.

Example

Let's decide $L = \{0^i 1^i \mid i \in \mathbb{N}\}$ by TM M, i.e., check whether an arbitrary input has the form $0^i 1^i$ for some i. M does:

- Scan word w and reject if 10 is found.
- Repeat as long as there are 0s and 1s on the tape:
  - Replace both the leftmost 0 and the rightmost 1 with blanks.
- If either only 0s or 1s are left: reject, otherwise accept.

1. How much "time" does M need, as a function f of w's length?

Example

Let's decide $L = \{0^i 1^i \mid i \in \mathbb{N}\}$ by TM M, i.e., check whether an arbitrary input has the form $0^i 1^i$ for some i. M does:

- Scan word w and reject if 10 is found.
- Repeat as long as there are 0s and 1s on the tape:
  - Replace both the leftmost 0 and the rightmost 1 with blanks.
- If either only 0s or 1s are left: reject, otherwise accept.

1. How much "time" does M need, as a function f of w's length?
   f adheres $f(2k) = f(2(k-1)) + 4k + 1$, which is in $\mathcal{O}(n^2)$.

w	ϵ	01	$0^2 1^2$	$0^3 1^3$	$0^4 1^4$	$0^5 1^5$
$f(\|w\|)$	2	8	19	34	53	76

(*exact* numbers depend on implementation details)

Example

Let's decide $L = \{0^i1^i \mid i \in \mathbb{N}\}$ by TM M, i.e., check whether an arbitrary input has the form 0^i1^i for some i. M does:

- Scan word w and reject if 10 is found.
- Repeat as long as there are 0s and 1s on the tape:
  - Replace both the leftmost 0 and the rightmost 1 with blanks.
- If either only 0s or 1s are left: reject, otherwise accept.

1. How much "time" does M need, as a function f of w's length?
   f adheres $f(2k) = f(2(k-1)) + 4k + 1$, which is in $\mathcal{O}(n^2)$.

w	ϵ	01	0^21^2	0^31^3	0^41^4	0^51^5
$f(\lvert w \rvert)$	2	8	19	34	53	76

   (*exact* numbers depend on implementation details)
2. How much "space" does M need?

Example

Let's decide $L = \{0^i 1^i \mid i \in \mathbb{N}\}$ by TM M, i.e., check whether an arbitrary input has the form $0^i 1^i$ for some i. M does:

- Scan word w and reject if 10 is found.
- Repeat as long as there are 0s and 1s on the tape:
  - Replace both the leftmost 0 and the rightmost 1 with blanks.
- If either only 0s or 1s are left: reject, otherwise accept.

1. How much "time" does M need, as a function f of w's length?
   f adheres $f(2k) = f(2(k-1)) + 4k + 1$, which is in $\mathcal{O}(n^2)$.

w	ϵ	01	$0^2 1^2$	$0^3 1^3$	$0^4 1^4$	$0^5 1^5$
$f(\lvert w \rvert)$	2	8	19	34	53	76

   (*exact* numbers depend on implementation details)

2. How much "space" does M need? $\mathcal{O}(n)$

Example

Let's decide $L = \{0^i 1^i \mid i \in \mathbb{N}\}$ by TM M, i.e., check whether an arbitrary input has the form $0^i 1^i$ for some i. M does:

- Scan word w and reject if 10 is found.
- Repeat as long as there are 0s and 1s on the tape:
  - Replace both the leftmost 0 and the rightmost 1 with blanks.
- If either only 0s or 1s are left: reject, otherwise accept.

1. How much "time" does M need, as a function f of w's length?
   f adheres $f(2k) = f(2(k-1)) + 4k + 1$, which is in $\mathcal{O}(n^2)$.

w	ϵ	01	$0^2 1^2$	$0^3 1^3$	$0^4 1^4$	$0^5 1^5$
$f(\|w\|)$	2	8	19	34	53	76

   (*exact* numbers depend on implementation details)
2. How much "space" does M need? $\mathcal{O}(n)$

So in total, M has polynomial time and space restriction!

Time Complexity – Abstraction

Problem. Exact "number of steps function" usually *very complicated*

- for example, $2n^{17} + 23n^2 - 5$
- and hard to find in the first place (see last slide!).

Solution. Consider *approximate* number of steps

- focus on *asymptotic* behaviour
- as we are only interested in *large* problems

Idea. Abstract details away by just focussing on upper bounds
E.g., $f(n) = 2n^{17} + 23n^2 + 5 \in \mathcal{O}(n^{17})$

Time Complexity – Abstraction

Problem. Exact "number of steps function" usually *very complicated*

- for example, $2n^{17} + 23n^2 - 5$
- and hard to find in the first place (see last slide!).

Solution. Consider *approximate* number of steps

- focus on *asymptotic* behaviour
- as we are only interested in *large* problems

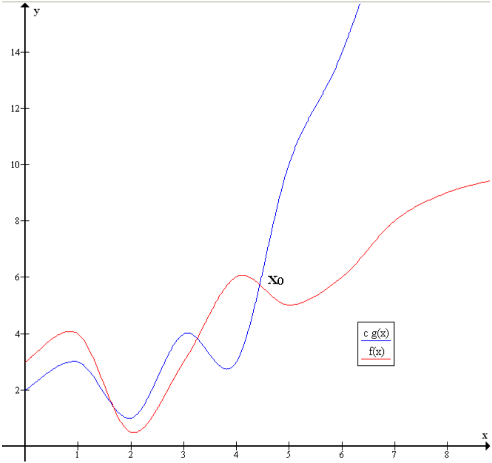
Idea. Abstract details away by just focussing on upper bounds
E.g., $f(n) = 2n^{17} + 23n^2 + 5 \in \mathcal{O}(n^{17})$

Big-$\mathcal{O}$ Notation. for f and g functions on natural numbers

- $f \in \mathcal{O}(g)$ if $\exists c.\ \exists n_0.\ \forall n \geq n_0.\ f(n) \leq c \cdot g(n)$
- "for large n, g is an upper bound to f up to a constant."
- E.g,. $f(n) \in \mathcal{O}(n^{17})$, since $g(n) = n^{17}$ and we can choose $c = 3$ so that we have $3n^{17} \geq f(n)$ for all $n \geq n_0$ (for a suitable n_0)

Graphical Illustration

Recall: $f \in \mathcal{O}(g)$ if $\exists c.\ \exists n_0.\ \forall n \geq n_0.\ f(n) \leq c \cdot g(n)$



Here, $f(x) \in \mathcal{O}(g(x))$ since $g(x)$ is at least as high as $f(x)$ for all $x \geq x_0$

Examples

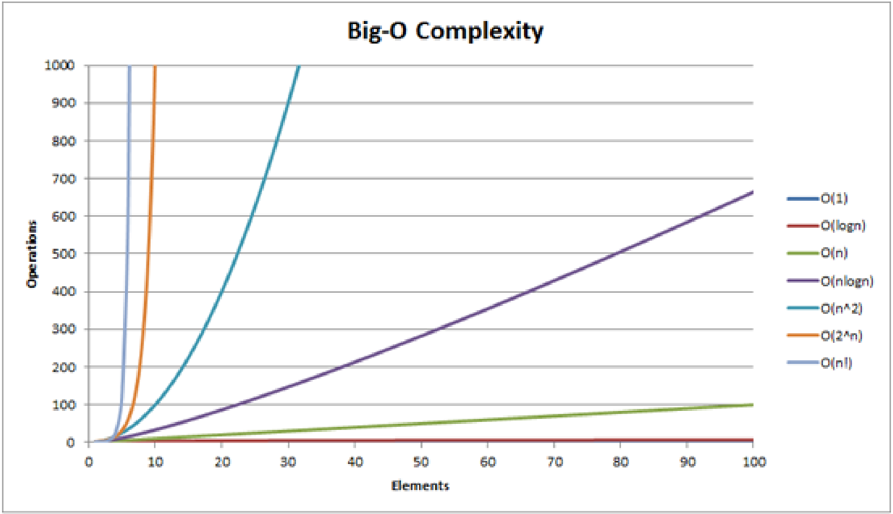
Examples.

- Polynomials: leading exponent dominates
- e.g. "$x^n +$ lower powers of x" $\in \mathcal{O}(x^n)$
- Exponentials: dominate polynomials
- e.g. "$2^n +$ polynomial" $\in \mathcal{O}(2^n)$

Important Special Cases.

- *linear*. f is linear if $f \in \mathcal{O}(n)$
- *polynomial*. f is polynomial if $f \in \mathcal{O}(n^k)$, for some k
- *exponential*. f is exponential if $f \in \mathcal{O}(2^n)$

Important Special Cases, Graphically



(Image copyright Lauren Kroner)

Complexity Classes

Complexity Classes

First some auxiliary definitions:

- **DTIME**$(t(n)) = \{L \mid \text{det. TM decides } L \text{ in } O(t(n)) \text{ time}\}$
- **NTIME**$(t(n)) = \{L \mid \text{non-det. TM decides } L \text{ in } O(t(n)) \text{ time}\}$

Complexity Classes

First some auxiliary definitions:

- **DTIME**$(t(n)) = \{L \mid$ det. TM decides L in $O(t(n))$ time $\}$
- **NTIME**$(t(n)) = \{L \mid$ non-det. TM decides L in $O(t(n))$ time $\}$
- **DSPACE**$(t(n)) = \{L \mid$ det. TM decides L with $O(t(n))$ space $\}$
- **NSPACE**$(t(n)) = \{L \mid$ non-det. TM decides L with $O(t(n))$ space $\}$

Complexity Classes

First some auxiliary definitions:

- $\mathbf{DTIME}(t(n)) = \{L \mid \text{det. TM decides } L \text{ in } O(t(n)) \text{ time }\}$
- $\mathbf{NTIME}(t(n)) = \{L \mid \text{non-det. TM decides } L \text{ in } O(t(n)) \text{ time }\}$
- $\mathbf{DSPACE}(t(n)) = \{L \mid \text{det. TM decides } L \text{ with } O(t(n)) \text{ space }\}$
- $\mathbf{NSPACE}(t(n)) = \{L \mid \text{non-det. TM decides } L \text{ with } O(t(n)) \text{ space }\}$

Now we can define some basic complexity classes:

- $\mathbf{P} = \bigcup_{k \in \mathbb{N}} \mathbf{DTIME}(n^k)$ $\mathbf{PSPACE} = \bigcup_{k \in \mathbb{N}} \mathbf{DSPACE}(n^k)$

Complexity Classes

First some auxiliary definitions:

- **DTIME**$(t(n)) = \{L \mid$ det. TM decides L in $O(t(n))$ time $\}$
- **NTIME**$(t(n)) = \{L \mid$ non-det. TM decides L in $O(t(n))$ time $\}$
- **DSPACE**$(t(n)) = \{L \mid$ det. TM decides L with $O(t(n))$ space $\}$
- **NSPACE**$(t(n)) = \{L \mid$ non-det. TM decides L with $O(t(n))$ space $\}$

Now we can define some basic complexity classes:

- **P** $= \bigcup_{k \in \mathbb{N}}$ **DTIME**(n^k) — **PSPACE** $= \bigcup_{k \in \mathbb{N}}$ **DSPACE**(n^k)
- **NP** $= \bigcup_{k \in \mathbb{N}}$ **NTIME**(n^k) — **NPSPACE** $= \bigcup_{k \in \mathbb{N}}$ **NSPACE**(n^k)

Complexity Classes

First some auxiliary definitions:

- $\textbf{DTIME}(t(n)) = \{L \mid \text{det. TM decides } L \text{ in } O(t(n)) \text{ time }\}$
- $\textbf{NTIME}(t(n)) = \{L \mid \text{non-det. TM decides } L \text{ in } O(t(n)) \text{ time }\}$
- $\textbf{DSPACE}(t(n)) = \{L \mid \text{det. TM decides } L \text{ with } O(t(n)) \text{ space }\}$
- $\textbf{NSPACE}(t(n)) = \{L \mid \text{non-det. TM decides } L \text{ with } O(t(n)) \text{ space }\}$

Now we can define some basic complexity classes:

- $\textbf{P} = \bigcup_{k\in\mathbb{N}} \textbf{DTIME}(n^k)$ $\qquad$ $\textbf{PSPACE} = \bigcup_{k\in\mathbb{N}} \textbf{DSPACE}(n^k)$
- $\textbf{NP} = \bigcup_{k\in\mathbb{N}} \textbf{NTIME}(n^k)$ $\qquad$ $\textbf{NPSPACE} = \bigcup_{k\in\mathbb{N}} \textbf{NSPACE}(n^k)$
- $\textbf{EXPTIME} = \bigcup_{k\in\mathbb{N}} \textbf{DTIME}(2^{n^k})$ $\qquad$ $\textbf{EXPSPACE} = \bigcup_{k\in\mathbb{N}} \textbf{DSPACE}(2^{n^k})$

Complexity Classes

First some auxiliary definitions:

- $\mathbf{DTIME}(t(n)) = \{L \mid \text{det. TM decides } L \text{ in } O(t(n)) \text{ time }\}$
- $\mathbf{NTIME}(t(n)) = \{L \mid \text{non-det. TM decides } L \text{ in } O(t(n)) \text{ time }\}$
- $\mathbf{DSPACE}(t(n)) = \{L \mid \text{det. TM decides } L \text{ with } O(t(n)) \text{ space }\}$
- $\mathbf{NSPACE}(t(n)) = \{L \mid \text{non-det. TM decides } L \text{ with } O(t(n)) \text{ space }\}$

Now we can define some basic complexity classes:

- $\mathbf{P} = \bigcup_{k \in \mathbb{N}} \mathbf{DTIME}(n^k)$ $\quad$ $\mathbf{PSPACE} = \bigcup_{k \in \mathbb{N}} \mathbf{DSPACE}(n^k)$
- $\mathbf{NP} = \bigcup_{k \in \mathbb{N}} \mathbf{NTIME}(n^k)$ $\quad$ $\mathbf{NPSPACE} = \bigcup_{k \in \mathbb{N}} \mathbf{NSPACE}(n^k)$
- $\mathbf{EXPTIME} = \bigcup_{k \in \mathbb{N}} \mathbf{DTIME}(2^{n^k})$ $\quad$ $\mathbf{EXPSPACE} = \bigcup_{k \in \mathbb{N}} \mathbf{DSPACE}(2^{n^k})$
- $\mathbf{NEXPTIME} = \bigcup_{k \in \mathbb{N}} \mathbf{NTIME}(2^{n^k})$ $\quad$ $\mathbf{NEXPSPACE} = \bigcup_{k \in \mathbb{N}} \mathbf{NSPACE}(2^{n^k})$

Complexity Classes

First some auxiliary definitions:

- **DTIME**$(t(n)) = \{L \mid$ det. TM decides L in $O(t(n))$ time $\}$
- **NTIME**$(t(n)) = \{L \mid$ non-det. TM decides L in $O(t(n))$ time $\}$
- **DSPACE**$(t(n)) = \{L \mid$ det. TM decides L with $O(t(n))$ space $\}$
- **NSPACE**$(t(n)) = \{L \mid$ non-det. TM decides L with $O(t(n))$ space $\}$

Now we can define some basic complexity classes:

- **P** $= \bigcup_{k \in \mathbb{N}}$ **DTIME**(n^k) **PSPACE** $= \bigcup_{k \in \mathbb{N}}$ **DSPACE**(n^k)
- **NP** $= \bigcup_{k \in \mathbb{N}}$ **NTIME**(n^k) **NPSPACE** $= \bigcup_{k \in \mathbb{N}}$ **NSPACE**(n^k)
- **EXPTIME** $= \bigcup_{k \in \mathbb{N}}$ **DTIME**(2^{n^k}) **EXPSPACE** $= \bigcup_{k \in \mathbb{N}}$ **DSPACE**(2^{n^k})
- **NEXPTIME** $= \bigcup_{k \in \mathbb{N}}$ **NTIME**(2^{n^k}) **NEXPSPACE** $= \bigcup_{k \in \mathbb{N}}$ **NSPACE**(2^{n^k})

We focus on classes **P** vs. **NP** vs. **EXPTIME**!
(The remaining ones are just listed for the sake of completeness.)

Relationships among Complexity Classes

How do time and space relate?

- On which TM can you solve harder/more problems? Using n movements or using n cells?

Relationships among Complexity Classes

How do time and space relate?

- On which TM can you solve harder/more problems? Using n movements or using n cells?
- In order to "use" a cell, we need to have a transition towards it.
- Thus, each space class can potentially contain more problems than their corresponding time class:

Relationships among Complexity Classes

How do time and space relate?

- On which TM can you solve harder/more problems? Using n movements or using n cells?
- In order to "use" a cell, we need to have a transition towards it.
- Thus, each space class can potentially contain more problems than their corresponding time class:
  - **P** $\subseteq$ **PSPACE** and **NP** $\subseteq$ **NPSPACE**
  - **EXPTIME** $\subseteq$ **EXPSPACE** and **NEXPTIME** $\subseteq$ **NEXPSPACE**

How do deterministic and non-deterministic classes relate?

Relationships among Complexity Classes

How do time and space relate?

- On which TM can you solve harder/more problems? Using n movements or using n cells?
- In order to "use" a cell, we need to have a transition towards it.
- Thus, each space class can potentially contain more problems than their corresponding time class:
  - **P** $\subseteq$ **PSPACE** and **NP** $\subseteq$ **NPSPACE**
  - **EXPTIME** $\subseteq$ **EXPSPACE** and **NEXPTIME** $\subseteq$ **NEXPSPACE**

How do deterministic and non-deterministic classes relate?

- Deterministic TMs are a special case of non-deterministic TMs.
- Thus, non-deterministic classes can potentially contain more problems than their corresponding deterministic classes:

Relationships among Complexity Classes

How do time and space relate?

- On which TM can you solve harder/more problems? Using n movements or using n cells?
- In order to "use" a cell, we need to have a transition towards it.
- Thus, each space class can potentially contain more problems than their corresponding time class:
  - **P** $\subseteq$ **PSPACE** and **NP** $\subseteq$ **NPSPACE**
  - **EXPTIME** $\subseteq$ **EXPSPACE** and **NEXPTIME** $\subseteq$ **NEXPSPACE**

How do deterministic and non-deterministic classes relate?

- Deterministic TMs are a special case of non-deterministic TMs.
- Thus, non-deterministic classes can potentially contain more problems than their corresponding deterministic classes:
  - **P** $\subseteq$ **NP** and **EXPTIME** $\subseteq$ **NEXPTIME**
  - **PSPACE** $\subseteq$ **NPSPACE** and **EXPSPACE** $\subseteq$ **NEXPSPACE**

Relationships among Complexity Classes: What's known

What's also known to the literature:

- **PSPACE** = **NPSPACE** and **EXPSPACE** = **NEXPSPACE** (Savitch's Theorem, 1970)
- **P** $\subsetneq$ **EXPTIME** (We know problems in **EXPTIME** which are provably not in **P**)

Relationships among Complexity Classes: What's known

What's also known to the literature:

- **PSPACE** = **NPSPACE** and **EXPSPACE** = **NEXPSPACE** (Savitch's Theorem, 1970)
- **P** $\subsetneq$ **EXPTIME** (We know problems in **EXPTIME** which are provably not in **P**)

So in total, we get:

- $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{(N)PSPACE} \subseteq \mathbf{EXPTIME} \subseteq \mathbf{NEXPTIME} \subseteq \mathbf{(N)EXPSPACE}$, where $\mathbf{P} \neq \mathbf{EXPTIME}$

For this course, only need to know about **P**, **NP**, and **EXPTIME**.

Relationships among Complexity Classes: What's known

What's also known to the literature:

- **PSPACE** = **NPSPACE** and **EXPSPACE** = **NEXPSPACE** (Savitch's Theorem, 1970)
- **P** $\subsetneq$ **EXPTIME** (We know problems in **EXPTIME** which are provably not in **P**)

So in total, we get:

- $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{(N)PSPACE} \subseteq \mathbf{EXPTIME} \subseteq \mathbf{NEXPTIME} \subseteq \mathbf{(N)EXPSPACE}$, with $\mathbf{P} \neq \mathbf{EXPTIME}$

For this course, only need to know about **P**, **NP**, and **EXPTIME**.

Note how every problem in **P** is also in **NP**. So if a problem is in **NP**, is it an *easy* one from **P** or a hard one like SAT? Hence: *Completeness!* (Later)

Reductions

Basic Definitions

This is the most important (and fun!) part of this week!

- We want to transform problems into each other – via *reduction*.
- I.e., we solve "our given problem" by turning it into a known one (which must be as least as hard; otherwise that's not possible).

Basic Definitions

This is the most important (and fun!) part of this week!

- We want to transform problems into each other – via *reduction*.
- I.e., we solve "our given problem" by turning it into a known one (which must be as least as hard; otherwise that's not possible).

Definition

$f : \Sigma^* \to \Sigma^*$ is a *polytime-computable* function if some polynomial time TM M exists that halts with just $f(w)$ on its tape, when started on any input $w \in \Sigma^*$.

Basic Definitions

This is the most important (and fun!) part of this week!

- We want to transform problems into each other – via *reduction*.
- I.e., we solve "our given problem" by turning it into a known one (which must be as least as hard; otherwise that's not possible).

Definition

$f : \Sigma^* \to \Sigma^*$ is a *polytime-computable* function if some polynomial time TM M exists that halts with just $f(w)$ on its tape, when started on any input $w \in \Sigma^*$.

Definition

$A \subseteq \Sigma_1^*$ is *polynomial time mapping-reducible* to $B \subseteq \Sigma_2^*$, written $A \leq_P B$, if a polytime-computable function $f : \Sigma_1^* \to \Sigma_2^*$ exists that is also a reduction (from A to B).

Reductions

Definition

- A reduction is a polynomial-time translation of the problem, say r.
- More precisely:
  1. $r(w)$ can be computed in time polynomial in $|w|$.
  2. $w \in A$ if and only if $r(w) \in B$ (so it "preserves the answer").

Example:

- $\text{EVEN} := \{n \mid n \bmod 2 = 0\}$, $\text{ODD} := \{n \mid n \bmod 2 = 1\}$
- Reduction from ODD to EVEN:
  - $r(k) = k + 1$, so we get $k \in \text{ODD}$ iff $r(k) \in \text{EVEN}$
  - So essentially we can define odd(n):=even($r(n)$) now.
  - This shows that EVEN is at least as hard as ODD.

Reductions

Definition

- A reduction is a polynomial-time translation of the problem, say r.
- More precisely:
  1. $r(w)$ can be computed in time polynomial in $|w|$.
  2. $w \in A$ if and only if $r(w) \in B$ (so it "preserves the answer").

Example:

- $\text{EVEN} := \{n \mid n \bmod 2 = 0\}$, $\text{ODD} := \{n \mid n \bmod 2 = 1\}$
- Reduction from ODD to EVEN:
  - $r(k) = k + 1$, so we get $k \in \text{ODD}$ iff $r(k) \in \text{EVEN}$
  - So essentially we can define odd(n):=even($r(n)$) now.
  - This shows that EVEN is at least as hard as ODD.
- If however our goal would have been to show that the 'new' problem ODD is at least as hard as EVEN, then we would have had to reduce from EVEN to ODD (though r would have been the same). Check this statement after "hardness" was introduced!

Example: Independent Set

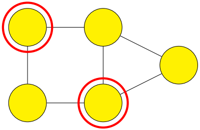
The Independent Set Problem:
Assume you want to throw a party. But you know that some of your friends don't get along. You only want to invite people that *do* get along.

Formalized as graph.

- vertices are your mates
- draw an edge between two vertices if people don't get along

Problem:
Given a graph and a $k \geq 0$, is there an *independent set*, i.e., a subset I of $\geq k$ vertices so that

- no two elements of I are connected with an edge.
- i.e., everybody in I gets along



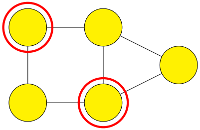
Example of an independent set of size 2 (*just* the red-circled vertices)

Solving the Independent Set Problem

Naive Implementation:

- loop through all subsets of size $\geq k$ (exponentially many!)
- and check whether they are independent sets

$\rightarrow$ Proves membership in **EXPTIME**



Solving the Independent Set Problem

Naive Implementation:

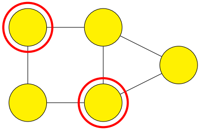
- loop through all subsets of size $\geq k$ (exponentially many!)
- and check whether they are independent sets

$\rightarrow$ Proves membership in **EXPTIME**

Using Non-deterministic Turing Machines:

- *guess* a subset of vertices of size $\geq k$
- *check* whether it is an independent set

$\rightarrow$ Proves membership in **NP**



Solving the Independent Set Problem

Naive Implementation:

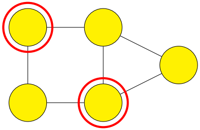
- loop through all subsets of size $\geq k$ (exponentially many!)
- and check whether they are independent sets

$\rightarrow$ Proves membership in **EXPTIME**

Using Non-deterministic Turing Machines:

- *guess* a subset of vertices of size $\geq k$
- *check* whether it is an independent set

$\rightarrow$ Proves membership in **NP**



Question: Can we do better? Is there a **P** algorithm?

Solving the Independent Set Problem

Naive Implementation:

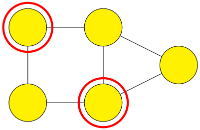
- loop through all subsets of size $\geq k$ (exponentially many!)
- and check whether they are independent sets

$\rightarrow$ Proves membership in **EXPTIME**

Using Non-deterministic Turing Machines:

- *guess* a subset of vertices of size $\geq k$
- *check* whether it is an independent set

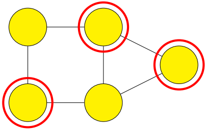
$\rightarrow$ Proves membership in **NP**



Question: Can we do better? Is there a **P** algorithm?
Answer: We don't know! But "hardness" helps giving a partial answer.

Example 2: Vertex Cover

Given a graph $G = \langle V, E \rangle$, a *vertex cover* is a set C of vertices such that every edge in G has at least one vertex in C.



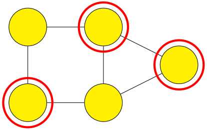
Example vertex cover:
The red-circled vertices.

Vertex Cover (Decision) Problem.

- Given graph $\langle V, E \rangle$ and $k \geq 0$, is there a vertex cover of size $\leq k$?
- $VC := \{(\langle V, E \rangle, k) \mid \langle V, E \rangle$ has a node cover $\leq k$, $k \in \mathbb{N}\ \}$

Example 2: Vertex Cover

Given a graph $G = \langle V, E \rangle$, a *vertex cover* is a set C of vertices such that every edge in G has at least one vertex in C.



Example vertex cover:
The red-circled vertices.

Vertex Cover (Decision) Problem.

- Given graph $\langle V, E \rangle$ and $k \geq 0$, is there a vertex cover of size $\leq k$?
- $VC := \{(\langle V, E \rangle, k) \mid \langle V, E \rangle$ has a node cover $\leq k$, $k \in \mathbb{N}\ \}$

Naive Algorithm:

- search through all subsets of size $\leq k$ (this is exponential)
- check whether it's a vertex cover

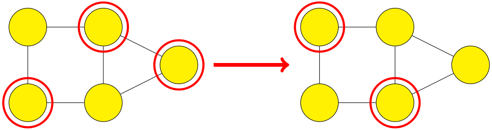
$\rightarrow$ This proves $VC \in$ **EXPTIME**, but we can do better!
(I.e., we could also guess and verify as before, giving $VC \in$ **NP**.)

From Independent Set to Vertex Cover

Reductions. Use solutions of one problem to solve another.

Observation. Let G be a graph with n vertices and $k \geq 0$.

- G has a VC of size $\leq k$ iff G has an IS of size $\geq n - k$



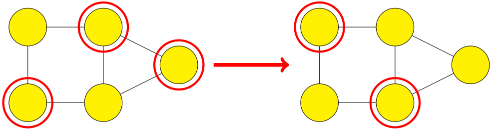
- Why?
  - VC with $\leq k$ vertices needs to cover *all* edges.
  - IS with $\geq n - k$ vertices can't cover *any* edge.

From Independent Set to Vertex Cover

Reductions. Use solutions of one problem to solve another.

Observation. Let G be a graph with n vertices and $k \geq 0$.

- G has a VC of size $\leq k$ iff G has an IS of size $\geq n - k$



- Why?
  - VC with $\leq k$ vertices needs to cover *all* edges.
  - IS with $\geq n - k$ vertices can't cover *any* edge.

What's the reduction? Vertex cover to independent set:

- $\langle G, k \rangle \in VC$ iff $r(\langle G, n \rangle) \in IS$, where $= r(\langle G, n \rangle) = \langle G, n - k \rangle$.
- Here, the reduction r only changes the number, but nothing else. But for most reductions, we will have to "translate problems", e.g., when turning a SAT problem into a VC problem (or vice versa)!

Important Note on Reductions

Be aware!

- So far, we only reduced problems, which were "equally hard", they were just "different flavors of the same problem":
  - EVEN vs. ODD
  - Independent Set (IS) vs. Vertex Cover (VC)

Important Note on Reductions

Be aware!

- So far, we only reduced problems, which were “equally hard”, they were just “different flavors of the same problem”:
  - EVEN vs. ODD
  - Independent Set (IS) vs. Vertex Cover (VC)
- But reductions also work (in one direction!) when one problem is “strictly harder” than another!
  - You should be able to reduce EVEN (or ODD) to Vertex Cover!

Important Note on Reductions

Be aware!

- So far, we only reduced problems, which were "equally hard", they were just "different flavors of the same problem":
  - EVEN vs. ODD
  - Independent Set (IS) vs. Vertex Cover (VC)
- But reductions also work (in one direction!) when one problem is "strictly harder" than another!
  - You should be able to reduce EVEN (or ODD) to Vertex Cover!
    (Reducing a problem in **P** to a problem that's **NP**-hard.)

(Hardness and completeness are explained in the next section...)

Important Note on Reductions

Be aware!

- So far, we only reduced problems, which were "equally hard", they were just "different flavors of the same problem":
  - EVEN vs. ODD
  - Independent Set (IS) vs. Vertex Cover (VC)
- But reductions also work (in one direction!) when one problem is "strictly harder" than another!
  - You should be able to reduce EVEN (or ODD) to Vertex Cover! (Reducing a problem in **P** to a problem that's **NP**-hard.)
  - You should be able to reduce Vertex Cover to the Sokoban game. (Reducing an **NP**-complete problem to one that's **PSPACE**-hard.)

(Hardness and completeness are explained in the next section...)

Membership, Hardness, Completeness

Membership, Hardness, and Completeness

Definition (NP completeness, NP membership, NP hardness)

A language B is ***NP**-complete* if

1. $B \in \mathbf{NP}$ $= \mathbf{NP}$ membership
2. *every* $A \in \mathbf{NP}$ is polytime-reducible to B. $= \mathbf{NP}$ hardness

Membership, Hardness, and Completeness

Definition (NP completeness, NP membership, NP hardness)

A language B is ***NP**-complete* if

1. $B \in \mathbf{NP}$ $= \mathbf{NP}$ membership
2. *every* $A \in \mathbf{NP}$ is polytime-reducible to B. $= \mathbf{NP}$ hardness

- So we have "for all A holds $A \leq_P B$", and therefore we know that B is "hard/expressive enough" to solve all other problems in **NP**. (Because we solve these other A-problems using our B-problem!)

Membership, Hardness, and Completeness

Definition (NP completeness, NP membership, NP hardness)

A language B is ***NP**-complete* if

1. $B \in$ **NP** = **NP** membership
2. *every* $A \in$ **NP** is polytime-reducible to B. = **NP** hardness

- So we have "for all A holds $A \leq_P B$", and therefore we know that B is "hard/expressive enough" to solve all other problems in **NP**. (Because we solve these other A-problems using our B-problem!)
- Therefore, **NP**-complete problems are the hardest ones in **NP**. (In particular they may be harder than those in **P**!)

Membership, Hardness, and Completeness

Definition (NP completeness, NP membership, NP hardness)

A language B is ***NP**-complete* if

1. $B \in \mathbf{NP}$ $\quad = \mathbf{NP}$ membership
2. *every* $A \in \mathbf{NP}$ is polytime-reducible to B. $\quad = \mathbf{NP}$ hardness

- So we have "for all A holds $A \leq_P B$", and therefore we know that B is "hard/expressive enough" to solve all other problems in **NP**. (Because we solve these other A-problems using our B-problem!)
- Therefore, **NP**-complete problems are the hardest ones in **NP**. (In particular they may be harder than those in **P**!)
- Hardness is the opposite of "practical exploitation of reductions": For hardness, reduce *from* a known problem rather than *to* one!

Motivation

Why are we interested in showing **NP**-hardness/completeness in the first place?

Motivation

Why are we interested in showing **NP**-hardness/completeness in the first place?

- If we fail in providing a **P** procedure for a new problem it could be:
  - Because we just did not discover it (yet? – keep searching!)
  - It doesn't even exist! (bail!)

Motivation

Why are we interested in showing **NP**-hardness/completeness in the first place?

- If we fail in providing a **P** procedure for a new problem it could be:
  - Because we just did not discover it (yet? – keep searching!)
  - It doesn't even exist! (bail!)
- So ... How to find out whether we should just work harder?

Motivation

Why are we interested in showing **NP**-hardness/completeness in the first place?

- If we fail in providing a **P** procedure for a new problem it could be:
  - Because we just did not discover it (yet? – keep searching!)
  - It doesn't even exist! (bail!)
- So ... How to find out whether we should just work harder?
  - If we can prove **NP**-completeness, then at least we know that nobody before you found a **P** procedure. (And maybe none even exists, which follows directly once somebody proves **P**$\neq$**NP**.)

Motivation

Why are we interested in showing **NP**-hardness/completeness in the first place?

- If we fail in providing a **P** procedure for a new problem it could be:
  - Because we just did not discover it (yet? – keep searching!)
  - It doesn't even exist! (bail!)
- So ... How to find out whether we should just work harder?
  - If we can prove **NP**-completeness, then at least we know that nobody before you found a **P** procedure. (And maybe none even exists, which follows directly once somebody proves **P**≠**NP**.)
- Why **NP**-completeness? Why not just showing **NP**-hardness?

Motivation

Why are we interested in showing **NP**-hardness/completeness in the first place?

- If we fail in providing a **P** procedure for a new problem it could be:
  - Because we just did not discover it (yet? – keep searching!)
  - It doesn't even exist! (bail!)
- So ... How to find out whether we should just work harder?
  - If we can prove **NP**-completeness, then at least we know that nobody before you found a **P** procedure. (And maybe none even exists, which follows directly once somebody proves **P**≠**NP**.)
- Why **NP**-completeness? Why not just showing **NP**-hardness?
  - Since the problem could be even harder! (E.g., **PSPACE**-hard, **EXPTIME**-hard, **NEXPTIME**-hard, . . . , and *infinitely* more!)
  - Each problem class has specific "properties". E.g., "**NP**-complete looks like Logic", "**PSPACE**-complete looks like planning", etc.

NP-Hardness

Theorem

*If B is **NP**-hard and $B \leq_P C$, then C is **NP**-hard.*

NP-Hardness

Theorem

*If B is **NP**-hard and $B \leq_P C$, then C is **NP**-hard.*

Corollary

*If B is **NP**-complete and $B \leq_P C$ for $C \in$ **NP**, then C is **NP**-complete.*

NP-Hardness

Theorem

*If B is **NP**-hard and $B \leq_P C$, then C is **NP**-hard.*

Corollary

*If B is **NP**-complete and $B \leq_P C$ for $C \in$ **NP**, then C is **NP**-complete.*

Proof.

Polynomial time reductions compose. □

NP-Hardness

Theorem

*If B is **NP**-hard and $B \leq_P C$, then C is **NP**-hard.*

Corollary

*If B is **NP**-complete and $B \leq_P C$ for $C \in$ **NP**, then C is **NP**-complete.*

Proof.

Polynomial time reductions compose. □

Important! This Corollary is of *major* importance!! Why?

NP-Hardness

Theorem

*If B is **NP**-hard and $B \leq_P C$, then C is **NP**-hard.*

Corollary

*If B is **NP**-complete and $B \leq_P C$ for $C \in$ **NP**, then C is **NP**-complete.*

Proof.

Polynomial time reductions compose. □

Important! This Corollary is of *major* importance!! Why?

$\rightarrow$ It gives us a convenient procedure to show **NP**-completeness!

- First, show **NP** membership. (That's almost always very easy.)
- Then, show hardness by grabbing an **NP**-complete problem and reduce it to yours!

Known **NP**-Complete Problems

List of known **NP**-complete problems:

- SAT (first problem proved **NP**-hard) and 3-SAT (see tutorials)
- Graph-Colourability and 3-Graph-Colourability (see tutorials)
- Independent Set and Vertex Cover (these slides)
- Hamiltonian path (not covered)
- Traveling Salesman Problem (not covered)
- Many more!

Known **NP**-Complete Problems

List of known **NP**-complete problems:

- SAT (first problem proved **NP**-hard) and 3-SAT (see tutorials)
- Graph-Colourability and 3-Graph-Colourability (see tutorials)
- Independent Set and Vertex Cover (these slides)
- Hamiltonian path (not covered)
- Traveling Salesman Problem (not covered)
- Many more!

Known problems in **P**:

- All regular languages! **Q.** Why?
- Of course, many more!

Known **NP**-Complete Problems

List of known **NP**-complete problems:

- SAT (first problem proved **NP**-hard) and 3-SAT (see tutorials)
- Graph-Colourability and 3-Graph-Colourability (see tutorials)
- Independent Set and Vertex Cover (these slides)
- Hamiltonian path (not covered)
- Traveling Salesman Problem (not covered)
- Many more!

Known problems in **P**:

- All regular languages! **Q.** Why?
- Of course, many more!

One of the most important problems in computer science is: $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$.

Summary

Summary Weeks 7-12

In weeks 7 to 11:

- We started with “machines” to recognizing only regular expressions.
- We added bits of computation power until we obtained a machine that can compute everything that’s possible. (Cf. Chosky Hierarchy.)
- On top languages from type 0 to 3, we also differentiate between recursive, recursively enumerable, and not recursively enumerable.

Summary Weeks 7-12

In weeks 7 to 11:

- We started with "machines" to recognizing only regular expressions.
- We added bits of computation power until we obtained a machine that can compute everything that's possible. (Cf. Chosky Hierarchy.)
- On top languages from type 0 to 3, we also differentiate between recursive, recursively enumerable, and not recursively enumerable.

In week 12:

- We looked at the "runtime" required to decide recursive problems.
- This is done in terms of transitions or space requirements, measured in terms of $|w|$ for the "best" Turing Machine (w is the word to decide).

Summary Weeks 7-12

In weeks 7 to 11:

- We started with "machines" to recognizing only regular expressions.
- We added bits of computation power until we obtained a machine that can compute everything that's possible. (Cf. Chosky Hierarchy.)
- On top languages from type 0 to 3, we also differentiate between recursive, recursively enumerable, and not recursively enumerable.

In week 12:

- We looked at the "runtime" required to decide recursive problems.
- This is done in terms of transitions or space requirements, measured in terms of $|w|$ for the "best" Turing Machine (w is the word to decide).
- We focused on complexity classes **P**, **NP**, and **EXPTIME**.
- Reductions turn one decision problem into another in polytime. Can be used for:

Summary Weeks 7-12

In weeks 7 to 11:

- We started with "machines" to recognizing only regular expressions.
- We added bits of computation power until we obtained a machine that can compute everything that's possible. (Cf. Chosky Hierarchy.)
- On top languages from type 0 to 3, we also differentiate between recursive, recursively enumerable, and not recursively enumerable.

In week 12:

- We looked at the "runtime" required to decide recursive problems.
- This is done in terms of transitions or space requirements, measured in terms of $|w|$ for the "best" Turing Machine (w is the word to decide).
- We focused on complexity classes **P**, **NP**, and **EXPTIME**.
- Reductions turn one decision problem into another in polytime. Can be used for:
  - Exploiting existing algorithms (reduce *to* known problem)
  - Prove hardness of your problem (reduce *from* known problem)

Summary Weeks 7-12

In weeks 7 to 11:

- We started with "machines" to recognizing only regular expressions.
- We added bits of computation power until we obtained a machine that can compute everything that's possible. (Cf. Chosky Hierarchy.)
- On top languages from type 0 to 3, we also differentiate between recursive, recursively enumerable, and not recursively enumerable.

In week 12:

- We looked at the "runtime" required to decide recursive problems.
- This is done in terms of transitions or space requirements, measured in terms of $|w|$ for the "best" Turing Machine (w is the word to decide).
- We focused on complexity classes **P**, **NP**, and **EXPTIME**.
- Reductions turn one decision problem into another in polytime. Can be used for:
  - Exploiting existing algorithms (reduce *to* known problem)
  - Prove hardness of your problem (reduce *from* known problem)
- Membership, hardness and completeness of problems.

Summary Weeks 7-12

In weeks 7 to 11:

- We started with "machines" to recognizing only regular expressions.
- We added bits of computation power until we obtained a machine that can compute everything that's possible. (Cf. Chosky Hierarchy.)
- On top languages from type 0 to 3, we also differentiate between recursive, recursively enumerable, and not recursively enumerable.

In week 12:

- We looked at the "runtime" required to decide recursive problems.
- This is done in terms of transitions or space requirements, measured in terms of $|w|$ for the "best" Turing Machine (w is the word to decide).
- We focused on complexity classes **P**, **NP**, and **EXPTIME**.
- Reductions turn one decision problem into another in polytime. Can be used for:
  - Exploiting existing algorithms (reduce *to* known problem)
  - Prove hardness of your problem (reduce *from* known problem)
- Membership, hardness and completeness of problems.

Conclusion

Some concluding words:

- I hope you enjoyed the course, and especially weeks 7 to 12! :)
- COMP3630, Theory of Computation, equals weeks 7 to 12 maxed out, for example:
  - The equivalence of different TM models is proved there (e.g., multi-tape TMs or semi-infinite TMs)
  - When we had proof sketches, the course covers the proof. (E.g., it shows that SAT is **NP**-hard (and hence **NP**-complete.)
  - It covers many more complexity classes, all mentioned before. And additional ones (esp. co-classes).
  - It focuses much more on doing reductions, e.g., proving problems **NP**-complete.

Conclusion

Some concluding words:

- I hope you enjoyed the course, and especially weeks 7 to 12! :)
- COMP3630, Theory of Computation, equals weeks 7 to 12 maxed out, for example:
  - The equivalence of different TM models is proved there (e.g., multi-tape TMs or semi-infinite TMs)
  - When we had proof sketches, the course covers the proof. (E.g., it shows that SAT is **NP**-hard (and hence **NP**-complete.)
  - It covers many more complexity classes, all mentioned before. And additional ones (esp. co-classes).
  - It focuses much more on doing reductions, e.g., proving problems **NP**-complete.
- Good luck in the exam!