Computational Complexity Tutorial

COMSOC 2017

Ronald de Haan

Plan for Today

- Tutorial on computational complexity theory:
  - Definition of complexity classes, in terms of time and space requirements of algorithms solving problems.
  - Hardness and completeness for complexity classes.
  - Proving NP-completeness.
  - Brief look at some complexity classes beyond NP.
- Focus will be on using complexity theory to analyze problems, rather than the theory itself.

Problems

- Computational problems are types of questions that might be solvable by computers.

  Examples:

  - Is the propositional formula $((a \vee b) \wedge (a \to c) \wedge (b \to c)) \to c$ a tautology?
  - Is there a path of length $\leq m$ between two given vertices in a given graph?

- Problems consist of an infinite set of inputs, together with the question that is to be answered for these inputs.

- The theory focuses mostly on decision problems: problems with a yes-no question.

  - A decision problem can be seen as a set of inputs: the set of all inputs for which the answer to the question is yes.

Example

- Problems typically look like this:

Reachability

Input: A directed graph $G = (V, E)$ and two vertices $v_1, v_2 \in V$.

Question: Is there a path in G from v_1 to v_2?

- It is easy to see that you can construct an algorithm to solve this problem.
- The question we are interested in is:
  how efficiently can we solve it?

How to measure complexity

- We measure the performance of algorithms.
- We can choose to measure one of several resources:
  - Time: how much time does it take to run the algorithm?
  - Space: how much memory does the algorithm need to run?
- There are several paradigms to choose from:
  - Worst-case analysis: how much resources will the algorithm use in the worst case?
  - Average-case analysis: how much resources will it use on average?
- The complexity of a problem is the complexity of the best algorithm that solves that problem.

Big-O Notation

- Take two functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$.
- Think of f as the function that specifies the amount $f(n)$ of time needed in the worst case to solve an input of size n. E.g., $f(n) = 2n^2 + 5n - 10$.
- Think of g as a "good approximation" of f and that is more convenient to talk about. E.g., $g(n) = n^2$.
- Big-O notation is a way of making this notion of approximation precise:
  - We say that $f(n)$ is in $O(g(n))$ iff there exist some $n_0 \in \mathbb{N}$ and some $c \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.
- In other words, from some n_0 onwards, the function f grows at most as fast as the function g, modulo some constant factor c that we don't really care about.

Tractability vs. Intractability

- An important (and often made) distinction is between tractable and intractable problems.
  - Tractable problems admit an algorithm that takes polynomial time—i.e., time $O(n^c)$ for some constant $c \in \mathbb{N}$.
  - Intractable problems are those for which algorithms need exponential time—e.g., $O(1.5^n)$ or $O(2^n)$.
- Some notes:
  - This is a theoretical distinction, that may not always correspond to which problem can be solved faster in practice. For example, an exponential algorithm running in time $2^{n/100}$ might behave better than a polynomial algorithm running in time n^{1000}. It turns out that for natural problems such odd functions do not come up. Also, for large enough n, the polynomial function will be smaller.
  - There are empirically successful algorithms for problems that are not known to be solvable in polynomial time. Such algorithms perform well on many inputs that come up in practice, but are not efficient in the general case.

Why is the notion of (in)tractability important?

> *Pretty well everybody outside the area of computer science thinks that if your program is running too slowly, what you need is a faster machine.*
>
> — *Rod Downey & Mike Fellows*

- We illustrate the difference between algorithms that run in time, say, $O(n^2)$ vs. algorithms that run in time, say, $O(2^n)$
- Time needed for 10^{10} steps per second:

n	n^2 steps	2^n steps
2	0.00000002 msec	0.00000002 msec
5	0.00000015 msec	0.00000019 msec
10	0.00001 msec	0.0001 msec
20	0.00004 msec	0.10 msec
50	0.00025 msec	31.3 hours
100	0.001 msec	9.4×10^{11} years
1000	0.100 msec	7.9×10^{282} years

- To compare: the # of atoms in the universe is around 10^{80}

Deterministic Complexity Classes

- A complexity class is a set of decision problems.

  Typically, a complexity classes are defined to contain problems with similar worst-case complexity bounds.

  - $\mathbf{TIME}(f(n))$ is the set of all decision problems that can be solved by an algorithm that runs in time $O(f(n))$.

    For example, $\textsc{Reachability} \in \mathbf{TIME}(n^2)$.

  - $\mathbf{SPACE}(f(n))$ is the set of all decision problems that can be solved by an algorithm that needs $O(f(n))$ memory space.

- These are sometimes called deterministic complexity classes, because the algorithms used to define them are deterministic.

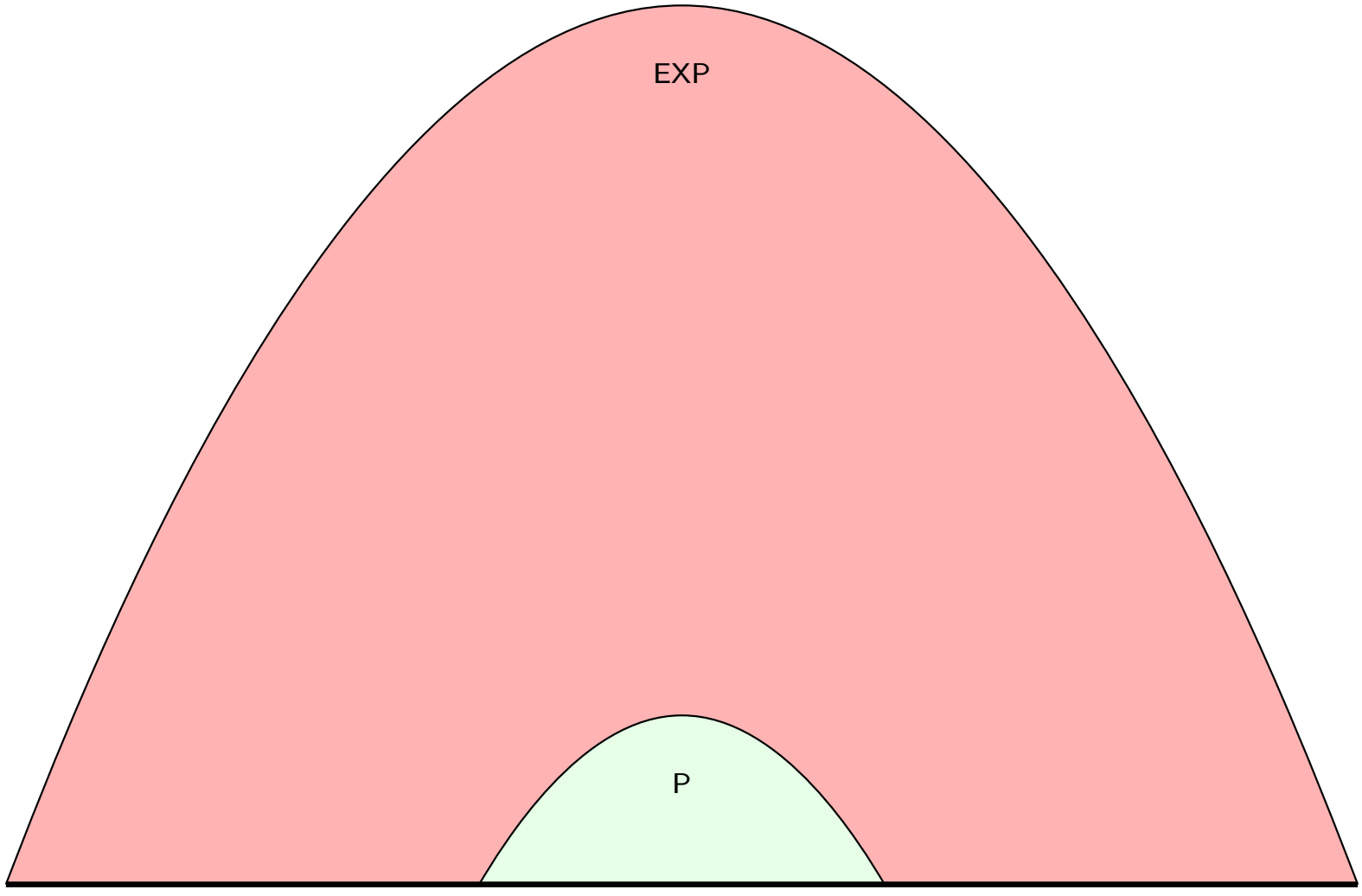
Two common deterministic complexity classes

- Two important deterministic complexity classes are:

$$\mathsf{P} = \bigcup_{k \in \mathbb{N}} \mathbf{TIME}(n^k)$$

$$\mathsf{EXP} = \bigcup_{k \in \mathbb{N}} \mathbf{TIME}(2^{(n^k)})$$

P and EXP (in a picture)



Example: Travelling Salesperson

- The decision problem variant of a famous problem:

Travelling Salesperson Problem (TSP)

Input: A list of n cities; for each pair of cities, the distance between them; and some $m \in \mathbb{N}$.

Question: Is there a route of length $\leq m$ that visits each city exactly once?

- One algorithm to solve TSP enumerates all possible routes that visit each city exactly once, and checks whether one of these is short enough. This takes time $O(n!)$.
- Slightly better algorithms are known, but all are exponential.
- Question: can we give evidence that no polynomial-time algorithm exists?

Answer: yes, using some more theory.

Nondeterministic Complexity Classes

- $\mathbf{NTIME}(f(n))$ is the set of all decision problems that can be solved by a **nondeterministic** algorithm that runs in time $O(f(n))$.
- $\mathbf{NSPACE}(f(n))$ is defined analogously.

- So what is a nondeterministic algorithm?
- Nondeterminism is about guessing a solution (and verifying that it is a correct solution).
- Many decision problems have a question of the form

  "Is there some X with property P?"

  Example: TSP. (Sometimes problems are already stated in this form, sometimes we can reformulate them in this form.)

Nondeterministic algorithms

- Nondeterministic machines:
  - Think of algorithms being implemented on a machine that moves from one state (memory configuration) to the next.

    For a nondeterministic algorithm the state transition function is underspecified (there can be more than one follow-up state).

    A machine solves a problem using a nondeterministic algorithm iff there exists a run that answers "yes."
  - We can think of this as a clairvoyant that tells us which is the best way to go at each choice-point in the algorithm.
- Nondeterminism as guess-and-check:
  - Another way to look at it: $\mathbf{NTIME}(f(n))$ consists of problems that can be solved by first guessing a candidate solution of size $O(f(n))$, and then in time $O(f(n))$ checking with a deterministic algorithm that it is really a valid solution.

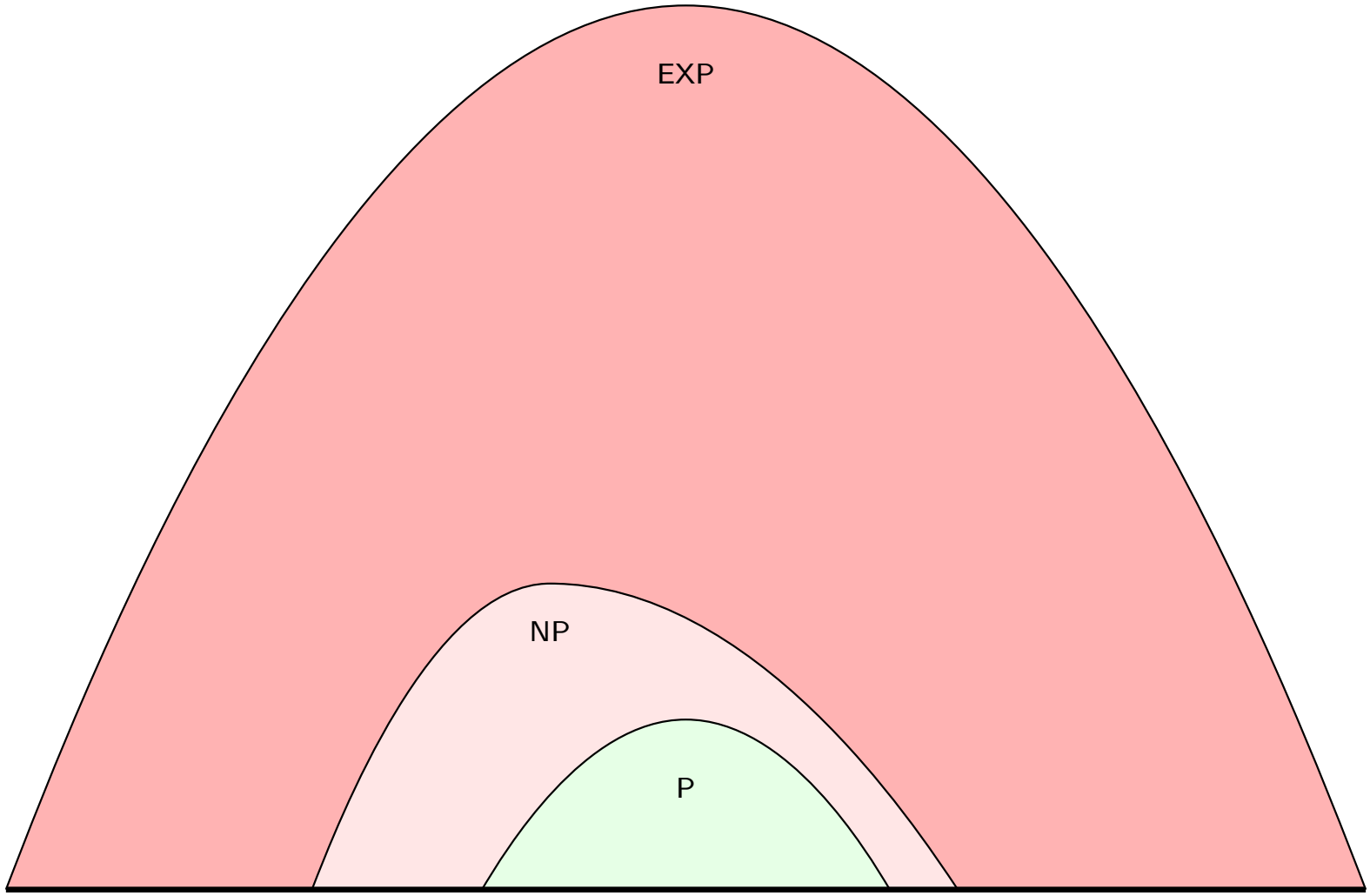
P and NP

- The class NP:

$$\mathrm{NP} = \bigcup_{k \in \mathbb{N}} \mathbf{NTIME}(n^k)$$

- For example, $\mathrm{TSP} \in \mathrm{NP}$.
  - Candidate solutions are routes that visit each city exactly once, i.e., permutations of $\{1, \ldots, n\}$. These are of size $O(n \log n)$.
  - Checking whether such a route is of length $\leq m$ can be done in (deterministic) polynomial time.

Adding NP to the picture



Other Common Complexity Classes

$$\mathsf{PSPACE} = \bigcup_{k \in \mathbb{N}} \mathbf{SPACE}(n^k)$$

$$\mathsf{NPSPACE} = \bigcup_{k \in \mathbb{N}} \mathbf{NSPACE}(n^k)$$

Relationships between Complexity Classes

- The following inclusions are known:

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} = \mathsf{NPSPACE} \subseteq \mathsf{EXP}$$

$$\mathsf{P} \subsetneq \mathsf{EXP}$$

- So at least one of the $\subseteq$'s above must be strict ($\subsetneq$), but we don't know which one. Most experts believe that they are all strict.
- In the case of $\mathsf{P} \overset{?}{\subsetneq} \mathsf{NP}$, the answer is worth \$1M.

Complements

- Let Q be a decision problem. The complement $\overline{Q}$ of Q is the set of all inputs that are no-inputs for Q.

  Example: SAT is the problem of deciding whether a given propositional logic formula is satisfiable. Its complement UNSAT is the problem of deciding whether a formula is not satisfiable.

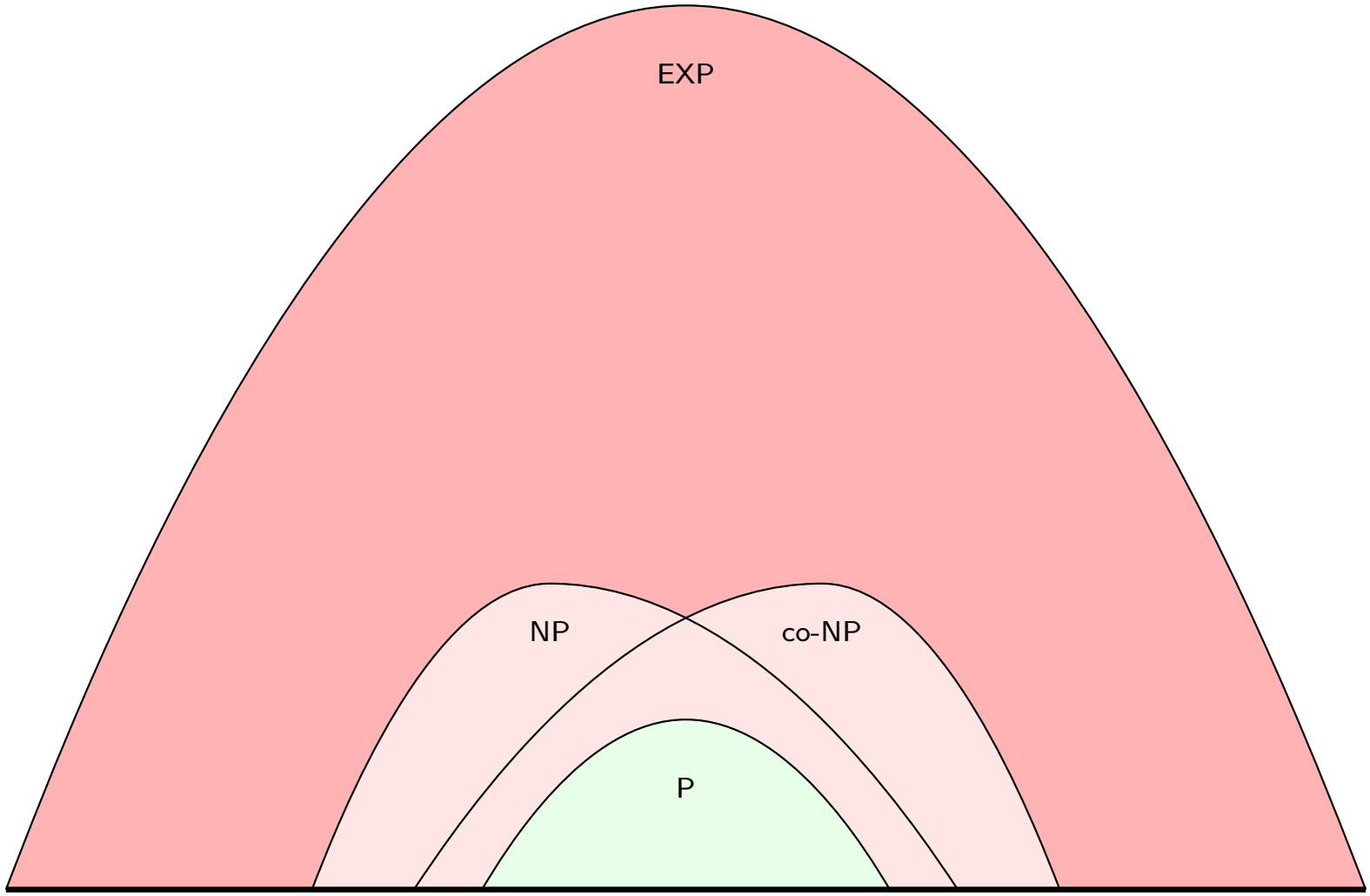
- For any complexity class K, the class co-K is defined as:

$$\text{co-K} = \{\, \overline{Q} : Q \in \text{K} \,\}$$

- For example, co-NP is the class of problems for which no-answers have a solution that can be guessed and checked in polynomial time.

- Clearly, P = co-P. It is unknown whether NP $\stackrel{?}{=}$ co-NP.

Adding co-NP to the picture



Polynomial-Time Reductions

- A reduction from some decision problem A to a decision problem B is an algorithm that translates any input x of A to an input y of B, such that the answer for y is the same as the answer for x.
- In other words, a reduction from A to B can be used with an algorithm for B to solve A.
- If the reduction runs in polynomial time, then it follows that B is at least as hard as A (modulo some polynomial-time overhead):
  - Suppose B is solvable in polynomial time. Then so is A.
  - I.e., if A is not solvable in polynomial time, then neither is B.
- So to prove that problem B is at least as hard as problem A:
  - Give a polynomial-time reduction from A to B.

Hardness and Completeness

- We can use polynomial-time reductions to identify the hardest problems in some complexity class.
- Let K be a complexity class.
  - A problem Q is K-hard if each $Q' \in \mathsf{K}$ is polynomial-time reducible to Q. That is, the K-hard problems include the hardest problems in K, and even harder ones.
  - A problem Q is K-complete if Q is K-hard and if $Q \in \mathsf{K}$. That is, these are the hardest problems in K, and only those.

The Cook-Levin Theorem

- The result that got complexity theory started (more or less), is that NP-complete problems actually exist, and that they are natural problems.
- The Cook-Levin Theorem: SAT is NP-complete.

Propositional Satisfiability (SAT)

Input: A propositional formula φ.

Question: Is φ satisfiable?

- This gives a useful starting point for proving other NP-completeness results:
  - If Q_1 is NP-hard, and you can reduce Q_1 to Q_2, then Q_2 must also be NP-hard.

NP-Completeness

- Recall our previous question: can we give evidence that problems such as TSP are not solvable in polynomial time?
- **Theorem**: TSP is NP-complete.

  Does this show that TSP does not admit a polynomial-time algorithm?
- **No!** We don't know if $\mathrm{P} = \mathrm{NP}$ or $\mathrm{P} \neq \mathrm{NP}$. If $\mathrm{P} = \mathrm{NP}$ turns out to be true, then TSP can be solved in polynomial time.
- **But!** If we use the working assumption that $\mathrm{P} \neq \mathrm{NP}$, then NP-complete problems cannot be solved in polynomial time.
- In other words, TSP is not solvable in polynomial time, unless $\mathrm{P} = \mathrm{NP}$.
- All NP-complete problems are equally hard. There are thousands of them, and nobody has found a polynomial-time algorithm for any of them.

NP-Completeness (in a picture)

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

M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman & Co., 1979.

Variants of Satisfiability

- If we restrict the structure of propositional formulas, then there is a chance that the satisfiability problem will become easier.

rSAT

Input: A formula φ in r-CNF (a conjunction of r-clauses).

Question: Is φ satisfiable?

(An r-clause is a disjunction of (at most) r literals.)

- **Theorem**: 3SAT is NP-complete.
- **Theorem**: 2SAT is in P.

MaxSAT

- Another variant of propositional satisfiability:

Maximum r-Satisfiability (Max$r$SAT)

Input: A set S of r-clauses, and some $m \in \mathbb{N}$.

Question: Is there a satisfiable $S' \subseteq S$ with $|S'| \geq m$?

- **Theorem**: Max2SAT is NP-complete.
  - *Proof sketch*:

    Max2SAT is in NP: you can guess some $S' \subseteq S$ together with a model α, and check in polynomial time that $|S'| \leq m$ and that α satisfies S'.

    Next, we show NP-hardness by reducing 3SAT to Max2SAT..

NP-Hardness of Max2SAT

- Consider the following 10 clauses:

$$(x), (y), (z), (w),$$
$$(\neg x \vee \neg y), (\neg y \vee \neg z), (\neg z \vee \neg x),$$
$$(x \vee \neg w), (y \vee \neg w), (z \vee \neg w)$$

- Any model satisfying $(x \vee y \vee z)$ can be extended to satisfy 7 of these clauses (and not more); all other models satisfy at most 6 of them.
- Given an input of $\mathrm{3SAT}$, construct an input of $\mathrm{Max2SAT}$:
  - For each clause $C_i = (x_i \vee y_i \vee z_i)$ in φ, write down these 10 clauses with a fresh w_i. If the input has n clauses, let $m = 7n$.
- Then φ is satisfiable iff (at least) m of the 2-clauses in the constructed input are (simultaneously) satisfiable.

Another Example: Independent Set

- Many conceptually simple problems that are NP-complete can be formulated as problems in graph theory, e.g.:

  Let $G = (V, E)$ be an undirected graph. An independent set is a set $I \subseteq V$ s.t. there are no edges between any of the vertices in I.

Independent Set (IS)

Input: An undirected graph $G = (V, E)$, and some $m \in \mathbb{N}$.

Question: Does G have an independent set I with $|I| \geq m$?

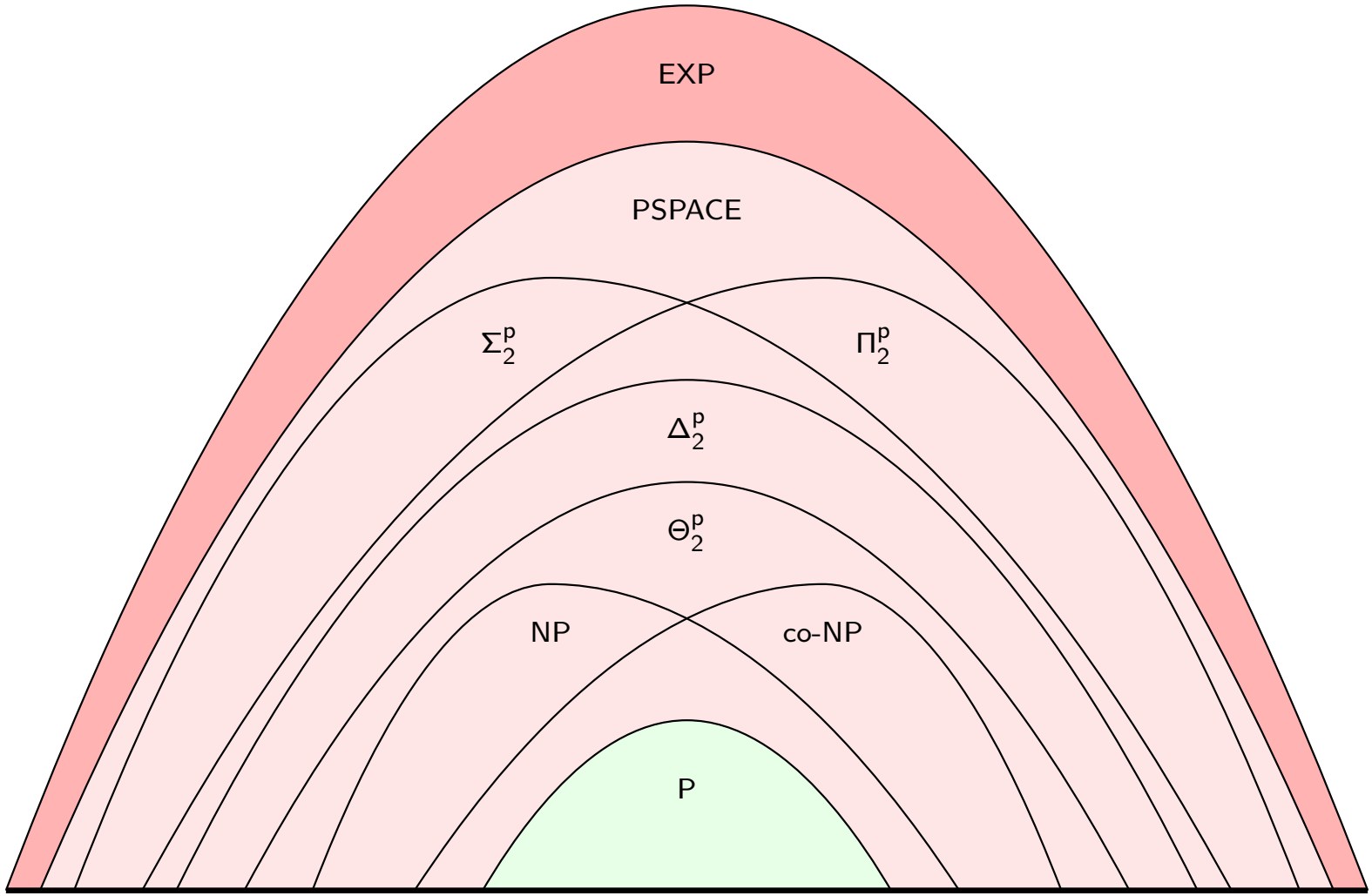
- **Theorem**: IS is NP-complete.
  - *Proof sketch*: NP-membership: easy.

    NP-hardness: by reduction from 3SAT. Given a conjunction φ of n 3-clauses, construct a graph $G = (V, E)$. V is the set of occurrences of literals in φ. Edges: make a "triangle" for each 3-clause, and connect complementary literals. Set $m = n$. Then φ is satisfiable iff there is an independent set of size m.

Oracles

- Imagine that you have access to an NP-oracle: a machine that can solve NP-complete problems in a single time step.
- Some complexity classes that are important for COMSOC:
  - $\Delta_2^{\mathrm{p}} = \mathrm{P}^{\mathrm{NP}}$: problems that can be decided in polynomial time by a machine with access to an NP oracle.
  - $\Theta_2^{\mathrm{p}} = \mathrm{P}_{||}^{\mathrm{NP}} = \mathrm{P}^{\mathrm{NP}}[\log]$: same, but all oracle queries need to be placed in parallel (equivalently, only log-many oracle queries).
  - $\Sigma_2^{\mathrm{p}} = \mathrm{NP}^{\mathrm{NP}}$: problems for which candidate solutions for a yes-instance can be checked in polynomial time with access to an NP-oracle.
    - *Example*: SAT for quantified Boolean formulas $\exists X.\forall Y.\psi$ is Σ_2^{p}-complete.
  - $\Pi_2^{\mathrm{p}} = \text{co-NP}^{\mathrm{NP}}$: complement of Σ_2^{p}.
    - *Example*: SAT for quantified Boolean formulas $\forall X.\exists Y.\psi$ is Π_2^{p}-complete.
- Σ_2^{p} and Π_2^{p} form the second level of the Polynomial Hierarchy

All classes that we've seen, in a picture



Summary

- We have seen:
  - Complexity classes P, NP, co-NP, PSPACE, EXP, ...
  - Relationships between complexity classes
  - Hardness and completeness for a complexity class
- Examples of NP-complete problems:
  - Logic: SAT, 3SAT, MAX2SAT
  - Graph Theory: INDEPENDENT SET
- You should be able to interpret complexity results, and to carry out simple reductions yourself to prove NP-completeness results.

Further Reading

- For quick look-ups of definitions:
  - The Complexity Zoo:
    `https://complexityzoo.uwaterloo.ca/Complexity_Zoo`
- Helpful textbooks include:
  - S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.
  - C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994.
  - M. Sipser. *Introduction to the Theory of Computation*. Course Technology, 1996.
- For large collections of NP-complete problems:
  - M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman & Co., 1979.
  - G. Ausiello et al. *Complexity and Approximation*. Springer, 1999.
    See also: `http://www.nada.kth.se/~viggo/wwwcompendium/`.