Languages and Algorithms for Artificial Intelligence (Third Module)

Between the Feasible and the Unfeasible

Ugo Dal Lago

ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA
Inria informatiques mathématiques

University of Bologna, Academic Year 2023/2024

The Border Between the Tractable and the Intractable

- Up to now, we have encountered (essentially speaking) two complexity classes, namely:
  - **P**, which contains those problems which can be solved in polynomial time, so the **tractable** ones.
  - **EXP**, which contains the whole of **P**, but also some problems which *cannot* be solved in polynomial time, intrinsically requiring exponential time (and as such, **intractable**).

The Border Between the Tractable and the Intractable

- Up to now, we have encountered (essentially speaking) two complexity classes, namely:
  - **P**, which contains those problems which can be solved in polynomial time, so the **tractable** ones.
  - **EXP**, which contains the whole of **P**, but also some problems which *cannot* be solved in polynomial time, intrinsically requiring exponential time (and as such, **intractable**).
- It's now time to study the "border" between tractability and intractability:
  - Between **P** and **EXP**, one can define *many other* classes. i.e. there are many ways of defining a class **A** such that

$$\mathbf{P} \subseteq \mathbf{A} \subseteq \mathbf{EXP}$$

  - This is a formidable tool to classify those problems in **EXP** for which *we do not know* whether they are in **P** (and there are *so many* of them).

Creating vs. Verifying

- Very often, the language we would like to classify can be written as follows:

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.(x,y) \in \mathcal{A}\}$$

where $p : \mathbb{N} \to \mathbb{N}$ and $\mathcal{A}$ is a set of pairs of strings.

Creating vs. Verifying

- Very often, the language we would like to classify can be written as follows:

  $$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.(x,y) \in \mathcal{A}\}$$

  where $p : \mathbb{N} \to \mathbb{N}$ and $\mathcal{A}$ is a set of pairs of strings.
- In other words, the elements of $\mathcal{L}$ are those strings for which we can find a *certificate* y (of polynomial length) such that the pair (x, y) passes the *test* $\mathcal{A} \subseteq \{0,1\}^* \times \{0,1\}^*$.

Creating vs. Verifying

- Very often, the language we would like to classify can be written as follows:

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.(x,y) \in \mathcal{A}\}$$

  where $p : \mathbb{N} \to \mathbb{N}$ and $\mathcal{A}$ is a set of pairs of strings.
- In other words, the elements of $\mathcal{L}$ are those strings for which we can find a *certificate* y (of polynomial length) such that the pair (x, y) passes the *test* $\mathcal{A} \subseteq \{0,1\}^* \times \{0,1\}^*$.
- What if $\mathcal{A}$ is itself decidable in polynomial time? Does this imply that $\mathcal{L}$ is itself decidable in polynomial time?

Creating vs. Verifying

- Very often, the language we would like to classify can be written as follows:

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.(x,y) \in \mathcal{A}\}$$

  where $p : \mathbb{N} \to \mathbb{N}$ and $\mathcal{A}$ is a set of pairs of strings.
- In other words, the elements of $\mathcal{L}$ are those strings for which we can find a *certificate* y (of polynomial length) such that the pair (x, y) passes the *test* $\mathcal{A} \subseteq \{0,1\}^* \times \{0,1\}^*$.
- What if $\mathcal{A}$ is itself decidable in polynomial time? Does this imply that $\mathcal{L}$ is itself decidable in polynomial time?
  - *Not necessarily*: given x, we can check whether $x \in \mathcal{L}$ by checking whether $(x, y) \in \mathcal{A}$ for all possible possible y such that $|y| \leq p(|x|)$, of which however there are *exponentially* many.
  - Of course this does *not* rule out other strategies to decide $\mathcal{L}$.

Creating vs. Verifying

- Very often, the language we would like to classify can be written as follows:

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.(x,y) \in \mathcal{A}\}$$

  where $p : \mathbb{N} \rightarrow \mathbb{N}$ and $\mathcal{A}$ is a set of pairs of strings.
- In other words, the elements of $\mathcal{L}$ are those strings for which we can find a *certificate* y (of polynomial length) such that the pair (x,y) passes the *test* $\mathcal{A} \subseteq \{0,1\}^* \times \{0,1\}^*$.
- What if $\mathcal{A}$ is itself decidable in polynomial time? Does this imply that $\mathcal{L}$ is itself decidable in polynomial time?
  - *Not necessarily*: given x, we can check whether $x \in \mathcal{L}$ by checking whether $(x,y) \in \mathcal{A}$ for all possible possible y such that $|y| \leq p(|x|)$, of which however there are *exponentially* many.
  - Of course this does *not* rule out other strategies to decide $\mathcal{L}$.
- The *take-away message* is thus the following: **crafting** a solution for the problem x (i.e., finding y) can potentially be more difficult than just **checking** y to be a solution to x.

The Complexity Class **NP**

- A language $\mathcal{L} \subseteq \{0,1\}^*$ is in the class **NP** iff there exist a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial time TM $\mathcal{M}$ such that

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.\mathcal{M}(\llcorner x, y \lrcorner) = 1\}$$

The Complexity Class **NP**

- A language $\mathcal{L} \subseteq \{0,1\}^*$ is in the class **NP** iff there exist a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial time TM $\mathcal{M}$ such that

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.\mathcal{M}(\llcorner x, y \lrcorner) = 1\}$$

- With the hypotheses above:
  - M is said to be the **verifier** for $\mathcal{L}$.
  - Any $y \in \{0,1\}^{p(|x|)}$ such that $\mathcal{M}(\llcorner (x,y) \lrcorner) = 1$ is said to be a **certificate** for x.

The Complexity Class **NP**

- A language $\mathcal{L} \subseteq \{0,1\}^*$ is in the class **NP** iff there exist a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial time TM $\mathcal{M}$ such that

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.\mathcal{M}(\llcorner x, y \lrcorner) = 1\}$$

- With the hypotheses above:
  - M is said to be the **verifier** for $\mathcal{L}$.
  - Any $y \in \{0,1\}^{p(|x|)}$ such that $\mathcal{M}(\llcorner(x,y)\lrcorner) = 1$ is said to be a **certificate** for x.
- Differently from **P** and **EXP**, the class **NP** does not have a natural counterpart as a class of *functions*.

The Complexity Class **NP**

- A language $\mathcal{L} \subseteq \{0,1\}^*$ is in the class **NP** iff there exist a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial time TM $\mathcal{M}$ such that

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.\mathcal{M}(\llcorner x, y \lrcorner) = 1\}$$

- With the hypotheses above:
  - M is said to be the **verifier** for $\mathcal{L}$.
  - Any $y \in \{0,1\}^{p(|x|)}$ such that $\mathcal{M}(\llcorner (x, y) \lrcorner) = 1$ is said to be a **certificate** for x.
- Differently from **P** and **EXP**, the class **NP** does not have a natural counterpart as a class of *functions*.

Theorem

$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$

Examples Problems in **NP**

1. **Maximum Independent Set**
   - In its decision form, it asks whether a pair $(\mathbb{G}, k)$ of an undirected graph and a natural number $k \in \mathbb{N}$ is such that $\mathbb{G} = (V, E)$ admits an independent set $W \subseteq V$ of cardinality at least k.

Examples Problems in **NP**

1. **Maximum Independent Set**
   - In its decision form, it asks whether a pair $(\mathbb{G}, k)$ of an undirected graph and a natural number $k \in \mathbb{N}$ is such that $\mathbb{G} = (V, E)$ admits an independent set $W \subseteq V$ of cardinality at least k.
   - Certificate: the certificate here is just W. Indeed, checking whether W is independent can easily be done in polynomial time.

Examples Problems in **NP**

1. **Maximum Independent Set**
   - In its decision form, it asks whether a pair $(\mathbb{G}, k)$ of an undirected graph and a natural number $k \in \mathbb{N}$ is such that $\mathbb{G} = (V, E)$ admits an independent set $W \subseteq V$ of cardinality at least k.
   - Certificate: the certificate here is just W. Indeed, checking whether W is independent can easily be done in polynomial time.
2. **Subset Sum**
   - It asks whether, given a sequence of natural numbers $n_1, \ldots, n_m$ and a number k, there exists a subset I of $\{1, \ldots, m\}$ such that $\sum_{i \in I} n_i = k$.

Examples Problems in **NP**

1. **Maximum Independent Set**
   - In its decision form, it asks whether a pair $(\mathbb{G}, k)$ of an undirected graph and a natural number $k \in \mathbb{N}$ is such that $\mathbb{G} = (V, E)$ admits an independent set $W \subseteq V$ of cardinality at least k.
   - Certificate: the certificate here is just W. Indeed, checking whether W is independent can easily be done in polynomial time.
2. **Subset Sum**
   - It asks whether, given a sequence of natural numbers $n_1, \ldots, n_m$ and a number k, there exists a subset I of $\{1, \ldots, m\}$ such that $\sum_{i \in I} n_i = k$.
   - Certificate: again, the certificate here is just I: checking whether $\sum_{i \in I} n_i = k$ just amounts to some additions and comparisons.

Examples Problems in **NP**

3. **Composite Numbers**
   - Given a number $n \in \mathbb{N}$, determine whether n is composite (i.e., not prime).

Examples Problems in **NP**

3. **Composite Numbers**
   - Given a number $n \in \mathbb{N}$, determine whether n is composite (i.e., not prime).
   - Certificate: it is the factorization of n, a pair (m, l) of natural numbers (greater than 2) such that $n = m \cdot l$.

Examples Problems in **NP**

3. **Composite Numbers**
   - Given a number $n \in \mathbb{N}$, determine whether n is composite (i.e., not prime).
   - Certificate: it is the factorization of n, a pair (m, l) of natural numbers (greater than 2) such that $n = m \cdot l$.
4. **Factoring**
   - Given three numbers n, m, l, it asks whether n has a prime factor in the interval $[m, l]$.

Examples Problems in **NP**

3. **Composite Numbers**
   - Given a number $n \in \mathbb{N}$, determine whether n is composite (i.e., not prime).
   - Certificate: it is the factorization of n, a pair (m, l) of natural numbers (greater than 2) such that $n = m \cdot l$.
4. **Factoring**
   - Given three numbers n, m, l, it asks whether n has a prime factor in the interval $[m, l]$.
   - Certificate: it can be taken to be the prime number p: checking that $p \in [m, l]$ and that p divides n is easy. Instead, checking that p is prime requires a lot of work, but can indeed be done in polynomial time.

Examples Problems in **NP**

3. **Decisional Linear Programming**
   - Given a sequence of m linear inequalities with rational coefficients over n variables (i.e. inequalities of the form $\sum_{i=1}^{n} a_i x_i \leq b$, where the coefficients $a_1, \ldots, a_n, b$ are in $\mathbb{Q}$), decide whether there is a rational assignment to the variables $x_1, \ldots, x_n$ which makes all the inequalities true.

Examples Problems in **NP**

3. **Decisional Linear Programming**
   - Given a sequence of m linear inequalities with rational coefficients over n variables (i.e. inequalities of the form $\sum_{i=1}^{n} a_i x_i \leq b$, where the coefficients $a_1, \ldots, a_n, b$ are in $\mathbb{Q}$), decide whether there is a rational assignment to the variables $x_1, \ldots, x_n$ which makes all the inequalities true.
   - Certificate: the assignment, which can be easily checked for correctness.

Examples Problems in **NP**

3. **Decisional Linear Programming**
   - Given a sequence of m linear inequalities with rational coefficients over n variables (i.e. inequalities of the form $\sum_{i=1}^{n} a_i x_i \leq b$, where the coefficients $a_1, \ldots, a_n, b$ are in $\mathbb{Q}$), decide whether there is a rational assignment to the variables $x_1, \ldots, x_n$ which makes all the inequalities true.
   - Certificate: the assignment, which can be easily checked for correctness.
4. **Decisional 0/1 Linear Programming**
   - Given a sequence of linear inequalities as above, decide whether there is an assignment *of zeros and ones* to the variables $x_1, \ldots, x_n$ rendering all the inequalities true.

Examples Problems in **NP**

3. **Decisional Linear Programming**
   - Given a sequence of m linear inequalities with rational coefficients over n variables (i.e. inequalities of the form $\sum_{i=1}^{n} a_i x_i \leq b$, where the coefficients $a_1, \ldots, a_n, b$ are in $\mathbb{Q}$), decide whether there is a rational assignment to the variables $x_1, \ldots, x_n$ which makes all the inequalities true.
   - Certificate: the assignment, which can be easily checked for correctness.
4. **Decisional 0/1 Linear Programming**
   - Given a sequence of linear inequalities as above, decide whether there is an assignment *of zeros and ones* to the variables $x_1, \ldots, x_n$ rendering all the inequalities true.
   - Certificate: again, the assignments suffices.

Examples Problems in **NP**

- Some of the aforementioned problems are also in **P**:
  - *Decisional linear programming* can be proved to be in **P** thanks to, e.g., the Ellipsoid algorithm.
  - The *composite numbers* problem can be proved itself to be in **P**, thanks to a breakthrough recent result, namely the so-called AKS algorithm.

Examples Problems in **NP**

- Some of the aforementioned problems are also in **P**:
  - *Decisional linear programming* can be proved to be in **P** thanks to, e.g., the Ellipsoid algorithm.
  - The *composite numbers* problem can be proved itself to be in **P**, thanks to a breakthrough recent result, namely the so-called AKS algorithm.
- All the other problems are currently **not known** to be in **P**.
  - Are they all equivalent in terms of their inherent computational difficulty?
  - Is there any way isolate those problems in **NP** whose difficult is maximal, i.e. they are at least as hard as all other problems in **NP**?

Nondeterministic Turing Machines

- The class **NP** can also be defined using a variant of Turing machines, called the *nondeterministic* Turing machines (NDTM for short).
  - This is the original definition by Hartmanis and Stearns, the founding fathers of computational complexity.
  - This is also the reason for the letter **N** in **NP**.

Nondeterministic Turing Machines

- The class **NP** can also be defined using a variant of Turing machines, called the *nondeterministic* Turing machines (NDTM for short).
  - This is the original definition by Hartmanis and Stearns, the founding fathers of computational complexity.
  - This is also the reason for the letter **N** in **NP**.
- The only differences between a NDTM and an ordinary TM is that the former has:
  - Two transition functions δ_0 and δ_1 rather than just one. *At every step*, the machine chooses nondeterministically one between the two transition functions and proceed according to it
  - A special state $q_{\mathtt{accept}}$.

Nondeterministic Turing Machines

- The class **NP** can also be defined using a variant of Turing machines, called the *nondeterministic* Turing machines (NDTM for short).
  - This is the original definition by Hartmanis and Stearns, the founding fathers of computational complexity.
  - This is also the reason for the letter **N** in **NP**.
- The only differences between a NDTM and an ordinary TM is that the former has:
  - Two transition functions δ_0 and δ_1 rather than just one. *At every step*, the machine chooses nondeterministically one between the two transition functions and proceed according to it
  - A special state $q_{\mathtt{accept}}$.
- We say that a NDTM M:
  - **Accepts** the input $x \in \{0,1\}^*$ iff *there exists* one among the many possible evolutions of the machine M when fed with x which makes it reaching $q_{\mathtt{accept}}$.
  - **Rejects** the input $x \in \{0,1\}^*$ iff *none* of the aformentioned evolutions leads to $q_{\mathtt{accept}}$.

Time-Bounded NDTMs

- We say that a NDTM M *runs in time* $T : \mathbb{N} \rightarrow \mathbb{N}$ iff for every $x \in \{0,1\}^*$ *and for every possible nondeterministic evolution*, M reaches either the halting state or $q_{\mathtt{accept}}$ within $c \cdot T(|x|)$ steps, where $c > 0$.

Time-Bounded NDTMs

- We say that a NDTM M *runs in time* $T : \mathbb{N} \to \mathbb{N}$ iff for every $x \in \{0,1\}^*$ *and for every possible nondeterministic evolution*, M reaches either the halting state or $q_{\mathtt{accept}}$ within $c \cdot T(|x|)$ steps, where $c > 0$.
- For every function $T : \mathbb{N} \to \mathbb{N}$ and $\mathcal{L} \subseteq \{0,1\}^*$, we say that $\mathcal{L} \in \mathbf{NDTIME}(T(n))$ iff there is a NDTM M working in time T and such that $M(x) = 1$ iff $x \in \mathcal{L}$.

Time-Bounded NDTMs

- We say that a NDTM M *runs in time* $T : \mathbb{N} \to \mathbb{N}$ iff for every $x \in \{0,1\}^*$ *and for every possible nondeterministic evolution*, M reaches either the halting state or $q_{\mathtt{accept}}$ within $c \cdot T(|x|)$ steps, where $c > 0$.
- For every function $T : \mathbb{N} \to \mathbb{N}$ and $\mathcal{L} \subseteq \{0,1\}^*$, we say that $\mathcal{L} \in \mathbf{NDTIME}(T(n))$ iff there is a NDTM M working in time T and such that $M(x) = 1$ iff $x \in \mathcal{L}$.

Theorem

$\mathbf{NP} = \cup_{c \in \mathbb{N}} \mathbf{NDTIME}(n^c)$

Time-Bounded NDTMs

- We say that a NDTM M *runs in time* $T : \mathbb{N} \to \mathbb{N}$ iff for every $x \in \{0,1\}^*$ *and for every possible nondeterministic evolution*, M reaches either the halting state or $q_{\mathtt{accept}}$ within $c \cdot T(|x|)$ steps, where $c > 0$.
- For every function $T : \mathbb{N} \to \mathbb{N}$ and $\mathcal{L} \subseteq \{0,1\}^*$, we say that $\mathcal{L} \in \mathbf{NDTIME}(T(n))$ iff there is a NDTM M working in time T and such that $M(x) = 1$ iff $x \in \mathcal{L}$.

Theorem

$\mathbf{NP} = \cup_{c \in \mathbb{N}} \mathbf{NDTIME}(n^c)$

- All this being said, NDTMs, contrarily to TMs, are *not* meant to model any form of physically realisable machine.

Are all Problems in **NP** Equivalent?

- The Maximum Independent Set, a problem we already know about, is in **NP**.

Are all Problems in **NP** Equivalent?

- The Maximum Independent Set, a problem we already know about, is in **NP**.
- The language of, say, palindrome words, is easy to be proved in **P**, thus in **NP**.

Are all Problems in **NP** Equivalent?

- The Maximum Independent Set, a problem we already know about, is in **NP**.
- The language of, say, palindrome words, is easy to be proved in **P**, thus in **NP**.
- Intuitively, however, the inherent difficulties of solving the two problems *should b*e different.

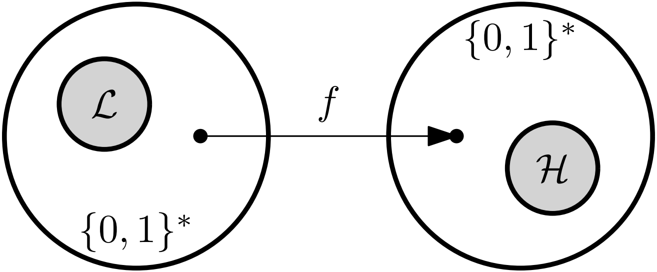
Are all Problems in **NP** Equivalent?

- The Maximum Independent Set, a problem we already know about, is in **NP**.
- The language of, say, palindrome words, is easy to be proved in **P**, thus in **NP**.
- Intuitively, however, the inherent difficulties of solving the two problems *should be* different.
- What can we thus conclude from the fact that a language $\mathcal{L}$ is **in** the class **NP**?
  - Not much, actually! We can only conclude that it is not *too* complicated to solve it.

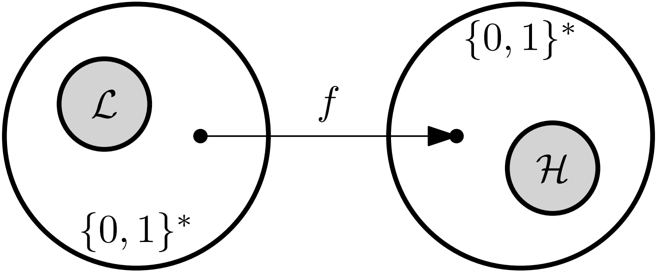
Are all Problems in **NP** Equivalent?

- The Maximum Independent Set, a problem we already know about, is in **NP**.
- The language of, say, palindrome words, is easy to be proved in **P**, thus in **NP**.
- Intuitively, however, the inherent difficulties of solving the two problems *should be* different.
- What can we thus conclude from the fact that a language $\mathcal{L}$ is **in** the class **NP**?
  - Not much, actually! We can only conclude that it is not *too* complicated to solve it.
- We need something else, namely a (pre-order) relation between languages such that two languages being in relation tells us something precise about the **relative** difficulty of deciding them.

Reductions



Reductions



- The language $\mathcal{L}$ is said to be **polynomial-time reducible** to another language $\mathcal{H}$ iff there is a polytime computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that $x \in \mathcal{L}$ iff $f(x) \in \mathcal{H}$.
- In this case, we write $\mathcal{L} \leq_p \mathcal{H}$.

Reductions and Complexity

- If $\mathcal{L} \leq_p \mathcal{H}$, then $\mathcal{H}$ is at least as difficult as $\mathcal{L}$, at least as far as classes like $\mathbf{P}$ (or above it) are concerned.
  - If, e.g., $\mathcal{L} \leq_p \mathcal{H}$ and $\mathcal{H} \in \mathbf{P}$, then also $\mathcal{L} \in \mathbf{P}$: a way to decide if $x \in \mathcal{L}$ consists in traslating it into $f(x)$ (which can be done in polynomial time), then checking whether $f(x) \in \mathcal{H}$.

Reductions and Complexity

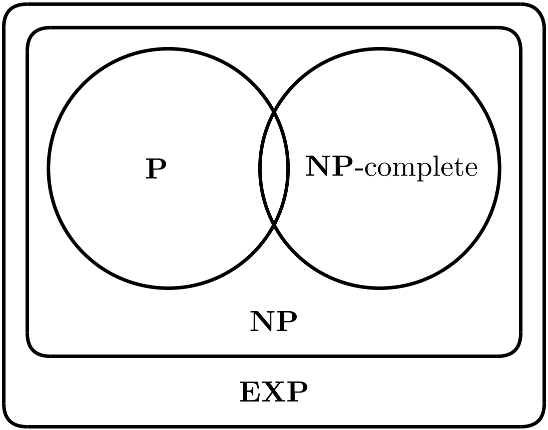
- If $\mathcal{L} \leq_p \mathcal{H}$, then $\mathcal{H}$ is at least as difficult as $\mathcal{L}$, at least as far as classes like $\mathbf{P}$ (or above it) are concerned.
  - If, e.g., $\mathcal{L} \leq_p \mathcal{H}$ and $\mathcal{H} \in \mathbf{P}$, then also $\mathcal{L} \in \mathbf{P}$: a way to decide if $x \in \mathcal{L}$ consists in traslating it into $f(x)$ (which can be done in polynomial time), then checking whether $f(x) \in \mathcal{H}$.
- A language $\mathcal{H} \subseteq \{0,1\}^*$ is said to be:
  - $\mathbf{NP}$-hard if $\mathcal{L} \leq_p \mathcal{H}$ for every $\mathcal{L} \in \mathbf{NP}$.
  - $\mathbf{NP}$-complete if $\mathcal{H}$ is $\mathbf{NP}$-hard, and $\mathcal{H} \in \mathbf{NP}$.

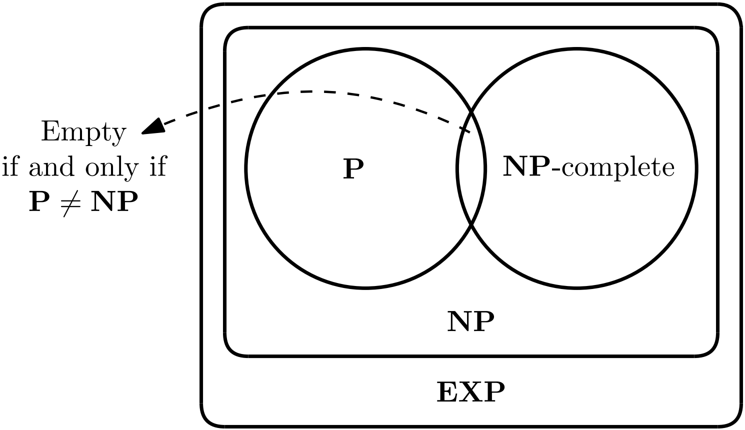
Reductions and Complexity

- If $\mathcal{L} \leq_p \mathcal{H}$, then $\mathcal{H}$ is at least as difficult as $\mathcal{L}$, at least as far as classes like $\mathbf{P}$ (or above it) are concerned.
  - If, e.g., $\mathcal{L} \leq_p \mathcal{H}$ and $\mathcal{H} \in \mathbf{P}$, then also $\mathcal{L} \in \mathbf{P}$: a way to decide if $x \in \mathcal{L}$ consists in traslating it into $f(x)$ (which can be done in polynomial time), then checking whether $f(x) \in \mathcal{H}$.
- A language $\mathcal{H} \subseteq \{0,1\}^*$ is said to be:
  - $\mathbf{NP}$-hard if $\mathcal{L} \leq_p \mathcal{H}$ for every $\mathcal{L} \in \mathbf{NP}$.
  - $\mathbf{NP}$-complete if $\mathcal{H}$ is $\mathbf{NP}$-hard, and $\mathcal{H} \in \mathbf{NP}$.

Theorem

1. *The relation $\leq_p$ is a pre-order (i.e. it is reflexive and transitive).*
2. *If $\mathcal{L}$ is $\mathbf{NP}$-hard and $\mathcal{L} \in \mathbf{P}$, then $\mathbf{P} = \mathbf{NP}$.*
3. *If $\mathcal{L}$ is $\mathbf{NP}$-complete, then $\mathcal{L} \in \mathbf{P}$ iff $\mathbf{P} = \mathbf{NP}$.*


P
NP-complete
NP
EXP


Empty
if and only if
$\mathbf{P} \neq \mathbf{NP}$
P
NP-complete
NP
EXP

The First Example of an **NP**-complete Problem

- One obvious way of building an **NP**-complete problem is to define it as the problem of *simulating* any Turing machine.

The First Example of an NP-complete Problem

- One obvious way of building an **NP**-complete problem is to define it as the problem of *simulating* any Turing machine.
- Let TMSAT be the following language:

$$\texttt{TMSAT} = \{(\alpha, x, 1^n, 1^t) \mid \exists u \in \{0,1\}^n . \mathcal{M}_\alpha \text{ outputs } 1 \\ \text{on input } (x, u) \text{ within } t \text{ steps}\}$$

The First Example of an **NP**-complete Problem

- One obvious way of building an **NP**-complete problem is to define it as the problem of *simulating* any Turing machine.
- Let `TMSAT` be the following language:

$$\texttt{TMSAT} = \{(\alpha, x, 1^n, 1^t) \mid \exists u \in \{0,1\}^n . \mathcal{M}_\alpha \text{ outputs } 1 \text{ on input } (x, u) \text{ within } t \text{ steps}\}$$

Theorem

`TMSAT` *is* **NP**-*complete.*

The First Example of an **NP**-complete Problem

- One obvious way of building an **NP**-complete problem is to define it as the problem of *simulating* any Turing machine.
- Let `TMSAT` be the following language:

$$\texttt{TMSAT} = \{(\alpha, x, 1^n, 1^t) \mid \exists u \in \{0,1\}^n . \mathcal{M}_\alpha \text{ outputs } 1 \text{ on input } (x, u) \text{ within } t \text{ steps}\}$$

Theorem

`TMSAT` *is* **NP**-*complete.*

- Although interesting from a purely theoretical perspective, the language `TMSAT` is very specifically tied to Turing Machines, and thus of no practical importance.

A Quick Recap on Propositional Logic

- Formulas of **propositional logic** are either:
  - Propositional variables, like $X, Y, Z, \ldots;$
  - Built from smaller formulas by way of the connective $\wedge$, $\vee$ and $\neg$.

  Formulas are indicated as $F, G, H, \ldots,$

A Quick Recap on Propositional Logic

- Formulas of **propositional logic** are either:
  - Propositional variables, like $X, Y, Z, \ldots;$
  - Built from smaller formulas by way of the connective $\wedge$, $\vee$ and $\neg$.

  Formulas are indicated as $F, G, H, \ldots,$
- Examples: $X \vee \neg X$, $X \wedge (Y \vee \neg Z)$, etc.

A Quick Recap on Propositional Logic

- Formulas of **propositional logic** are either:
  - Propositional variables, like $X, Y, Z, \ldots;$
  - Built from smaller formulas by way of the connective $\wedge$, $\vee$ and $\neg$.

  Formulas are indicated as $F, G, H, \ldots,$
- Examples: $X \vee \neg X$, $X \wedge (Y \vee \neg Z)$, etc.
- Given a formula F and an assignment ρ of elements from $\{0,1\}$ to the propositional variables in F, one can define the **truth value** for F, indicated as $[\![F]\!]_\rho$, by induction on F:

$$[\![X]\!]_\rho = \rho(X) \qquad [\![F \vee G]\!]_\rho = [\![F]\!]_\rho + [\![G]\!]_\rho$$

$$[\![F \wedge G]\!]_\rho = [\![F]\!]_\rho \cdot [\![G]\!]_\rho \qquad [\![\neg F]\!]_\rho = 1 - [\![F]\!]_\rho$$

A Quick Recap on Propositional Logic

- Formulas of **propositional logic** are either:
  - Propositional variables, like $X, Y, Z, \ldots$;
  - Built from smaller formulas by way of the connective $\wedge$, $\vee$ and $\neg$.

  Formulas are indicated as $F, G, H, \ldots$,
- Examples: $X \vee \neg X$, $X \wedge (Y \vee \neg Z)$, etc.
- Given a formula F and an assignment ρ of elements from $\{0, 1\}$ to the propositional variables in F, one can define the **truth value** for F, indicated as $[\![F]\!]_\rho$, by induction on F:

$$
\begin{aligned}
[\![X]\!]_\rho &= \rho(X) & \qquad [\![F \vee G]\!]_\rho &= [\![F]\!]_\rho + [\![G]\!]_\rho \\
[\![F \wedge G]\!]_\rho &= [\![F]\!]_\rho \cdot [\![G]\!]_\rho & \qquad [\![\neg F]\!]_\rho &= 1 - [\![F]\!]_\rho
\end{aligned}
$$

- Examples: $[\![X \vee \neg X]\!]_\rho = 1$ for every ρ, while the truth value $[\![X \wedge (Y \vee \neg Z)]\!]_\rho$ equals 1 only for some of the possible ρ.

A Quick Recap on Propositional Logic

- Formulas of **propositional logic** are either:
  - Propositional variables, like $X, Y, Z, \dots;$
  - Built from smaller formulas by way of the connective $\wedge$, $\vee$ and $\neg$.

  Formulas are indicated as $F, G, H, \dots,$
- Examples: $X \vee \neg X$, $X \wedge (Y \vee \neg Z)$, etc.
- Given a formula F and an assignment ρ of elements from $\{0,1\}$ to the propositional variables in F, one can define the **truth value** for F, indicated as $[\![F]\!]_\rho$, by induction on F:

$$[\![X]\!]_\rho = \rho(X) \qquad [\![F \vee G]\!]_\rho = [\![F]\!]_\rho + [\![G]\!]_\rho$$
$$[\![F \wedge G]\!]_\rho = [\![F]\!]_\rho \cdot [\![G]\!]_\rho \qquad [\![\neg F]\!]_\rho = 1 - [\![F]\!]_\rho$$

- Examples: $[\![X \vee \neg X]\!]_\rho = 1$ for every ρ, while the truth value $[\![X \wedge (Y \vee \neg Z)]\!]_\rho$ equals 1 only for some of the possible ρ.
- A formula F is **satisfiable** iff there is one ρ such that $[\![F]\!]_\rho = 1$.

The Cook-Levin Theorem

- A propositional formula F is said to be in **conjunctive normal form** (or a **CNF**) when it is a conjuction of disjunctions of *literals* (a literal being a variable or its negation).
- Examples: $X \vee \neg X$ and $X \wedge (Y \vee \neg Z)$ are both CNFs, while a formula which is *not* a CNF is $X \vee (Y \wedge \neg Z)$.

The Cook-Levin Theorem

- A propositional formula F is said to be in **conjunctive normal form** (or a **CNF**) when it is a conjuction of disjunctions of *literals* (a literal being a variable or its negation).
- Examples: $X \vee \neg X$ and $X \wedge (Y \vee \neg Z)$ are both CNFs, while a formula which is *not* a CNF is $X \vee (Y \wedge \neg Z)$.
- The disjunctions in a CNF are said to be **clauses**, and a k**CNF** is a CNF whose clauses contains at most $k \in \mathbb{N}$ literals. Examples: the two formulas $X \vee \neg X$ and $X \wedge (Y \vee \neg Z)$ are 2CNFs, but not 1CNFs.

The Cook-Levin Theorem

- A propositional formula F is said to be in **conjunctive normal form** (or a **CNF**) when it is a conjuction of disjunctions of *literals* (a literal being a variable or its negation).
- Examples: $X \vee \neg X$ and $X \wedge (Y \vee \neg Z)$ are both CNFs, while a formula which is *not* a CNF is $X \vee (Y \wedge \neg Z)$.
- The disjunctions in a CNF are said to be **clauses**, and a k**CNF** is a CNF whose clauses contains at most $k \in \mathbb{N}$ literals. Examples: the two formulas $X \vee \neg X$ and $X \wedge (Y \vee \neg Z)$ are 2CNFs, but not 1CNFs.

Theorem (Cook-Levin)

The following two languages are **NP***-complete:*

$$\mathtt{SAT} = \{\llcorner F \lrcorner \mid F \textit{ is a satisfiable CNF}\}$$

$$\mathtt{3SAT} = \{\llcorner F \lrcorner \mid F \textit{ is a satisfiable 3CNF}\}$$

The Cook-Levin Theorem

- The proof of the Cook-Levin Theorem is outside the scope of this course.
  - It would require at least one lecture, alone!.

The Cook-Levin Theorem

- The proof of the Cook-Levin Theorem is outside the scope of this course.
  - It would require at least one lecture, alone!.
- The **structure** of the proof is as follows:
  - We consider any language $\mathcal{L} \in \mathbf{NP}$, and we show that $\mathcal{L} \leq_p \texttt{SAT}$.

The Cook-Levin Theorem

- The proof of the Cook-Levin Theorem is outside the scope of this course.
  - It would require at least one lecture, alone!.
- The **structure** of the proof is as follows:
  - We consider any language $\mathcal{L} \in \mathbf{NP}$, and we show that $\mathcal{L} \leq_p \texttt{SAT}$.
  - To do that, we consider any possible polynomial p and any polytime deterministic TM $\mathcal{M}$.
    - Intuitively, these exist because $\mathcal{L} \in \mathbf{NP}$

The Cook-Levin Theorem

- The proof of the Cook-Levin Theorem is outside the scope of this course.
  - It would require at least one lecture, alone!.
- The **structure** of the proof is as follows:
  - We consider any language $\mathcal{L} \in \mathbf{NP}$, and we show that $\mathcal{L} \leq_p \mathtt{SAT}$.
  - To do that, we consider any possible polynomial p and any polytime deterministic TM $\mathcal{M}$.
    - Intuitively, these exist because $\mathcal{L} \in \mathbf{NP}$
  - We define a polynomial-time transformation $x \mapsto \varphi_x$ from strings to CNFs such that

$$\varphi_x \in \mathtt{SAT} \Leftrightarrow \exists y \in \{0,1\}^{p(|x|)}.\mathcal{M}(x,y) = 1$$

    - This is the crucial step, and requires quite some work. It amounts to showing that computation in TM is *inherently local*.

The Cook-Levin Theorem

- The proof of the Cook-Levin Theorem is outside the scope of this course.
  - It would require at least one lecture, alone!.
- The **structure** of the proof is as follows:
  - We consider any language $\mathcal{L} \in \mathbf{NP}$, and we show that $\mathcal{L} \leq_p \texttt{SAT}$.
  - To do that, we consider any possible polynomial p and any polytime deterministic TM $\mathcal{M}$.
    - Intuitively, these exist because $\mathcal{L} \in \mathbf{NP}$
  - We define a polynomial-time transformation $x \mapsto \varphi_x$ from strings to CNFs such that

$$\varphi_x \in \texttt{SAT} \Leftrightarrow \exists y \in \{0,1\}^{p(|x|)}.\mathcal{M}(x,y) = 1$$

    - This is the crucial step, and requires quite some work. It amounts to showing that computation in TM is *inherently local*.
  - Finally, we need to show that $\texttt{SAT} \leq_p \texttt{3SAT}$.

Proving a Problem Hard

- Suppose you are studying the complexity of a language $\mathcal{L}$, and you are convinced that the underlying problem is **hard**. How should you back your claim?
- By proving that $\mathcal{L} \in \mathbf{P}$?
  - **Of course not**, this way you rather prove that $\mathcal{L}$ is easy.

Proving a Problem Hard

- Suppose you are studying the complexity of a language $\mathcal{L}$, and you are convinced that the underlying problem is **hard**. How should you back your claim?
- By proving that $\mathcal{L} \in \mathbf{P}$?
  - **Of course not**, this way you rather prove that $\mathcal{L}$ is easy.
- By proving that $\mathcal{L} \in \mathbf{EXP}$?
  - **No**, the fact that there is an exponential-time algorithm deciding $\mathcal{L}$ does **not** mean that no polynomial-time algorithm for $\mathcal{L}$ exist.

Proving a Problem Hard

- Suppose you are studying the complexity of a language $\mathcal{L}$, and you are convinced that the underlying problem is **hard**. How should you back your claim?
- By proving that $\mathcal{L} \in \mathbf{P}$?
  - **Of course not**, this way you rather prove that $\mathcal{L}$ is easy.
- By proving that $\mathcal{L} \in \mathbf{EXP}$?
  - **No**, the fact that there is an exponential-time algorithm deciding $\mathcal{L}$ does **not** mean that no polynomial-time algorithm for $\mathcal{L}$ exist.
- By proving that $\mathcal{L} \in \mathbf{NP}$?
  - Again, this **does not** mean much.

Proving a Problem Hard

- Suppose you are studying the complexity of a language $\mathcal{L}$, and you are convinced that the underlying problem is **hard**. How should you back your claim?
- By proving that $\mathcal{L} \in \mathbf{P}$?
  - **Of course not**, this way you rather prove that $\mathcal{L}$ is easy.
- By proving that $\mathcal{L} \in \mathbf{EXP}$?
  - **No**, the fact that there is an exponential-time algorithm deciding $\mathcal{L}$ does **not** mean that no polynomial-time algorithm for $\mathcal{L}$ exist.
- By proving that $\mathcal{L} \in \mathbf{NP}$?
  - Again, this **does not** mean much.
- By proving that $\mathcal{L}$ is $\mathbf{NP}$-complete?
  - **Yes**, this way you prove that the problem is not so hard (being in $\mathbf{NP}$), but not so easy either (unless $\mathbf{P} = \mathbf{NP}$).

Proving a Problem **NP**-complete

- If we want to prove $\mathcal{L}$ to be **NP**-complete, we have to prove two statements:

Proving a Problem **NP**-complete

- If we want to prove $\mathcal{L}$ to be **NP**-complete, we have to prove two statements:
  1. That $\mathcal{L}$ is in **NP**.
     - This amounts to showing that there are p polynomial and $\mathcal{M}$ polytime TM such that $\mathcal{L}$ can be written as

       $$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.\mathcal{M}(x,y) = 1\}$$

     - This is typically rather easy.

Proving a Problem **NP**-complete

- If we want to prove $\mathcal{L}$ to be **NP**-complete, we have to prove two statements:
  1. That $\mathcal{L}$ is in **NP**.
     - This amounts to showing that there are p polynomial and $\mathcal{M}$ polytime TM such that $\mathcal{L}$ can be written as

       $$\mathcal{L} = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{p(|x|)}.\mathcal{M}(x,y) = 1\}$$

     - This is typically rather easy.
  2. That any other language $\mathcal{H} \in \mathbf{NP}$ is such that $\mathcal{H} \leq_p \mathcal{L}$.
     - We can of course prove the statement directly.
     - More often (e.g. when showing `3SAT` **NP**-complete), one rather proves that $\mathcal{J} \leq_p \mathcal{L}$ for a language $\mathcal{J}$ which is already known to be **NP**-complete.
     - This is correct, simply because $\leq_p$ is transitive:

$$\begin{array}{c}
\vdots \\
\mathcal{H} \xrightarrow{\leq_p} \mathcal{J} \xrightarrow{\leq_p} \mathcal{L} \\
\vdots
\end{array}$$

NP-complete Problems: Examples

Theorem

The Maximum Independent Set Problem `INDSET` *is* **NP***-complete.*

NP-complete Problems: Examples

Theorem

The Maximum Independent Set Problem `INDSET` *is* **NP**-*complete.*

- There is a polytime reduction from `SAT` to `INDSET`.

NP-complete Problems: Examples

Theorem

The Maximum Independent Set Problem `INDSET` *is* **NP***-complete.*

- There is a polytime reduction from `SAT` to `INDSET`.

Theorem

The Subset Sum Problem `SUBSETSUM` *is* **NP***-complete.*

- There are polytime reductions from `3SAT` to a variation `OL3SAT` of it, and from the latter to `SUBSETSUM`.

NP-complete Problems: Examples

Theorem

The Maximum Independent Set Problem `INDSET` *is* **NP**-*complete.*

- There is a polytime reduction from `SAT` to `INDSET`.

Theorem

The Subset Sum Problem `SUBSETSUM` *is* **NP**-*complete.*

- There are polytime reductions from `3SAT` to a variation `OL3SAT` of it, and from the latter to `SUBSETSUM`.

Theorem

The Decisional 0/1 Linear Programming Problem `ILP` *is* **NP**-*complete*

- There is an easy polytime reduction from `SAT` to `ILP`.

The Zoo of **NP**-complete Problems

- For any pair $\mathcal{L}, \mathcal{H}$ of **NP**-complete problems, we have that

$$\mathcal{L} \leq_p \mathcal{H} \qquad \mathcal{H} \leq_p \mathcal{L}$$

  - In other words, $\mathcal{L}$ and $\mathcal{H}$ are equivalent.

The Zoo of **NP**-complete Problems

- For any pair $\mathcal{L}, \mathcal{H}$ of **NP**-complete problems, we have that

$$\mathcal{L} \leq_p \mathcal{H} \qquad \mathcal{H} \leq_p \mathcal{L}$$

  - In other words, $\mathcal{L}$ and $\mathcal{H}$ are equivalent.
- This being said, defining a reduction from $\mathcal{L}$ to $\mathcal{H}$ is sometimes easy, and sometimes very difficult, and it is instructive to think at **NP**-complete problems as forming a graph, where edges are *natural* polytime reductions
  - A fragment, the one we have encountered so far is the following

$$\begin{tikzcd}
 & \texttt{ILP} & & \\
\texttt{SAT} \arrow[ur] \arrow[r] \arrow[dr] & \texttt{INDSET} & & \\
 & \texttt{3SAT} \arrow[r] & \texttt{OL3SAT} \arrow[r] & \texttt{SUBSETSUM}
\end{tikzcd}$$

The Zoo of **NP**-complete Problems

- For any pair $\mathcal{L}, \mathcal{H}$ of **NP**-complete problems, we have that

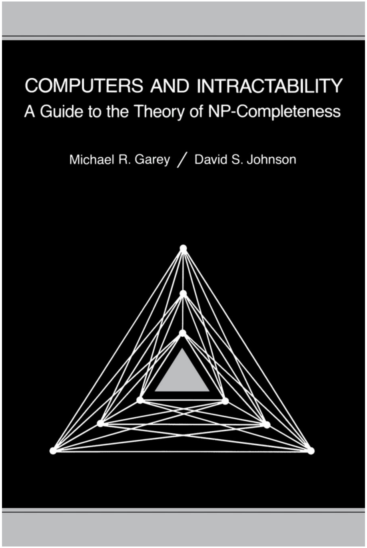
$$\mathcal{L} \leq_p \mathcal{H} \qquad \mathcal{H} \leq_p \mathcal{L}$$

  - In other words, $\mathcal{L}$ and $\mathcal{H}$ are equivalent.
- This being said, defining a reduction from $\mathcal{L}$ to $\mathcal{H}$ is sometimes easy, and sometimes very difficult, and it is instructive to think at **NP**-complete problems as forming a graph, where edges are *natural* polytime reductions
  - A fragment, the one we have encountered so far is the following

$$\begin{tikzcd}
 & \mathtt{ILP} & & \\
\mathtt{SAT} \arrow[ur] \arrow[r] \arrow[dr] & \mathtt{INDSET} & & \\
 & \mathtt{3SAT} \arrow[r] & \mathtt{OL3SAT} \arrow[r] & \mathtt{SUBSETSUM}
\end{tikzcd}$$

  - In fact, this graph is **huge**: thousands of different problems are known to be **NP**-complete

The Zoo of **NP**-complete Problems



Mitigating the Computational Difficulty of **NP**-complete Problems

- What if we actually prove a problem $\mathcal{L}$ we are interesting in solving to actually *be* **NP**-complete?
  - Is the hope to solve it for inputs of nontrivial length lost?

Mitigating the Computational Difficulty of **NP**-complete Problems

- What if we actually prove a problem $\mathcal{L}$ we are interesting in solving to actually *be* **NP**-complete?
  - Is the hope to solve it for inputs of nontrivial length lost?
- We know that this implies that, given the state-of-the-art in computational complexity, no polynomial time algorithm for $\mathcal{L}$ is known.

Mitigating the Computational Difficulty of **NP**-complete Problems

- What if we actually prove a problem $\mathcal{L}$ we are interesting in solving to actually *be* **NP**-complete?
  - Is the hope to solve it for inputs of nontrivial length lost?
- We know that this implies that, given the state-of-the-art in computational complexity, no polynomial time algorithm for $\mathcal{L}$ is known.
- On the other hand, given that $\mathcal{L}$ is in **NP**, and that `SAT` is **NP**-complete, we know that $\mathcal{L} \leq_p \texttt{SAT}$, i.e. that instances of $\mathcal{L}$ can be efficiently translated into instances of `SAT`.

Mitigating the Computational Difficulty of **NP**-complete Problems

- What if we actually prove a problem $\mathcal{L}$ we are interesting in solving to actually *be* **NP**-complete?
  - Is the hope to solve it for inputs of nontrivial length lost?
- We know that this implies that, given the state-of-the-art in computational complexity, no polynomial time algorithm for $\mathcal{L}$ is known.
- On the other hand, given that $\mathcal{L}$ is in **NP**, and that `SAT` is **NP**-complete, we know that $\mathcal{L} \leq_p$ `SAT`, i.e. that instances of $\mathcal{L}$ can be efficiently translated into instances of `SAT`.
- This is often **very useful**, because specialised tools for `SAT`, called SAT-solvers do exist.
  - They do not work in polynomial time.
  - Concretely, they work extremely well on a relatively large class of formulas.

Is This the End of The Story?

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$$

Is This the End of The Story?

$$\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \begin{array}{c}\mathbf{NP} \\ \mathbf{coNP}\end{array} \subseteq \cdots \subseteq \mathbf{PH} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP} \subseteq \cdots$$

Thank You!

Questions?