THE CLASS NP

Many important algorithmic problems have the following property:

- there is no known polynomial time algorithm to find a solution to the problem;
- if a solution to the problem is given, it is easy to *verify* (in polynomial time) that it is indeed a solution.

We can formalize the idea as follows (see section 7.3):
Definition. An algorithm (or a Turing machine) V verifies a language A if

$$A = \{w \mid V \text{ accepts } < w, s > \text{ for some string } s\}$$

Above V is a polynomial time verifier if it runs in time polynomial in the length of w.

Note: Above the string s is called the certificate for membership in A. The length of a certificate can be arbitrary and is not included in the running time. However, a polynomial time verifier can use only certificates of polynomial length. (Why?)

- NP is defined to consist of the languages that have *polynomial verifiers.*

Alternatively, NP could be characterized by polynomial time nondeterministic Turing machines (which is sometimes used as the definition).

Theorem. (Th. 7.20 in the text) NP consists exactly of languages recognized by nondeterministic polynomial time Turing machines.

- The question P $=^?$ NP remains still open. Also it is not known whether or not NP is closed under complementation.

The so called NP-complete problems occupy an important place in our study of complexity: if it were to be proven that P $\neq$ NP, then none of the NP-complete problems are in P.

For estimating/comparing the difficulty of problems in NP we can use a time-bounded variant of the mapping reductions (considered earlier in Chapter 5).

A function $f : \Sigma^* \longrightarrow \Gamma^*$ is *computable in polynomial time* if there is a deterministic Turing machine M and a polynomial p such that when M is started with input w ($\in \Sigma^*$) it halts after at most $p(|w|)$ steps and the tape contents is $f(w)$.

Definition. (Def. 7.29, section 7.4) A language A is *polynomial time reducible* to language B, $A \leq_P B$, if there is a polynomial time computable function f with the property that

$$(\forall w \in \Sigma^*) \quad w \in A \quad \Leftrightarrow \quad f(w) \in B.$$

Polynomial time reductions have the following properties:

1. $A \leq_P A$ for all languages A.
2. If $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$ (here A, B, C are languages).
3. If $A \leq_P B$ and $B \in$ P, then $A \in$ P (P is closed under $\leq_P$).

Definition. (Def. 7.34) A language B is NP-complete if

1. $B \in$ NP, and
2. every language in NP is polynomial time reducible to B.

Theorem. Let B be an NP-complete language.

1. If $B \in$ P then P = NP.

2. If $B \leq_P C$ and $C \in$ NP, then also C is NP-complete.

The "original" NP-complete problem is satisfiability

$$\text{SAT} = \{< \phi > | \ \phi \text{ is a satisfiable Boolean formula } \}.$$

Cook–Levin theorem: SAT is NP-complete.

The proof of the Cook–Levin theorem requires an involved construction. In the following example we consider a language that has a simple NP-completeness proof. The below problem (language) has an "artificial" definition and is not very useful for establishing NP-completeness of other problems. However, it provides a simple way to prove that NP-complete problems do exist.

Example. Here we consider a "simple NP-complete problem", that is, a problem for which we can establish the NP-completeness straightforwardly (relying only on the definition of NP-completeness).

Define

$$B_{\text{simple}} = \{< M, w > \cdot 1^t \ \mid \ M \text{ is an NTM that accepts } w \text{ in at most } t \text{ steps } \}.$$

That is, the strings of the language B_{simple} consist of an encoding of a nondeterministic TM M and input string w followed by t copies of symbol "1" where the NTM M accepts w in a computation with at most t steps.

The language B_{simple} is defined directly in terms of computations of an arbitrary Turing machine, where the time bound is given as part of the input and in unary notation. For this reason, the language B_{simple} is not very useful for establishing the NP-hardness of natural combinatorial problems – if we want to reduce B_{simple} to a combinatorial problem C_{comb} (involving graphs, scheduling etc.) this could be roughly as hard as showing that an arbitry

problem defined by a polynomial time bounded NTM can be reduced to C_{comb}. The "advantage" of the language definition B_{simple} is that it allows a simple proof of NP-completeness, based only on the definition of NP-completeness.

- In class we will show that B_{simple} is NP-complete.

There exist "hard" problems inside NP that are not known to be NP-complete.

Example. Graphs G and H are called isomorphic if the nodes of G can be "reordered" so that it is identical to H.

The graph isomorphism problem is encoded as the language

$$\text{ISO} = \{< G, H > |\ G \text{ and } H \text{ are isomorphic graphs } \}$$

It is easy to see that ISO is in NP. (How?) There is no known polynomial time algorithm for ISO but, on the other hand, ISO is not known to be NP-complete. It would be very hard to prove that ISO is <u>not</u> NP-complete. (Why?)

To conclude we note the following:

- It is not known whether NP is closed under complementation. The language family consisting of complements of languages in NP is denoted coNP.
- If we can show that NP $\neq$ coNP, what are the consequences?
- Is it possible that NP = coNP but NP $\neq$ P?
- Also it is not known whether or not graph isomorphism is in coNP.
- Assume it is shown that P = NP. Under this assumption, what can we say about the class of NP-complete problems?

  On the other hand, if we were to show that ISO is *not* NP-complete, what are the consequences?