Chapter 10

Intractable Problems

(2015/12/25)

Lion Monument in Lucerne, Switzerland 1998

Outline

10.0 Introduction

- **Concepts to be taught ---**
  - ♦ We will study the theory of "intractability." That is, we will study the techniques for showing problems not solvable in polynomial time.

  - ♦ Definition of *intractable* problems – problems which can only be solved in exponential time.

  - ♦ Review of two concepts ---
    - The problems solvable on computers are exactly those solvable on Turing machines.
    - Problems requiring polynomial time are solvable in amounts of time which we can tolerate, while those requiring exponential time generally cannot be solved in reasonable time except for small instances.

  - ♦ We will study a "(boolean) satisfiability" problem equivalent to L_u and PCP.
  - ♦ We also reduce tractable or intractable problems but the *reduction should be done in polynomial time*. That is, we need polynomial-time reductions.

  - ♦ Let $\mathcal{P}$ denote the class of problems which are solvable by deterministic TMs (DTMs) in polynomial time.
  - ♦ Let $\mathcal{NP}$ denote the class of problems which are solvable by nondeterministic TMs (NTMs) in polynomial time.

  - ♦ A major assumption in the theory of intractability is $\mathcal{P} \neq \mathcal{NP}$ *(still an open problem)*.
  - ♦ $\mathcal{P} \neq \mathcal{NP}$ means: $\mathcal{NP}$ includes at least some problems which are not in $\mathcal{P}$ (even if we allow a higher-degree polynomial time for the DTM).

  - ♦ There are thousands of problems in $\mathcal{NP}$ which are easily solved by a polynomial-time NTM but no polynomial-time DTM is known for their solution.
  - ♦ *Either all* of these problems in $\mathcal{NP}$ have polynomial-time deterministic solutions *or* none does (i.e., they require exponential time).

10.1 The Classes $\mathcal{P}$ and $\mathcal{NP}$

- **Concepts to be taught ---**
  - ♦ $\mathcal{P}$
  - ♦ $\mathcal{NP}$
  - ♦ Technique of polynomial-time reduction
  - ♦ NP-completeness

10.1.1 Problems Solvable in Polynomial Time

- **Definitions ---**
  - ♦ A TM M is said to be of time complexity $T(n)$ [or to have "running time $T(n)$"] if

whenever M is given an input w of length n, *M halts* after making at most $T(n)$ moves, regardless of whether or not M accepts.

- ♦ A language L is in class $\mathcal{P}$ if there is some polynomial $T(n)$ such that $L = L(M)$ for some DTM M of time complexity $T(n)$.

- **Questions ---**
  - ♦ (in-box discussion, p. 427) Is there anything between polynomial time $O(n^k)$ and exponential time $O(2^{cn})$ for some constant c ?

    Answer: Yes! It is $O(n^{\log_2 n}) = O(2^{(\log_2 n)2})$. Why?

    - $\log_2 n > k$ for large n
    - $cn > (\log_2 n)^2$ for large n

10.1.2 <u>An Example: Kruskal's Algorithm</u>

- **Definitions –**
  - ♦ *Graphs* --- nodes + edges + weights
  - ♦ *Spanning tree* --- a subset of edges such that all nodes are connected
  - ♦ *Minimum-weight spanning tree* (MWST) --- a spanning tree with the least possible total edge weight

- Kruskal provides a "greedy' algorithm for finding an MWST.
- Kruskal's algorithm may be solved in polynomial time by a computer:
  - ♦ in $O(n^2)$ easily;
  - ♦ in $O(n\log n)$ more efficiently.

- **The modified MWST problem ---**

  "does graph G has an MWST of total weight W or less?"

  - ♦ This problem may solved in polynomial time $O(n^4)$ by a DTM (see pp. 430-431 in the textbook).

- **Conclusion ---**

  The MWST problem is in $\mathcal{P}$.

10.1.3 <u>Nondeterministic Polynomial Time</u>

- **Definition ---**

  A language L is in class $\mathcal{NP}$ if there is some polynomial $T(n)$ such that $L = L(M)$ for some NTM M of time complexity $T(n)$, where n is the length of an input.

  (Note: NP means nondeterministic polynomial)

- Because DTM's are also NTM's, so $\mathcal{P} \subseteq \mathcal{NP}$.
- It seems *some* problems in $\mathcal{NP}$ is not in $\mathcal{P}$, but actually "whether $\mathcal{P} = \mathcal{NP}$?" is an open problem.
- That is*, whether everything that can be done in polynomial time by an NTM can in fact be done by a DTM in polynomial time, perhaps with a higher-degree polynomial,* is unknown yet.

10.1.4 An $\mathcal{NP}$ Example: The Traveling Salesman Problem

■ **Definition of *traveling salesman problem* (TSP) ---**

Given a graph with integer weights on edges and a weight limit, if there is a Hamilton circuit of total weight at most W in the graph?

♦ Hamilton circuit --- a set of edges that connect the nodes into a single cycle ("*completing the traversal in one way to save time and gas*" "一趟走完, 省時省油").

■ **Properties of the TSP ---**

♦ It appears that all ways to solve the TSP have to try all cycles and computing their total weights.

♦ The number of cycles in a graph with m nodes is $O(m!)$ which is more than the exponential time $O(2^{cm})$ for any constant c.

♦ If we have a nondeterministic computer or NTM, we can guess all permutations of nodes and compute their weights in order in polynomial time $O(n)$ and $O(n^4)$, respectively, using a single-tape TM. (note: n here = m in the last page)

♦ So, the TSP is in $\mathcal{NP}$.

10.1.5 Polynomial-Time Reductions

■ **Concepts ---**

♦ To prove a problem P_2 not in $\mathcal{P}$,

we can reduce a problem P_1 also not in $\mathcal{P}$ to it. (A)

♦ An illustrative diagram is Fig. 10.1 (Fig. 10.2 in the textbook) below (similar to Fig. 8.7).

♦ The reduction algorithm should take polynomial time; otherwise, the proof will not be valid.

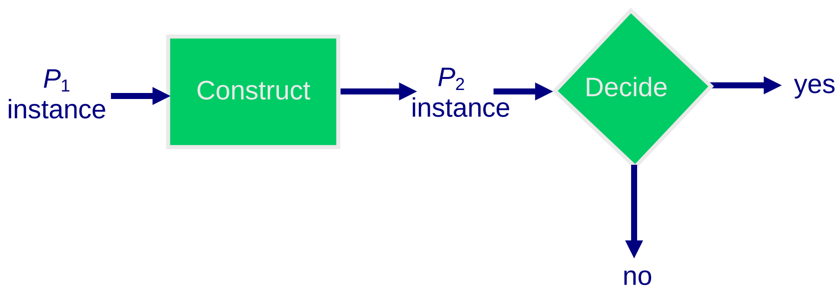

Figure 10.1 Reduction of problems.

■ **Proof of statement (A) above (by contradiction) ---**

♦ Assume P_2 is in $\mathcal{P}$.

♦ Given an input to P_1, the reduction includes translation of P_1 to P_2 and the output of P_2.

♦ Polynomial-time reduction means:

- the translation takes time $O(m^j)$ on input of length m;
- the output instance of P_2 cannot be longer than the number of steps $O(m^j)$, so that its length is at most $O(cm^j)$.

♦ Suppose that we can decide the membership in P_2 in time $O(n^k)$ for an input of length n.

- Then we can decide the membership of P_1 for an input of length m by conducting:
  - the reduction of translating P_1 to P_2 with output instance of P_2 of length $O(cm^j)$; and
  - performing the decision work about P_2.
- The total work takes time $O(m^j) + O((cm^j)^k) = O(m^j + cm^{jk})$, which is an order of polynomial time (since c, j, k are all constants). (See the illustration in Fig. 10.2).
- Therefore, decision of P_1 takes polynomial time. That is, P_1 is in $\mathcal{P}$.
- This is a contradiction because we have known that P_1 is not in $\mathcal{P}$.
- Therefore, the assumption "P_2 is $\mathcal{P}$" made initially is wrong. Done**.**

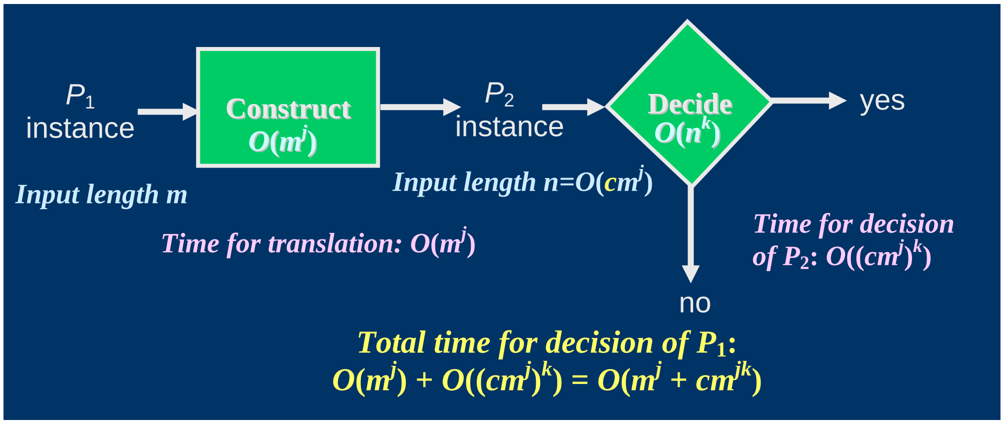

Fig. 10.2 Time complexity of problem reduction.

- **Concepts ---**
  - Reversely, we can also say that if P_2 is in $\mathcal{P}$, and P_1 can be reduced to P_2 in polynomial time, then P_1 is also in $\mathcal{P}$.
  - Summary: if $P_1 \rightarrow_{\text{reduce}} P_2$, then
    - P_1 not in $\mathcal{P} \Rightarrow P_2$ not in $\mathcal{P}$;
    - P_2 in $\mathcal{P} \Rightarrow P_1$ in $\mathcal{P}$.
  - Only polynomial-reductions will be used in the study of intractability.

10.1.6 NP-Complete Problems

- **Definition of NP-completeness ---**

  Let L be a language (problem). We say L is NP-complete if the following statements about L are true:

  - L is in $\mathcal{NP}$.
  - For *every* language L' in $\mathcal{NP}$, there is a polynomial-time reduction of L' to L *(every: "completeness")*.

- **Some comments on NP-completeness ---**
  - As will be seen, an NP-complete problem is the TSP.

- It appears that $\mathcal{P} \neq \mathcal{NP}$, and that all NP-complete problems are in $\mathcal{NP} - \mathcal{P}$, so we view a proof of NP-completeness of a problem as a proof of the fact that the problem is *not* in $\mathcal{P}$.
- We will show our first NP-complete problem to be the (boolean) satisfiability problem (SAT) by showing that the language of every polynomial-time NTM has a polynomial-time reduction to the SAT.
- Once we have an NP-complete problem, we can prove a new problem P to be NP-complete by reducing some known NP-complete problem to it (P), using a polynomial-time reduction.

- **Theorem 10.4 ---**

  If P_1 is NP-complete, P_2 is in $\mathcal{NP}$, and there is a polynomial-time reduction of P_1 to P_2, then P_2 is NP-complete.

Proof.

- By the 2nd point of the definition of NP-completeness, we have to show every language L in $\mathcal{NP}$ *polynomial-time reduces* to P_2.
- Since P_1 is NP-complete, we know that L may be reduced to P_1 in polynomial-time $p(n)$.
- Thus, a string w in L of length n is converted to a string x in P_1 of length at most $p(n)$.
- Also, we know P_1 may be reduced to P_2 in polynomial time, say, $q(m)$.
- This reduction transforms x to a string y in P_2, taking time at most $q(p(n))$.
- So, the transformation of w to y takes time at most $p(n) + q(p(n))$, which is a polynomial.
- Therefore, L is polynomial-time reducible to P_2. Done.
  (A diagram like the previous one may be drawn.)

- **Theorem 10.5 ---**

  If some NP-complete problem P is in $\mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.

(A wish to achieve so that the open problem can be solved!)

Proof.

- Since P is NP-complete, all languages L in $\mathcal{NP}$ reduce to P in polynomial time. And Since P is in $\mathcal{P}$, then L is in $\mathcal{P}$ (by Section 10.1.5, green line in p.27).
- That is, all languages L in $\mathcal{NP}$ are also in $\mathcal{P}$, i.e., $\mathcal{NP} \subset \mathcal{P}$.
- By definition, we have $\mathcal{P} \subset \mathcal{NP}$. So, $\mathcal{NP} = \mathcal{P}$. **Done.**

10.2 An NP-Complete Problems

- **NP-hard problem** (An in-box note of the last section) ---
  - Some problems are so hard that we can prove Condition (2) of the definition of NP-completeness ("every language in $\mathcal{NP}$ reduces to language L in polynomial time")

but we cannot prove Condition (1) ("*L* is in $\mathcal{NP}$.")

- "Intractable" is usually used to mean "NP-hard".

10.2.1 The Satisfiability Problem

- **Definition ---**

The boolean expressions are built from the following elements.

- Variables with values 1 (true) and 0 (false).
- Binary operators $\wedge$ and $\vee$ for logical AND and OR, respectively.
- Unary operator $\neg$ for logical NOT (negation).
- Parentheses (and) used to alter the default precedence of operators: $\neg$ (highest), $\wedge$, $\vee$ (lowest).

- **Example 10.6 ---**

An example of boolean expression is $E = x \wedge \neg (y \vee z)$.

- For *E* to be true, the only truth assignment *T* is: *x* is true, *y* is false, and *z* is false.

- **Definitions ---**
- A truth assignment *T* for a given boolean expression *E* assigns either true or false to each of the variables mentioned in *E*.
- The value assigned to a variable *x* is denoted by $T(x)$.
- The overall value of *E* is denoted by $E(T)$.
- A truth assignment *T* is said to *satisfy* boolean expression *E* if $E(T) = 1$.
- A boolean expression is said to be *satisfiable* if there exists at least one truth assignment *T* that satisfies *E*.

- **Example 10.7 ---**

The boolean expression *E* of the last example is satisfiable because the truth assignment *T* defined by $T(x) = 1$, $T(y) = 0$, and $T(z) = 0$ satisfies *E*.

- It can be figured out that the boolean expression $E' = x \wedge (\neg x \vee y) \wedge \neg y$ is not satisfiable (for details, see the textbook)

- **Definition ---**

The *satisfiability problem* is:

given a boolean expression, is it satisfiable?

which will be abbreviated as SAT.

- Stated as a *language*, the problem SAT is the set of (*coded*) boolean expressions that are satisfiable.

10.2.2 Representing SAT Instances

- **Concepts ---**

- ♦ We assume the variables are numbered as x_1, x_2, …
- ♦ To represent the boolean expression by codes,
  - the symbols $\wedge$, $\vee$, $\neg$, (, and) are represented by themselves;
  - the variable x_i is represented by x followed by 0's and 1's that represent i in binary.

■ **Example 10.8 ---**

The boolean expression of Example 10.6 $E = x \wedge \neg (y \vee z)$ may be coded as $x1 \wedge \neg (x10 \vee x11)$ after regarding x, y, and z as x_1, x_2, and x_3, respectively.

10.2.3 NP-completeness of the SAT Problem

■ **Concepts ---**

- ♦ The SAT problem is NP-complete.
- ♦ To prove this, we have to do the following:
  - show the SAT problem is in $\mathcal{NP}$; and
  - reduce every language in $\mathcal{NP}$ to the SAT problem.

■ **Theorem 10.9 (Cook's Theorem)** *(The greatest theorem in computational complexity)***---**

SAT is NP-complete.

Proof. (too long; only a sketch is shown here)

(part A --- proving that SAT is in $\mathcal{NP}$)

- ♦ use the nondeterministic ability of an NTM to guess a truth assignment T for the given expression E in polynomial time $O(n^4)$ (see the textbook for the details).

(part B --- proving if language L is in $\mathcal{NP}$, there is a polynomial-time reduction of L to SAT)

- ♦ describe the sequence of ID's of the NTM accepting L in terms of boolean variables;
- ♦ express acceptance of an input w by writing a boolean expression that is *satisfiable* if and only if *M accepts w* by a sequence of at most $p(n)$ moves where $n = |w|$ (see the textbook for the details).

10.3 A Restricted Satisfiable Problem

■ **Concepts to be taught ---**

- ♦ We want to prove a wide variety of problems, such as the TSP, to be NP-complete.
- ♦ For this purpose, we may reduce SAT to each of these problems in polynomial time.
- ♦ But before that, we introduce a simpler SAT problem, called 3SAT, and reduce SAT to a *normal form* of it, called CSAT, in polynomial time.
- ♦ That is, we want to perform reductions in a sequence of SAT $\Rightarrow$ CSAT $\Rightarrow$ 3SAT $\Rightarrow$ other problems.

10.3.1 Normal forms for Boolean Expressions

■ **Definitions –**

- ♦ A *literal* is either a variable or a negated one, like x and $\neg x$. And we use $\bar{y}$ for $\neg y$;

- A *clause* is a logical OR of one or more literals, like x, $x \vee y$, and $x \vee \bar{y} \vee z$.
- A boolean expression is said to be in *conjunction normal form* or *CNF*, if it is the AND of clauses.

- **Notations for compression –**
  - use + for $\vee$;
  - treat $\wedge$ as a product and use juxtaposition (no operator) for it (like concatenation).

- **Example 10.10 ---**
  - Boolean expression $(x\vee\neg y)\wedge(\neg x\vee z)$ now becomes $(x + \bar{y})(\bar{x} + z)$ which is in CNF.
  - Boolean expression $(x + y\bar{z})(x + y + z)(\bar{y} + \bar{z})$ is not in CNF because $x + y\bar{z}$ is not a clause.

- **Definition ---**
  - A boolean expression is said to be in k-CNF if it is the product of clauses, each being of the sum of exactly k distinct literals.
    - For example, $(x + \bar{y})(\bar{x} + z)$ is in 2-CNF because every clause has two literals.

- **Definitions ---**
  - CSAT is the problem: "given a boolean expression in CNF, is it satisfiable?"
  - kSAT is the problem: "given a boolean expression in k-CNF, is it satisfiable?"

- **Properties ---**
  - It can be proved that CSAT, 3SAT and kSAT with $k > 3$ are all NP-complete (later in Sections 10.3.2 & 10.3.3).
  - However, there are linear-time algorithms for 1SAT and 2SAT.

10.3.2 Converting Expressions to CNF

- **Concepts ---**
  - Two boolean expressions are said to be *equivalent* if they have the same result on any truth assignment to their variables.
  - If two expressions are equivalent, then either both are satisfiable or neither is.
  - We want to reduce SAT to CSAT, by taking an SAT instance E and convert it to a CSAT instance F such that F is satisfiable if and only if E is. (E and F need *not* be equivalent.)

- **Reduction of SAT to CSAT ---**
  - The above-mentioned reduction of SAT to CSAT consists of two parts:
    - Step 1 - *Push all ¬'s down* so that negations are **of** variables and the new expression becomes an AND and OR of literals (equivalent to the original).
    - Step 2 - Write the above result into a product F of clauses to become CNF *in polynomial time* (*not* need to be equivalent to the result of last step), so that F is satisfiable if and only if the old expression E is.
  - The 2nd step above is implemented by creating an *extension* of the original assignment T.
  - We say S is an *extension* of T if S assigns the same value as T to each variable that T assigns, but S may also assign a value to variables that T does not mention.
  - The 1st step above is implemented as follows.
    - $\neg(E\wedge F) \Rightarrow \neg(E)\vee\neg(F)$ (one of DeMorgan's laws)

- $\neg(E \vee F) \Rightarrow \neg(E) \wedge \neg(F)$ (the other of DeMorgan's laws)
- $\neg(\neg(E)) \Rightarrow E$ (Law of double negation)

■ **Example 10.11 ---**

The boolean expression $E = \neg((\neg(x + y))(\bar{x} + y))$ may be simplified by the above rules to be

$$E = \neg((\neg(x + y))(\bar{x} + y))$$

$$\Rightarrow \neg(\neg(x + y)) + \neg(\bar{x} + y)$$

$$\Rightarrow (x + y) + (\neg(\bar{x}))(\bar{y})$$

$$\Rightarrow x + y + x\bar{y}$$

which is an OR-and-AND expression of literals.

■ **Theorem 10.12 ---**

Every boolean expression E is equivalent to an expression F in which *the only negations occur in literals*, i.e., they apply directly to variables. Moreover, the length of F is *linear* in the number of symbols of E, and F can be constructed from E in *polynomial time*.

(for proof, see the textbook; if E has n operators, then F has no more then $2n - 1$ ones)

■ **A comment ---** the details of the 2nd step mentioned in the last section, Section 10.3.2, will be implemented in the proof of the following theorem.

■ **Theorem 10.13 ---**

CSAT is NP-complete.

Proof.

- ♦ We prove the theorem by reducing SAT to CSAT.
- ♦ The 1st step is to use Theorem 10.12 to convert the given instance of SAT to an expression E whose $\neg$'s are only in literals.
- ♦ We show the 2nd step of how to convert E to a CNF expression F in polynomial time here such that F is satisfiable if and only if E is.

- ♦ The construction of F is by an *induction* on the length of E.
  - **Basis:** if E consists of one or two symbols, then it is a literal which is also a clause, and so E is already in CNF.
  - **Induction:** assume every expression shorter than E has been converted into clauses. Two cases need be checked.

    (1) $E = E_1 \wedge E_2$.

    By induction, let F_1 and F_2 be CNF expressions derived from E_1 and E_2, respectively. Then, let $F = F_1 \wedge F_2$ which is also in CNF.

(2) $E = E_1 \vee E_2$.

By induction, let $F_1 = g_1 \wedge g_2 \wedge \ldots \wedge g_p$, $F_2 = h_1 \wedge h_2 \wedge \ldots \wedge h_q$ be CNF expressions derived from E_1 and E_2, respectively. Then, introduce a new variable y and let

$$F = (y + g_1) \wedge (y + g_2) \wedge \ldots \wedge (y + g_p) \wedge (\bar{y} + h_1) \wedge (\bar{y} + h_2) \wedge \ldots \wedge (\bar{y} + h_q).$$

♦ For the rest of the proof, see the textbook.

■ **Example 10.14 ---**

Given the boolean expression $E = x\bar{y} + \bar{x}(y + z)$, the corresponding CNF is constructed as follows.

♦ $y + z \Rightarrow (v + y)(\bar{v} + z)$ with v as an introduced variable.

♦ $\bar{x}(y + z) \Rightarrow \bar{x}(v + y)(\bar{v} + z)$.

♦ $x\bar{y} + \bar{x}(y + z) \Rightarrow x\bar{y} + \bar{x}(v + y)(\bar{v} + z) \Rightarrow (u + x)(u + \bar{y})(\bar{u} + \bar{x})(\bar{u} + v + y)(\bar{u} + \bar{v} + z)$ with u as an introduced variable.

■ **Theorem 10.15 ---**

3SAT is NP-complete.

Proof.

♦ First, 3SAT is in $\mathcal{NP}$ since SAT is in $\mathcal{NP}$.

♦ Next, we want to reduce CSAT to 3SAT. Since SAT has already been reduced to CSAT, it means that SAT can be reduced to 3SAT, and we are done.

♦ Given a CNF expression $E = e_1 \wedge e_2 \wedge \ldots \wedge e_k$ which is an instance of CSAT, we want to reduce it to an instance of 3SAT by transforming each e_i into a valid form F for 3SAT in the following way:

(1) If e_i is a single literal, say (x), then introduce two new variables u and v, and replace (x) by the four clauses $(x + u + v)(x + u + \bar{v})(x + \bar{u} + v)(x + \bar{u} + \bar{v})$. <u>The only way to make this expression true is for x to be true</u>, as desired.

(2) If e_i is the sum of two literals, $(x + y)$, then introduce a new variable z and replace e_i by $(x + y + z)(x + y + \bar{z})$. <u>The only way to make this expression true is for $(x + y)$ to be true</u>, as desired.

(3) If e_i is the sum of three literals, then it is already in the form required for 3-CNF.

(4) If $e_i = (x_1 + x_2 + \ldots + x_m)$ for $m \geq 4$, then introduce new variables $y_1, y_2, \ldots, y_{m-3}$ and replace e_i by the product of clauses

$$(x_1 + x_2 + y_1)(x_3 + \bar{y}_1 + y_2)(x_4 + \bar{y}_2 + y_3)\ldots(x_{m-2} + \bar{y}_{m-4} + y_{m-3})(x_{m-1} + x_m + \bar{y}_{m-3}). \quad \textbf{(10.2)}$$

If e_i is true to make E true because one of its literal x_j is true, then we may make y_1 through y_{j-2} as well as y_{j-1} through y_{m-3} true for the clauses of (10.2) above to be true.

♦ For other parts of the proof, see the textbook.

10.4 Additional NP-complete Problems

10.4.1~10.4.6

- **Theorems ---** the following problems are all NP-complete:
  - ♦ The problem of independent sets (IS)
  - ♦ The node-cover problem (NC)
  - ♦ The directed Hamilton-circuit problem (DHC)
  - ♦ The (undirected) Hamilton-circuit problem (HC)
  - ♦ The traveling salesman problem (TSP)

- **Comments ---**
  - ♦ The reductions of all the above problems and others studied before are illustrated in Fig. 10.12.
  - ♦ An *independent set* or *stable set* in a graph is a set of nodes, no two of which are adjacent (see Fig. 10.8 for an example).
  - ♦ A *node cover of a graph* is a set of nodes such that each edge of the graph is incident to at least one node of the set (cf. edge cover).

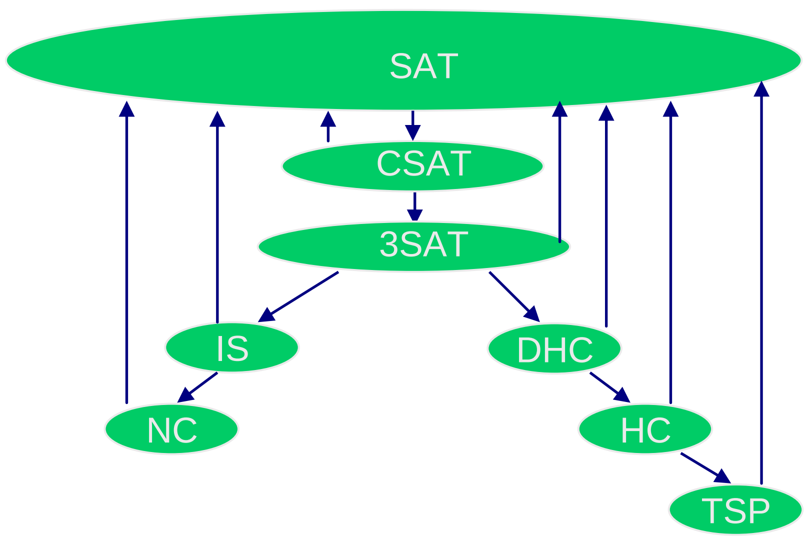

Figure 10.12 A hierarchy of problem reduction of the problems mentioned in this chapter.

- **A mention of some content in Chapter 11 ---**
  - ♦ Co- $\mathcal{NP}$ = complements of $\mathcal{NP}$.
  - ♦ See Figure 11.1.

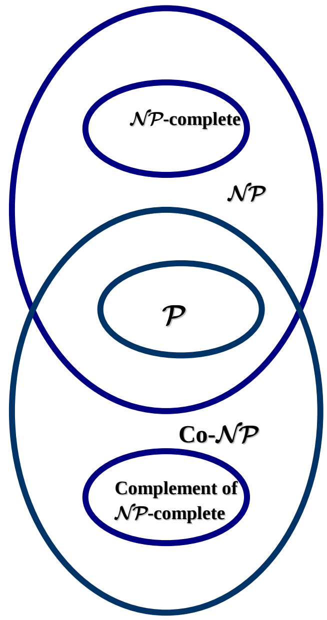

Figure 11.1 Relations of $\mathcal{NP}$-related problems.