Time Complexity and NP-Completeness
(Based on [Sipser 2006, 2013])

Yih-Kuen Tsay

Department of Information Management
National Taiwan University

Time Complexity

- Decidability of a problem merely indicates that the problem is computationally solvable *in principle*.
- It may not be solvable *in practice* if the solution requires an inordinate amount of time or memory.
- We shall introduce a way of measuring the time used to solve a problem.
- We then show how to classify problems according to the amount of time required.

Measuring Time Complexity

- Let $A = \{0^k 1^k \mid k \geq 0\}$.
- How much time does a single-tape TM need to decide A?
- A single-tape TM M_1 for A works as follows:
  1. Scan across the tape and *reject* if a 0 appears to the right of a 1.
  2. Repeat Stage 3 if both 0s and 1s remain on the tape.
  3. Scan across the tape, crossing off a single 0 and a single 1.
  4. If no 0s or 1s remain on the tape, *accept*; otherwise, *reject*.

Measuring Time Complexity

- Let $A = \{0^k 1^k \mid k \geq 0\}$.
- How much time does a single-tape TM need to decide A?
- A single-tape TM M_1 for A works as follows:
  1. Scan across the tape and *reject* if a 0 appears to the right of a 1.
  2. Repeat Stage 3 if both 0s and 1s remain on the tape.
  3. Scan across the tape, crossing off a single 0 and a single 1.
  4. If no 0s or 1s remain on the tape, *accept*; otherwise, *reject*.
- Intuitively, the running time of the Turing machine will be longer when the input is longer.

Measuring Time Complexity (cont.)

- We shall compute the running time of an algorithm purely as a function of the length of the string representing the input.

Definition (7.1)

Let M be a deterministic TM that halts on all inputs.
The **running time** or **time complexity** of M is the function $f : \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the *maximum* number of steps that M uses on any input of length n.
If $f(n)$ is the running time of M, we say that *M runs in time $f(n)$* or that *M is an $f(n)$ time Turing machine*.

Measuring Time Complexity (cont.)

- We shall compute the running time of an algorithm purely as a function of the length of the string representing the input.

Definition (7.1)

Let M be a deterministic TM that halts on all inputs.
The **running time** or **time complexity** of M is the function $f : \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the *maximum* number of steps that M uses on any input of length n.
If $f(n)$ is the running time of M, we say that *M runs in time $f(n)$* or that *M is an $f(n)$ time Turing machine*.

- We will mostly focus on *worst-case analysis*, measuring the longest running time of all inputs of a particular length.

Asymptotic Analysis

- The exact running time of an algorithm is a complex expression.
- We seek to understand the running time of the algorithm when it is run on large inputs.
- We do so by considering only the highest-order term of the expression of its running time (discarding the coefficient of that term and any lower-order terms).
- For example, if $f(n) = 6n^3 + 2n^2 + 20n + 45$, we say that f is *asymptotically* at most n^3.
- The *asymptotic notation*, or *big-O notation*, for describing this relationship is $f(n) = O(n^3)$.

Asymptotic Bounds

- Let $\mathcal{R}^+$ be the set of positive real numbers.

Definition (7.2)

Let f and g be two functions $f, g : \mathcal{N} \longrightarrow \mathcal{R}^+$.
We say that $f(n) = O(g(n))$ if positive integers c and n_0 exist so that, for every integer $n \geq n_0$,

$$f(n) \leq cg(n).$$

When $f(n) = O(g(n))$, we say that $g(n)$ is an (asymptotic) *upper bound* for $f(n)$.

Asymptotic Bounds (cont.)

- Intuitively, $f(n) = O(g(n))$ means that f is less than or equal to g if we disregard differences up to a constant factor.
- Big-O notation gives a way to say that one function is asymptotically *no more than* another.
- Big-O notation can appear in arithmetic expressions such as $O(n^2) + O(n)$ $(= O(n^2))$ and $2^{O(n)}$.
- Bounds of the form n^c, for $c > 0$, are called *polynomial bounds*.
- Bounds of the form 2^{n^c}, for $c > 0$, are called *exponential bounds*.

Asymptotic Bounds (cont.)

- To say that one function is asymptotically *less than* another, we use small-o notation.

Definition (7.5)

Let f and g be two functions $f, g : \mathcal{N} \longrightarrow \mathcal{R}^+$.
We say that $f(n) = o(g(n))$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

- For example, $\sqrt{n} = o(n)$ and $n \log n = o(n^2)$.

Analyzing Algorithms

- Consider the single-tape TM M_1 for deciding $\{0^k 1^k \mid k \geq 0\}$.
- Stage 1 takes $2n$ $(= O(n))$ steps: n steps to scan the input and another n steps to reposition the head at the left-hand end of the tape.
- Each execution of Stage 3 takes $2n$ steps and at most $n/2$ such executions are required. So, Stages 2 and 3 take at most $(n/2)2n$ $(= O(n^2))$ steps.
- Stage 4 takes n $(= O(n))$ steps.

Complexity Classes

Definition (7.7)

Let $t : \mathcal{N} \longrightarrow \mathcal{N}$ be a function.
Define the **time complexity class** $\text{TIME}(t(n))$ to be $\{L \mid L$ is a language decided by an $O(t(n))$ time Turing machine$\}$.

- $A\ (= \{0^k 1^k \mid k \geq 0\}) \in \mathrm{TIME}(n^2)$, since M_1 decides A in time $O(n^2)$.
- Is there a machine that decides A asymptotically faster?
- In other words, is A in $\text{TIME}(t(n))$ for $t(n) = o(n^2)$?

Complexity Classes (cont.)

- Below is a faster single-tape TM for deciding A $(= \{0^k 1^k \mid k \geq 0\})$.
- $M_2 =$ "On input string w:
  1. Same as Stage 1 of M_1.
  2. Repeat Stages 3 and 4 if both 0s and 1s remain on the tape.
  3. If the total number of 0s and 1s remaining is odd, *reject*.
  4. Cross off every other 0 and then every other 1.
  5. If no 0s or 1s remain on the tape, *accept*; otherwise, *reject*."
- The running time of M_2 is $O(n \log n)$ and hence $A \in \mathrm{TIME}(n \log n)$.

Complexity Classes (cont.)

- Below is an even faster TM, which has *two* tapes, for deciding A $(= \{0^k 1^k \mid k \geq 0\})$.
- M_3 = "On input string w:
  1. Same as Stage 1 of M_1.
  2. Copy the 0s on Tape 1 onto Tape 2.
  3. Scan across the 1s on Tape 1 until the end of the input, crossing off a 0 on Tape 2 for each 1. If there are not enough 0s, *reject*.
  4. If all the 0s have now been crossed off, *accept*; otherwise, *reject*."
- The running time of M_3 is $O(n)$.
- This indicates that the complexity of A depends on the model of computation selected.

Complexity Relationships among Models

IM NTU

Theorem (7.8)

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.

- Let M be a k-tape TM running in $t(n)$ time.
- A single-tape TM S simulating M requires $O(t(n))$ tape cells to store the current contents of M's tapes and the respective head positions.
- It takes $O(t(n))$ time for S to simulate each of M's $t(n)$ steps.
- So, the running time of S is $t(n) \times O(t(n)) = O(t^2(n))$.

Complexity Relationships among Models (cont.)

Definition (7.9)

The running time of a nondeterministic TM N is the function $f : \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the *maximum* number of steps that N uses on *any* branch of its computation on any input of length n.

Theorem (7.11)

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.

Complexity Relationships among Models (cont.)

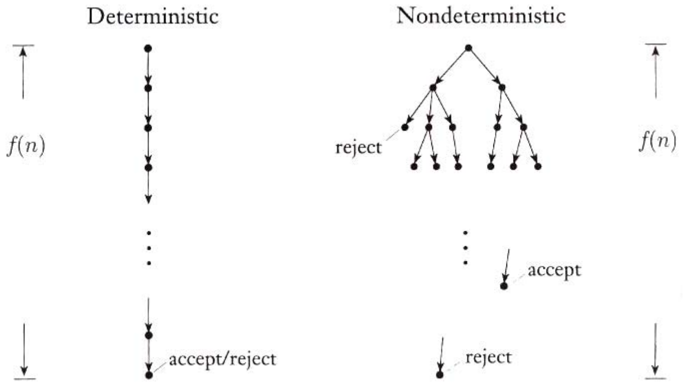


FIGURE 7.10
Measuring deterministic and nondeterministic time

Source: [Sipser 2006]

Complexity Relationships among Models (cont.)

- Every branch of N's computation tree has a length of at most $t(n)$.
- The total number of nodes in the tree is $O(b^{t(n)})$, where b is the maximum number of legal choices given by N's transition function.
- The running time of a simulating deterministic 3-tape TM is $O(t(n)) \times O(b^{t(n)}) = 2^{O(t(n))}$.
- The running time of a simulating deterministic single-tape TM is $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$.

Polynomial Time

- For our purposes, *polynomial differences* in running time are considered to be small, whereas *exponential differences* are considered to be large.
- Exponential time algorithms typically arise when we solve problems by searching through a space of solutions, called *brute-force search*.
- All "reasonable" deterministic computational models are *polynomially equivalent*, i.e., any one of them can simulate another with a polynomial increase in running time.
- We shall focus on aspects of time complexity theory that are unaffected by polynomial differences in running time.

The Class P

Definition (7.12)

P is the class of languages that are decidable in *polynomial* time on a *deterministic* single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k)$$

- P is invariant for all models of computing that are polynomially equivalent to the deterministic single-tape Turing machine.
- P roughly corresponds to the class of problems that are "realistically solvable" on a computer.

Analyzing Algorithms for P Problems

IM NTU

- Suppose that we have given a high-level description of a polynomial-time algorithm with stages. To analyze the algorithm,
    1. we first give a polynomial upper bound on the number of stages that the algorithm uses, and
    2. we then show that the individual stages can be implemented in polynomial time on a reasonable deterministic model.
- A "reasonable" encoding method for problems should be used, which allows for polynomial-time encoding and decoding of objects into natural internal representation or into other reasonable encodings.

Problems in P

- $PATH = \{\langle G, s, t\rangle \mid G$ is a directed graph that has a directed path from s to $t\}$.

Theorem (7.14)

$PATH \in P$.

- $M =$ "On input $\langle G, s, t\rangle$:
  1. Place a mark on node s.
  2. Repeat Stage 3 until no additional nodes are marked.
  3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
  4. If t is marked, *accept*; otherwise, *reject*."

Problems in P (cont.)

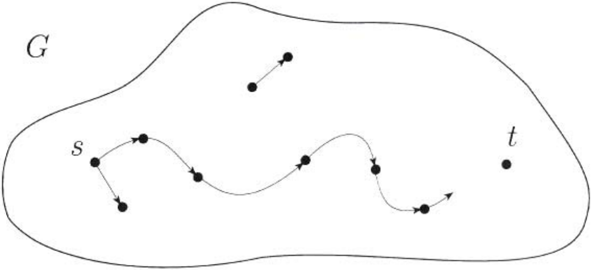


FIGURE 7.13
The *PATH* problem: Is there a path from s to t?

Source: [Sipser 2006]

Problems in P (cont.)

- $RELPRIME = \{\langle x, y\rangle \mid x$ and y are relatively prime$\}$.

Theorem (7.15)

$RELPRIME \in P$.

- The input size of a number x is $\log x$ (not x itself).
- $E =$ "On input $\langle x, y\rangle$:
  1. Repeat Stages 2 and 3 until $y = 0$.
  2. Assign $x \leftarrow x \bmod y$.
  3. Exchange x and y.
  4. Output x."
- $R =$ "On input $\langle x, y\rangle$:
  1. Run E on $\langle x, y\rangle$.
  2. If E's output is 1, *accept*; otherwise, *reject*."

Problems in P (cont.)

Theorem (7.16)

Every context-free language belongs to P.

We assume that a CFG in Chomsky normal form is given for the context-free language.

$D =$ "On input $w = w_1 w_2 \cdots w_n$,

1. If $w = \varepsilon$ and $S \rightarrow \varepsilon$ is a rule, *accept*.
2. For $i = 1$ to n,
3.   For each variable A,
4.   Is $A \rightarrow b$, where $b = w_i$, a rule?
5.   If yes, add A to $table(i, i)$.
6. For $l = 2$ to n,
7.   For $i = 1$ to $n - l + 1$,
8.   Let $j = i + l - 1$,
9.   For $k = i$ to $j - 1$,
10.   For each rule $A \rightarrow BC$,
11.   If $B \in table(i, k)$ and $C \in table(k + 1, j)$,
          then put A in $table(i, j)$.
12. If $S \in table(1, n)$, *accept*; otherwise, *reject*."

The Hamiltonian Path Problem

- A *Hamiltonian path* in a directed graph is a directed path that goes through each node exactly once.
- $HAMPATH = \{\langle G, s, t\rangle \mid G$ is a directed graph with a Hamiltonian path from s to $t\}$.
- We can easily obtain an exponential time algorithm for *HAMPATH*.
- No one knows whether *HAMPATH* is solvable in polynomial time.

The Hamiltonian Path Problem

- A *Hamiltonian path* in a directed graph is a directed path that goes through each node exactly once.
- $HAMPATH = \{\langle G, s, t\rangle \mid G$ is a directed graph with a Hamiltonian path from s to $t\}$.
- We can easily obtain an exponential time algorithm for *HAMPATH*.
- No one knows whether *HAMPATH* is solvable in polynomial time.
- However, *verifying* the existence of a Hamiltonian path may be much easier than *determining* its existence.

The Hamiltonian Path Problem (cont.)

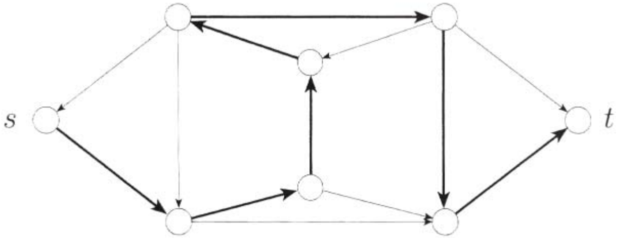


FIGURE 7.17
A Hamiltonian path goes through every node exactly once

Source: [Sipser 2006]

The Class NP

IM NTU

Definition (7.18)

A **verifier** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

The information represented by the symbol c is called a *certificate*, or *proof*, of membership in A.
A *polynomial-time verifier* runs in polynomial time in the length of w.

Definition (7.19)

NP is the class of *polynomially verifiable* languages, i.e., languages that have polynomial-time verifiers.

The Class NP (cont.)

Theorem (7.20)

A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.

- Let V be a verifier for $A \in \mathrm{NP}$ that runs in time n^k. Construct a decider N for A as follows.
- $N =$ "On input w of length n:
  1. Nondeterministically select string c of length n^k.
  2. Run V on input $\langle w, c \rangle$.
  3. If V accepts, *accept*; otherwise, *reject*."

The Class NP (cont.)

- Let N be a nondeterministic decider for a language A that runs in time n^k. Construct a verifier V for A as follows.
- $V =$ "On input $\langle w, c \rangle$:
  1. Simulate N on input w, treating each symbol of c as a description of the nondeterministic choice to make at each step.
  2. If this branch of N's computation accepts, *accept*; otherwise, *reject*."

The Class NP (cont.)

Definition (7.21)

$\mathrm{NTIME}(t(n)) = \{L \mid L$ is a language decided by an $O(t(n))$ time nondeterministic Turing machine$\}$.

Corollary (7.22)

$\mathrm{NP} = \bigcup_k \mathrm{NTIME}(n^k)$.

Analyzing Algorithms for NP Problems

- The class NP is insensitive to the choice of reasonable nondeterministic computational model.
- Like in the deterministic case, we use a high-level description to present a nondeterministic polynomial-time algorithm.
  1. Each stage of a nondeterministic polynomial-time algorithm must have an obvious implementation in polynomial time on a reasonable nondeterministic model.
  2. Every branch of its computation tree uses at most polynomially many stages.

Problems in NP

- A *clique* in an undirected graph is a subgraph, wherein every two nodes are connected by an edge.
- A *k-clique* is a clique that contains k nodes.
- $CLIQUE = \{\langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$.

Theorem (7.24)

CLIQUE is in NP.

Problems in NP (cont.)

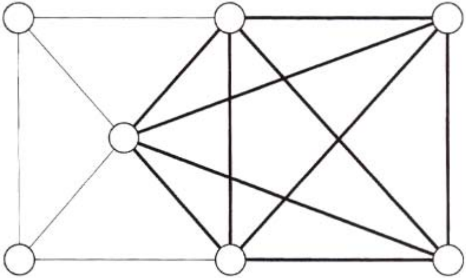


FIGURE **7.23**
A graph with a 5-clique

Source: [Sipser 2006]

Problems in NP (cont.)

- $V =$ "On input $\langle\langle G, k\rangle, c\rangle$:
  1. Test whether c is a set of k nodes in G.
  2. Test whether G contains all edges connecting nodes in c.
  3. If both pass, *accept*; otherwise, *reject*."

Problems in NP (cont.)

- $V =$ "On input $\langle\langle G, k\rangle, c\rangle$:
  1. Test whether c is a set of k nodes in G.
  2. Test whether G contains all edges connecting nodes in c.
  3. If both pass, *accept*; otherwise, *reject*."
- Alternatively,
  $N =$ "On input $\langle G, k\rangle$:
  1. Nondeterministically select a subset c of k nodes in G.
  2. Test whether G contains all edges connecting nodes in c.
  3. If yes, *accept*; otherwise, *reject*."

Problems in NP (cont.)

- $SUBSET\_SUM = \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $\{y_1, \cdots, y_l\} \subseteq S$, we have $\sum y_i = t\}$.

Theorem (7.25)

SUBSET_SUM is in NP.

- $V =$ "On input $\langle\langle S, t\rangle, c\rangle$:
  1. Test whether c is a collection of numbers that sum to t.
  2. Test whether S contains the numbers in c.
  3. If both pass, *accept*; otherwise, *reject*."

Problems in NP (cont.)

- $SUBSET\_SUM = \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $\{y_1, \cdots, y_l\} \subseteq S$, we have $\sum y_i = t\}$.

Theorem (7.25)

SUBSET_SUM is in NP.

- $V =$ "On input $\langle\langle S, t\rangle, c\rangle$:
  1. Test whether c is a collection of numbers that sum to t.
  2. Test whether S contains the numbers in c.
  3. If both pass, *accept*; otherwise, *reject*."
- Alternatively,
  $N =$ "On input $\langle S, t\rangle$:
  1. Nondeterministically select a subset c of the numbers in S.
  2. Test whether c is a collection of numbers that sum to t.
  3. If yes, *accept*; otherwise, *reject*."

The Class co-NP

- The complements of *CLIQUE* and *SUBSET_SUM*, namely $\overline{CLIQUE}$ and $\overline{SUBSET\_SUM}$, are not obviously members of NP.
- Verifying that something is *not* present seems to be more difficult than verifying that it *is* present.
- The complexity class co-NP contains the languages that are complements of languages in NP.

The Class co-NP

- The complements of *CLIQUE* and *SUBSET_SUM*, namely $\overline{CLIQUE}$ and $\overline{SUBSET\_SUM}$, are not obviously members of NP.
- Verifying that something is *not* present seems to be more difficult than verifying that it *is* present.
- The complexity class co-NP contains the languages that are complements of languages in NP.
- We do not know whether co-NP is different from NP.

P vs. NP

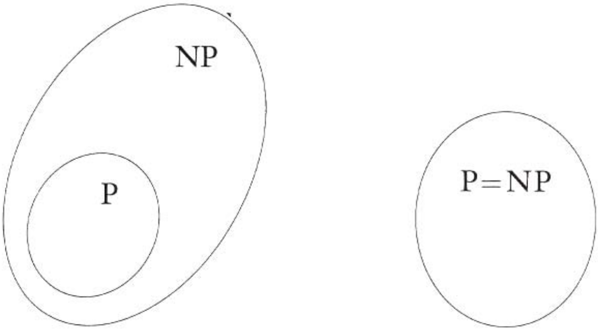


FIGURE 7.26
One of these two possibilities is correct

Source: [Sipser 2006]

NP-Completeness

- The complexity of certain problems in NP is related to that of the entire class [Cook and Levin].
- If a polynomial-time algorithm exists for any of the problems, all problems in NP would be polynomial-time solvable.
- These problems are called **NP-complete**.

NP-Completeness

IM NTU

- The complexity of certain problems in NP is related to that of the entire class [Cook and Levin].
- If a polynomial-time algorithm exists for any of the problems, all problems in NP would be polynomial-time solvable.
- These problems are called **NP-complete**.
- $SAT = \{\langle\phi\rangle \mid \phi$ is a satisfiable Boolean formula$\}$.

Theorem (7.27; Cook-Levin)

$SAT \in P$ iff $P = NP$.

(Equivalently, $SAT \notin P$ iff $P \neq NP$.)

Polynomial-Time Reducibility

- When problem A is *efficiently* reducible to problem B, an efficient solution to B can be used to solve A efficiently.

Definition (7.28)

A function $f : \Sigma^* \longrightarrow \Sigma^*$ is a **polynomial-time computable function** if some polynomial-time Turing machine M, on every input w, halts with just $f(w)$ on its tape.

Definition (7.29)

Language A is **polynomial-time mapping reducible** (polynomial-time reducible) to language B, written $A \leq_{\mathrm{P}} B$, if there is a polynomial-time computable function $f : \Sigma^* \longrightarrow \Sigma^*$, where for every w,

$$w \in A \Longleftrightarrow f(w) \in B.$$

Polynomial-Time Reducibility (cont.)

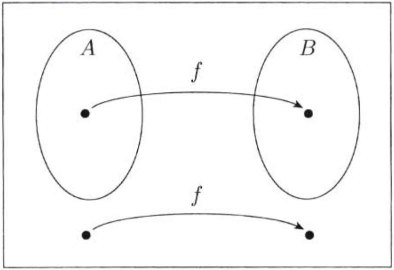


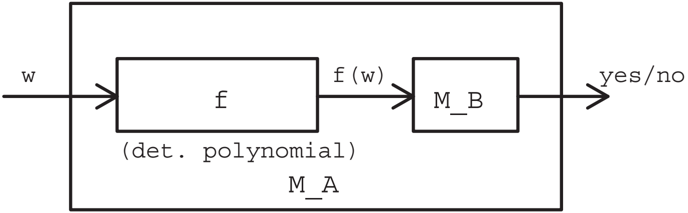
FIGURE 7.30
Polynomial time function f reducing A to B

Source: [Sipser 2006]

Function f transforms the membership problem of A to that of B.

Polynomial-Time Reducibility (cont.)

- $A \leq_{\mathrm{P}} B$, like $A \leq_{\mathrm{M}} B$, means that a Turing machine M_A for A can be constructed from a given Turing machine M_B for B.



- Furthermore, if M_B is a polynomial-time decider for B, then M_A is a polynomial-time decider for A.

Polynomial-Time Reducibility (cont.)

IM NTU

Theorem (7.31)

If $A \leq_{\mathrm{P}} B$ and $B \in P$, then $A \in P$.

- Let M_B be a polynomial-time algorithm deciding B and f be the polynomial-time reduction from A to B.
- $M_A =$ "On input w:
  1. Compute $f(w)$.
  2. Run M_B on input $f(w)$ and output whatever M_B outputs."

Example Polynomial-Time Reducibility

- A Boolean formula is in *conjunctive normal form*, called a CNF-formula, if it comprises several clauses connected with $\wedge$s, as in

$$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6})$$

- It is a 3CNF-formula if all the clauses have three literals, as in

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_4 \vee x_5 \vee x_6)$$

- $3SAT = \{\langle\phi\rangle \mid \phi$ is a satisfiable 3CNF-formula$\}$.

Theorem (7.32)

$3SAT$ is polynomial-time reducible to $CLIQUE$.

Example Polynomial-Time Reducibility (cont.)

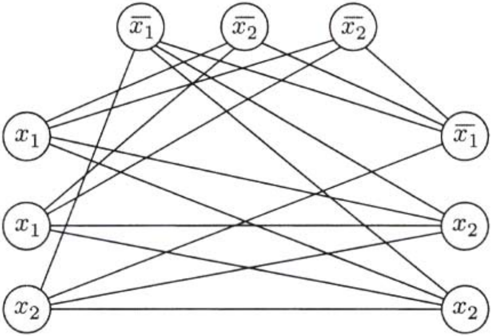


FIGURE 7.33
The graph that the reduction produces from
$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

Source: [Sipser 2006]

NP-Completeness

IM NTU

Definition (7.34)

A language B is **NP-complete** if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial-time reducible to B (in which case, we say that B is NP-hard).

Theorem (7.35)

If B is NP-complete and $B \in \mathrm{P}$, then $\mathrm{P} = \mathrm{NP}$.

NP-Completeness (cont.)

- The polynomial-time reducibility relation $\leq_{\mathrm{P}}$ is a transitive relation. (Mathematically, $\leq_{\mathrm{P}}$ is a pre-order, i.e., it is reflexive and transitive.)

NP-Completeness (cont.)

- The polynomial-time reducibility relation $\leq_{\text{P}}$ is a transitive relation. (Mathematically, $\leq_{\text{P}}$ is a pre-order, i.e., it is reflexive and transitive.)
- Transitivity of $\leq_{\text{P}}$ allows one to prove NP-completeness of a problem via a known NP-complete problem.
- If B is NP-complete and $B \leq_{\text{P}} C$, then every problem in P is polynomial-time reducible to C.

Theorem (7.36)

If B is NP-complete and $B \leq_{\text{P}} C$ for some $C \in \text{NP}$, then C is NP-complete.

The Cook-Levin Theorem

Theorem (7.37)

SAT is NP-complete.

- *SAT* is in NP, as a nondeterministic polynomial-time TM can guess an assignment to a given formula ϕ and accept if the assignment satisfies ϕ.
- We next construct a polynomial-time reduction for each language A in NP to *SAT*.
- The reduction takes a string w and produces a Boolean formula ϕ that simulates the NP machine N for A on input w.
- Assume that N runs in time n^k (with some constant difference) for some $k > 0$.

The Cook-Levin Theorem (cont.)

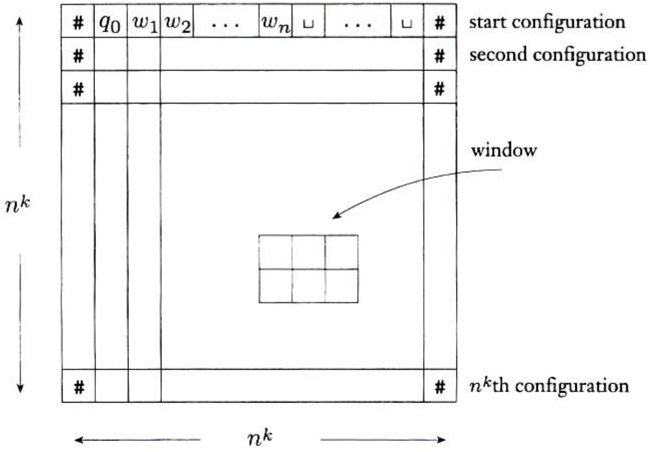


FIGURE 7.38
A tableau is an $n^k \times n^k$ table of configurations

Source: [Sipser 2006]

The Cook-Levin Theorem (cont.)

- If N accepts, ϕ has a satisfying assignment that corresponds to the accepting computation.
- If N rejects, no assignment satisfies ϕ.
- Let $C = Q \cup \Gamma \cup \{\#\}$. For $1 \leq i, j \leq n^k$ and $s \in C$, we have a variable $x_{i,j,s}$.
- Variable $x_{i,j,s}$ is assigned 1 iff $cell[i,j]$ contains an s.
- Construct ϕ as $\phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{move}}$, where ...
  - Size of ϕ_{cell}: $O(n^{2k})$.
  - Size of ϕ_{start}: $O(n^k)$.
  - Size of ϕ_{accept}: $O(n^{2k})$.
  - Size of ϕ_{move}: $O(n^{2k})$.

The Cook-Levin Theorem (cont.)

$$\phi_{\mathrm{cell}} = \bigwedge_{1 \le i,j \le n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{s,t \in C, s \ne t} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right] .$$

$$\begin{aligned} \phi_{\mathrm{start}} = \; & x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \cdots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \cdots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#} \; . \end{aligned}$$

$$\phi_{\mathrm{accept}} = \bigvee_{1 \le i,j \le n^k} x_{i,j,q_{\mathrm{accept}}} \; .$$

$$\phi_{\mathrm{move}} = \bigwedge_{1 \le i \le (n^k-1), 2 \le j \le (n^k-1)} (\text{window } (i,j) \text{ is legal}) \; .$$

The Cook-Levin Theorem (cont.)

IM NTU

- Assume that $\delta(q_1, a) = \{(q_1, b, R)\}$ and $\delta(q_1, b) = \{(q_2, c, L), (q_2, a, R)\}$.

(a)

a	q_1	b
q_2	a	c

(b)

a	q_1	b
a	a	q_2

(c)

a	a	q_1
a	a	b

(d)

#	b	a
#	b	a

(e)

a	b	a
a	b	q_2

(f)

b	b	b
c	b	b

FIGURE 7.39
Examples of legal windows

Source: [Sipser 2006]

The Cook-Levin Theorem (cont.)

- Assume that $\delta(q_1, a) = \{(q_1, b, R)\}$ and $\delta(q_1, b) = \{(q_2, c, L), (q_2, a, R)\}$.

(a)

a	b	a
a	a	a

(b)

a	q_1	b
q_1	a	a

(c)

b	q_1	b
q_2	b	q_2

FIGURE 7.40
Examples of illegal windows

Source: [Sipser 2006]

The Cook-Levin Theorem (cont.)

- The condition "window (i, j) is legal" can be expressed as

$$\bigvee_{a_1,\cdots,a_6 \text{ legal}} \begin{array}{l}(x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge \\ x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6})\end{array}$$

Another Two NP-Complete Problems

IM NTU

Theorem

3SAT is NP-complete.

- The proof of the Cook-Levin theorem can be modified so that the Boolean formula involved is in conjunctive normal form.
- A CNF-formula can be converted in polynomial time to a 3CNF-formula (with a length polynomially bounded in the length of the CNF-formula).
- If a clause contains l literals $(a_1 \vee a_2 \vee \cdots \vee a_l)$, we can replace it with the $l - 2$ clauses

$$(a_1 \vee a_2 \vee z_1) \wedge (\overline{z_1} \vee a_3 \vee z_2) \wedge (\overline{z_2} \vee a_4 \vee z_3) \wedge \\ \cdots \wedge (\overline{z_{l-4}} \vee a_{l-2} \vee z_{l-3}) \wedge (\overline{z_{l-3}} \vee a_{l-1} \vee a_l)$$

Another Two NP-Complete Problems (cont.)

IM NTU

Theorem

CLIQUE is NP-complete.

CLIQUE is in NP and $3SAT \leq_{\mathrm{P}} CLIQUE$.