Slides for a Course on the Analysis and Design of Algorithms

Chapter 9: The Complexity Classes $\mathcal{P}$ and $\mathcal{NP}$

Stephen J. Hegner
Department of Computing Science
Umeå University
Sweden
`hegner@cs.umu.se`
`http://www.cs.umu.se/~hegner`

9. The Complexity Classes $\mathcal{P}$ and $\mathcal{NP}$

9.1 Abstract Problems and Reductions

9.1.1 Background

- In this chapter, the question of how the inherent complexity of problem A is compared to that of problem B is investigated.
- To keep things focused, attention will be restricted to *decision problems*; that is, problems which have a yes/no answer.
- Because of the nature of this sort of question, it is necessary to begin by developing some relatively abstract background.

9.1.2 Abstract decision problems An *abstract decision problem* is an ordered pair $P = (I, \rho)$ in which

(a) I is a set of *problem instances*.

(b) The function

$$\rho : I \to \{0,1\}$$

assigns to each problem instance a truth value, with 0 representing false and 1 representing true.

9.1.3 Reductions Let $P_1 = (I_1, \rho_1)$ and $P_2 = (I_2, \rho_2)$ be abstract decision problems. A *reduction* of P_1 to P_2 is a function

$$e : I_1 \to I_2$$

with the property that the following diagram commutes.

$$\begin{tikzcd}
I_1 \arrow[r, "\rho_1"] \arrow[d, "e"'] & \{0,1\} \\
I_2 \arrow[ur, "\rho_2"] &
\end{tikzcd}$$

- To formalize the notion of computing the solution of an abstract decision problem, a special subclass, known as language problems is introduced.

9.1.4 Language problems

(a) For a finite set Σ, Σ^* denotes the set of all finite strings of elements of Σ.

(b) An abstract decision problem $P = (I, \rho)$ is called a *language decision problem* if $I \subseteq \Sigma^*$ for some finite set Σ.

- In the above context, Σ is often called an *alphabet*.
- To address the problem of computational solution of abstract language problems, a very abstract notion of a computer – the Turing machine – is introduced.

9.2 The Deterministic Turing Machine

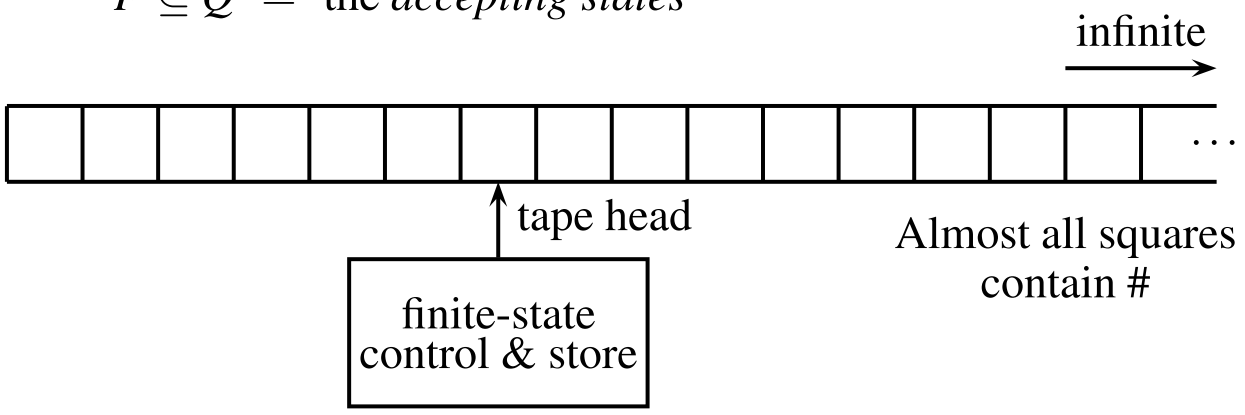
9.2.1 Basic definition: deterministic Turing machines

(a) A *deterministic Turing machine* (*DTM*) is a seven-tuple

$$M = (Q, \Sigma, \Gamma, \#, \delta, q_0, F)$$

in which

$$\begin{aligned} Q &= \text{a finite set of } \textit{states} \\ \Sigma &= \text{a finite } \textit{input alphabet} \\ \Gamma &= \text{a finite } \textit{tape alphabet} \text{ with } \Sigma \subseteq \Gamma \\ \# \in \Gamma \setminus \Sigma &= \text{the } \textit{blank symbol} \\ \delta : Q \times \Gamma \to Q \times \Gamma &\times \{R, L, N\} \\ &= \text{the } \textit{state-transition function} \text{ (a partial function)} \\ q_0 \in Q &= \text{the } \textit{initial state} \\ F \subseteq Q &= \text{the } \textit{accepting states} \end{aligned}$$

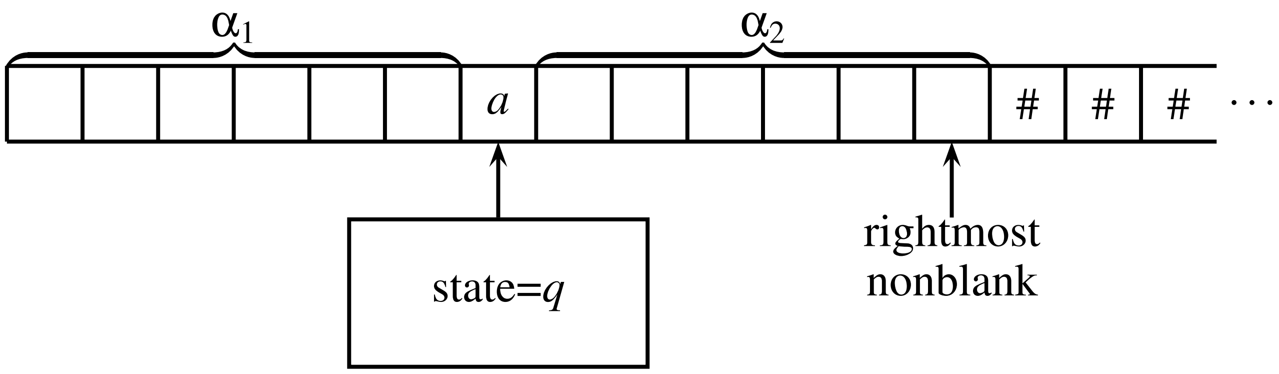


Interpretation of δ:

$$\underset{\substack{\uparrow\\ \text{current}\\ \text{state}}}{Q} \times \underset{\substack{\uparrow\\ \text{contents of}\\ \text{current}\\ \text{tape square}}}{\Gamma} \longrightarrow \underset{\substack{\uparrow\\ \text{next state}}}{Q} \times \underset{\substack{\uparrow\\ \text{new value}\\ \text{of current}\\ \text{tape square}}}{\Gamma} \times \underset{\substack{\uparrow\\ \text{direction}\\ \text{to move}\\ \text{tape head}}}{\{R, L, N\}}$$

(b) An *instantaneous description* (*ID*) for M is a quadruple

$$(\alpha_1, a, \alpha_2, q) \in \Gamma^* \times \Gamma \times \Gamma^* \times Q$$



(c) The set of all ID's for M is denoted $\mathcal{D}_M$.

- Note that only finitely many tape squares may contain other than # in an ID.

9.2.2 The next-ID function for a DTM Let $M = (Q, \Sigma, \Gamma, \#, \delta, q_0, F)$ be a deterministic Turing machine. The *next-ID function* for M is the partial function

$$\hat{\delta}_M : \mathcal{D}_M \to \mathcal{D}_M$$

defined by the following rules. For

$$D = (\langle a_1, \ldots, a_m \rangle, a, \langle b_1, \ldots, b_n \rangle, q) \in \mathcal{D}_M$$

(i) If $\delta(q, a) = (q', a', L)$ (move tape head left), then

$$\hat{\delta}_M(D) = \begin{cases} (\langle a_1, \ldots, a_{m-1} \rangle, a_m, \langle a', b_1, \ldots, b_n \rangle, q') & \text{if } m \geq 1 \\ \text{undefined} & \text{if } m = 0 \end{cases}$$

(ii) If $\delta(q, a) = (q', a', R)$ (move tape head right), then

$$\hat{\delta}_M(D) = \begin{cases} (\langle a_1, \ldots, a_m, a' \rangle, b_1, \langle b_2, \ldots, b_n \rangle, q') & \text{if } n \geq 1 \\ (\langle a_1, \ldots, a_m, a' \rangle, \#, \varepsilon, q') & \text{if } n = 0 \end{cases}$$

- In the above ε is the empty string.

(iii) If $\delta(q, a) = (q', a', N)$ (stationary tape head), then

$$\hat{\delta}_M(D) = (\langle a_1, \ldots, a_m \rangle, a', \langle b_1, \ldots, b_n \rangle, q')$$

(iv) If $\delta(q, a)$ is undefined, so too is $\hat{\delta}_M(D)$.

9.2.3 The global transition function for a DTM Let $M = (Q, \Sigma, \Gamma, \#, \delta, q_0, F)$ be a deterministic Turing machine.

(a) For each $n \in \mathbb{N}$, define the partial function

$$\hat{\delta}^n_M : \mathcal{D}_M \to \mathcal{D}_M$$

inductively as follows.

$$\begin{aligned} \hat{\delta}^0_M(D) &= D \quad \text{for all } D \in \mathcal{D}_M \\ \hat{\delta}^{n+1}_M(D) &= \begin{cases} \hat{\delta}_M(\hat{\delta}^n_M(D)) & \text{if } \hat{\delta}^n_M(D) \text{ is defined;} \\ \text{undefined} & \text{if } \hat{\delta}^n_M(D) \text{ is not defined.} \end{cases} \end{aligned}$$

(b) Define the partial function

$$\hat{\delta}^*_M : \mathcal{D}_M \to \mathcal{D}_M$$

by

$$\hat{\delta}^*_M(D) = \begin{cases} \hat{\delta}^n_M(D) & \text{if } \hat{\delta}^n_M(D) \text{ is defined but } \hat{\delta}^{n+1}_M(D) \text{ is not defined;} \\ \text{undefined} & \text{if } \hat{\delta}^n_M(D) \text{ is defined. for all } n \in \mathbb{N} \end{cases}$$

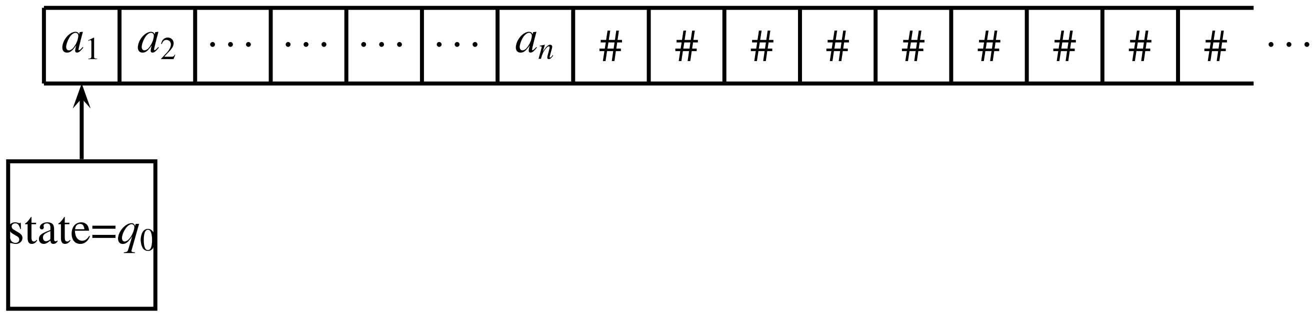
- $\hat{\delta}^*_M$ is called the *global transition function* for M.

(c) The machine M is said to *halt* in configuration D' from starting configuration D if $\hat{\delta}^*_M(D) = D'$. In that case, D' is called a *halting configuration* for M.

9.2.4 The notion of acceptance for a DTM Let $M = (Q, \Sigma, \Gamma, \#, \delta, q_0, F)$ be a deterministic Turing machine.

(a) For each $w = \langle a_1, a_2, \ldots, a_n \rangle \in \Sigma^*$, the *input configuration* for w, denoted $\mathcal{I}(M, w)$, is defined as follows.

$$\mathcal{I}(M,w) = \begin{cases} (\varepsilon, a_1, \langle a_2, \ldots, a_n \rangle, q_0) & \text{if } n \geq 1 \\ (\varepsilon, \#, \varepsilon, q_0) & \text{if } n = 0 \end{cases}$$



(b) The language $\mathcal{L}(M)$ accepted by M is the set of all $w \in \Sigma^*$ such that

(i) $\hat{\delta}^*_M(\mathcal{I}(M,w))$ is defined (say $\hat{\delta}^*_M(\mathcal{I}(M,w)) = (\alpha_1, a, \alpha_2, q)$), with

(ii) $q \in F$.

(c) M is a *decider* if $\hat{\delta}^*_M(\mathcal{I}(M,w))$ is defined for every $w \in \Sigma^*$. In this case, it is called a *decider for* $\mathcal{L}(M)$.

- Note that if M is a decider, then $\hat{\delta}^*_M(I(w))$ is always defined.
- The only question for acceptance is whether or not the final state q is in F.

9.2.5 The notion of a computation for a DTM Let $M = (Q, \Sigma, \Gamma, \#, \delta, q_0, F)$ be a deterministic Turing machine.

- Informally, a computation is just a sequence of legal configurations, starting with an initial configuration and terminating with a halting configuration.

(a) The symbol $\vdash_M$ is used to denote a single step in the computation of a Turing machine. Thus,

$$D_1 \vdash_M D_2 \quad \text{means that} \quad D_2 = \hat{\delta}_M(D_1)$$

(b) Formally, a *finite computation* of M is a finite sequence

$$\langle D_0, D_1, \ldots, D_n \rangle$$

with the following properties:

(i) D_0 is an input configuration of the form $\mathcal{I}(M, w)$ for some $w \in \Sigma^*$.

(ii) D_n is a halting configuration for M.

(iii) $D_0 \vdash_M D_1 \vdash_M \ldots \vdash_M D_n$.

- Note in particular that $D_n = \hat{\delta}^*_M(D_0)$.

9.2.6 The function computed by a DTM Let $M = (Q, \Sigma, \Gamma, \#, \delta, q_0, F)$ be a deterministic Turing machine.

(a) For any

$$D = (\langle a_1, \ldots, a_m \rangle, a, \langle b_1, \ldots, b_n \rangle, q) \in \mathcal{D}_M$$

define $\mathsf{TapeVal}_1(D)$ to be the longest initial substring of $\langle a_1, \ldots, a_n, a, b_1, \ldots, b_n \rangle$ which does not contain a #.

- Note that if $a_1 = \#$, then $\mathsf{TapeVal}_1(D) = \varepsilon$.

(b) For $w \in \Sigma^*$, The partial function

$$f_M : \Sigma^* \to \Sigma^*$$

computed by M is defined by

$$f_M(w) = \begin{cases} \mathsf{TapeVal}_1(\hat{\delta}^*_M(\mathcal{I}(M, w))) & \text{if } \hat{\delta}^*_M(\mathcal{I}(M, w)) \text{ is defined;} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

- In words, the partial function computed by M takes input string w represented as the initial configuration $\mathcal{I}(M, w)$.
- The result is defined iff the machine halts.
- In the case that it does halt, $f_M(w)$ is read as the string starting at the leftmost position on the tape, and continuing until the first # is encountered.
- Neither the final state nor the final position of the tape head is used in determining this value.

9.2.7 Computable functions

Let Σ be a finite alphabet

(a) A partial function

$$f : \Sigma^* \to \Sigma^*$$

is *computable* if there is some Turing machine $M = (Q, \Sigma, \Gamma, \#, \delta, q_0, F)$ with $f = f_M$.

- Note that *partial* here means that the function *need* not be defined for all values of Σ^*, but it may be.
- A *total* function is a special case of a partial function.

9.2.8 Turing machines and universal computation

- Recall from the very first part of these notes that *Church's thesis* posits that there is an ideal upper bound on the power of all programming devices.
- It is well known that (deterministic) Turing machines lie in this group of "universal" computing devices.
- Thus, a Turing machine can compute whatever any other computing device can.
- Note that the Turing model seems to require that the algorithm be "burned into" the hardware, but this is not in fact necessary.
- A Turing machine can be designed which is a universal interpreter (or, at least, an interpreter for your favorite programming language).

9.3 The Nondeterministic Turing Machine

9.3.1 Basic definition: nondeterministic Turing machines

(a) Formally, a *nondeterministic Turing machine* (*NDTM*) $M = (Q, \Sigma, \Gamma, \#, \delta, q_0, F)$ follows exactly the same definition as that of a deterministic Turing machine of 9.2.1, save that the state-transition function is now a total function of the form

$$\delta : Q \times \Gamma \to \mathbf{2}^{Q \times \Gamma \times \{R,L,N\}}$$

- It is difficult to give an NDTM a good "physical" model.
- It is more of a mathematical abstraction, albeit a useful one.
- The idea is that each element of $\delta(q,a)$ gives a possible next move for the machine.
- In its operation, the machine selects one of these moves in some unspecified and unrepeatable way.
- It is easy to see that this notion subsumes that of a deterministic Turing machine.
- Given a DTM $M = (Q, \Sigma, \Gamma, \#, \delta, q_0, F)$, define the associated NDTM as $M = (Q, \Sigma, \Gamma, \#, \delta', q_0, F)$, with

$$\delta'(q,a) = \begin{cases} \{\delta(q,a)\} & \text{if } \delta(q,a) \text{ is defined;} \\ \emptyset & \text{if } \delta(q,a) \text{ is not defined.} \end{cases}$$

9.3.2 The next-ID function for an NDTM Let $M = (Q, \Sigma, \Gamma, \#, \delta, q_0, F)$ be a nondeterministic Turing machine. The *next-ID function* for M is the total function

$$\hat{\delta}_M : \mathcal{D}_M \to \mathbf{2}^{\mathcal{D}_M}$$

defined by the following rules. For

$$D = (\langle a_1, \ldots, a_m \rangle, a, \langle b_1, \ldots, b_n \rangle, q) \in \mathcal{D}_M$$

$D' \in \hat{\delta}_M(D)$ iff one of the following conditions is satisfied:

(i) $m \geq 1$, $(q', a', L) \in \delta(q, a)$, and

$$D' = (\langle a_1, \ldots, a_{m-1} \rangle, a_m, \langle a', b_1, \ldots, b_n \rangle, q')$$

(ii) $n \geq 1$, $(q', a', R) \in \delta(q, a)$, and

$$D' = (\langle a_1, \ldots, a_m, a' \rangle, b_1, \langle b_2, \ldots, b_n \rangle, q')$$

(ii′) $n = 0$, $(q', a', R) \in \delta(q, a)$, and

$$D' = (\langle a_1, \ldots, a_m, a' \rangle, \#, \varepsilon, q')$$

(iii) $(q', a', N) \in \delta(q, a)$ and

$$D' = (\langle a_1, \ldots, a_m \rangle, a', \langle b_1, \ldots, b_n \rangle, q')$$

9.3.3 The global transition function for an NDTM Let $M = (Q, \Sigma, \Gamma, \#, \delta, q_0, F)$ be a nondeterministic Turing machine.

(a) For each $n \in \mathbb{N}$, define the partial function

$$\hat{\delta}_M^n : \mathcal{D}_M \to \mathbf{2}^{\mathcal{D}_M}$$

inductively as follows.

$$\begin{aligned} \hat{\delta}_M^0(D) &= \{D\} \quad \text{for all } D \in \mathcal{D}_M \\ \hat{\delta}_M^{n+1}(D) &= \bigcup_{D' \in \hat{\delta}_M^n(D)} \hat{\delta}_M(D') \end{aligned}$$

(b) Define the partial function

$$\hat{\delta}_M^* : \mathcal{D}_M \to \mathbf{2}^{\mathcal{D}_M}$$

by

$$\hat{\delta}_M^*(D) = \bigcup_{n \in \mathbb{N}} \{\hat{\delta}_M^n(D) \mid \hat{\delta}_M^{n+1}(D) = \emptyset\}$$

- $\hat{\delta}_M^*$ is called the *global transition function* for M.

(c) The machine M is said to *halt* in configuration D' from starting configuration D if $D' \in \hat{\delta}_M^*(D)$.

9.3.4 The notion of acceptance for an NDTM Let $M = (Q, \Sigma, \Gamma, \#, \delta, q_0, F)$ be a nondeterministic Turing machine.

- (a) The notion of *input configuration* is defined exactly as in 9.2.4; *i.e.*,

(a) For each $w = \langle a_1, a_2, \ldots, a_n \rangle \in \Sigma^*$, the *input configuration* for w, denoted $\mathcal{I}(M, w)$, is defined as follows.

$$\mathcal{I}(M,w) = \begin{cases} (\varepsilon, a_1, \langle a_2, \ldots, a_n \rangle, q_0) & \text{if } n \geq 1 \\ (\varepsilon, \#, \varepsilon, q_0) & \text{if } n = 0 \end{cases}$$

(b) The language $\mathcal{L}(M)$ accepted by M is the set of all $w \in \Sigma^*$ such that there is $D' = (\alpha_1, a, \alpha_2, q) \in \hat{\delta}^*_M(\mathcal{I}(M, w))$ with the property that $q \in F$.

- The extension of the notion of a decider to the NDTM context relies upon the distinction between finite and infinite computations, and will be presented in 9.3.5(d) below.
- Note that the definition of acceptance for an NDTM is inherently asymmetric.
- For $w \in \Sigma^*$ to be accepted by M, it suffices to find one nondeterministic computation with initial configuration $\mathcal{I}(M, w)$ which halts in an accepting state.
- For $w \in \Sigma^*$ to be rejected by M, it must be shown that **every** nondeterministic computation with initial configuration $\mathcal{I}(M, w)$ fails to halt in an accepting state.

9.3.5 The notion of a computation for an NDTM Let $M = (Q, \Sigma, \Gamma, \#, \delta, q_0, F)$ be a nondeterministic Turing machine.

(a) As in the case of a DTM, the symbol $\vdash_M$ is used to denote a single step in the computation of a Turing machine. Thus,

$$D_1 \vdash_M D_2 \quad \text{means that} \quad D_2 \in \hat{\delta}_M(D_1)$$

(b) Formally, a *finite computation* of M is a finite sequence

$$\langle D_0, D_1, \ldots, D_n \rangle$$

with the following properties:

(i) D_0 is of the form $\mathcal{I}(M, w)$ for some $w \in \Sigma^*$.

(ii) D_n is a halting configuration for M.

(iii) $D_0 \vdash_M D_1 \vdash_M \ldots \vdash_M D_n$.

- Note in particular that $D_n \in \hat{\delta}^*_M(D_0)$.

(c) An *infinite computation* of M is an infinite sequence of the form

$$\langle D_0, D_1, D_2, \ldots \rangle$$

with the following properties:

(i) D_0 is of the form $\mathcal{I}(M, w)$ for some $w \in \Sigma^*$.

(ii) For all $n \in \mathbb{N}$, $D_n \vdash_M D_{n+1}$.

(d) M is a *decider* if it does not have any infinite computations. It is then called a *decider for* $\mathcal{L}(M)$.

- It is easy to see that this notion of decider is consistent with that of 9.2.4(c) for DTM's.

9.3.6 Computing a function with an NDTM

- While it is possible to define some notions of an NDTM computing a function, they are complex and, in any case, irrelevant to the current context.
- They will not be expanded here.
- Therefore, in these notes, *computable functions will **always** be computed on **deterministic** Turing machines.*
- Accepters and deciders may of course be nondeterministic.

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

9.4.1 Length and complexity Let $M = (Q, \Sigma, \Gamma, \#, \delta, q_0, F)$ be either a deterministic or else a nondeterministic Turing machine which is a decider.

(a) The length of an input configuration $\mathcal{I}(M, w)$ is just the length $\mathsf{Length}(w)$ of the string w.

(b) The length of a finite computation

$$\langle D_0, D_1, \ldots, D_n \rangle$$

is n, the number of steps in the computation.

(c) Let $A \subseteq \Sigma^*$, and let

$$g : \mathbb{N} \to \mathbb{N}$$

be any function on the natural numbers. M is said to have complexity $\bar{O}(g)$ on A if, for any $w \in A$, the length of every computation with initial configuration $\mathcal{I}(M, w)$ is no more than $g(\mathsf{Length}(w)))$.

- Note that this definition does *not* employ an "up to constant multiple notion," as does the usual $O(g)$ notion.

9.4.2 Polynomial-time computations

(a) Let $M = (Q, \Sigma, \Gamma, \#, \delta, q_0, F)$ be a Turing machine, either deterministic or nondeterministic, and let $A \subseteq \Sigma^*$. M is said to be *polynomial* on A if there is a polynomial

$$\begin{array}{rcl} p : \mathbb{N} & \to & \mathbb{N} \\ n & \mapsto & \sum_{i=0}^{k} a_k \cdot n^k \;\; (a_i \in \mathbb{N}) \end{array}$$

such that M has complexity $\bar{O}(p)$ on A.

(b) Given a finite alphabet Σ and a subset $A \subseteq \Sigma^*$, a computable function

$$f : \Sigma^* \to \Sigma^*$$

is said to be *polynomial on* A if there is a deterministic Turing machine $M = (Q, \Sigma, \Gamma, \#, \delta, q_0, F)$ with $f = f_M$ which is polynomial on A.

(c) A language decision problem $P = (I, \rho)$ with $I \subseteq \Sigma^*$ is *deterministic polynomial* (resp. *nondeterministic polynomial*) if there is a DTM (resp. NTDM) $M = (Q, \Sigma, \Gamma, \#, \delta, q_0, F)$ such that:

  (i) M is a decider; and

  (ii) M is polynomial on $\{\mathcal{I}(M, w) \mid w \in I\}$.

9.4.3 Definitions of $\mathcal{P}$ and $\mathcal{NP}$

(a) $\mathcal{P}$ denotes the class of all deterministic polynomial language problems.

(b) $\mathcal{NP}$ denotes the class of all nondeterministic polynomial language problems.

- Clearly $\mathcal{P} \subseteq \mathcal{NP}$.
- The question of whether or not

$$\mathcal{P} = \mathcal{NP}$$

holds is perhaps the most famous outstanding problem in the foundations of computer science.

(c) Problems which are known to lie in $\mathcal{P}$ are often termed *tractable*.

(d) Problems which lie outside of $\mathcal{P}$ are often termed *intractable*.

- Thus, the question of whether or not

$$\mathcal{P} = \mathcal{NP}$$

holds is the same as that of asking whether or not there exist problems in $\mathcal{NP}$ which are intractable.

9.4.4 Justification for the Turing machine model

Question: Why is the Turing machine model used in the development of comparative problem complexity, instead of a higher level model, such as an imperative programming language?

Formulation of nondeterminism: There is a straightforward and "natural" formulation of nondeterminism within this model, which seems less artificial than in other models.

Expression of the actual size of computations: When comparing the complexities of different algorithms operating on distinct problems, it is extremely important not to make assumptions about the length of time required to perform certain operations, which may bias the comparison.

Data size: A similar comment applies to the measurement of the actual size of data.

Example: Consider the representation of and operations on integers.

- The usual assumption in a programming language is that integers are of a fixed size, and basic operations require constant time.
- This assumption is only valid if there is a fixed upper bound on the size of integers used in the input and in the computation.
- This assumption is not universally valid across all algorithms.

Polynomial equivalence to other models: It is possible (although generally tedious) to show that a typical computational model (such as an imperative programming language) may be implemented on a DTM in such a way the each basic (*i.e.*, constant-time) operation in the language runs in polynomial time on the DTM.

9.5 Abstract Problems and Reductions

9.5.1 Abstract problems with length and their encodings

(a) An *abstract problem with length* is a triple $P = (I, \rho, L)$ in which (P, ρ) is an abstract problem and

$$L : I \to \mathbb{N}$$

is a function, called the *length function* for P.

(b) An *encoding scheme* with respect to the alphabet Σ for the abstract problem with length $P = (I, \rho, L)$ is a triple (A, ρ', e) in which (A, ρ') in which

(i) (A, ρ') is a language decision problem; and

(ii) e is a reduction of P to (A, ρ):

$$\begin{tikzcd}
I \arrow[r, "\rho"] \arrow[d, "e"'] & \{0,1\} \\
A \subseteq \Sigma^* \arrow[ur, "\rho'"'] &
\end{tikzcd}$$

(c) The encoding scheme is *polynomial* if there are integers $m, k \geq 0$ with the property that for each $i \in I$,

$$\mathsf{Length}(e(i)) \leq k \cdot L(i)^m$$

9.5.2 Structured strings and reasonable encodings

Problem: The length of an input can be made abnormally long by:

(i) padding with irrelevant junk; and/or

(ii) using a cumbersome encoding scheme (*e.g.*, encode numbers in unary).

Problem: A very clever encoding scheme may be employed to realize "fast" algorithms (*e.g.*, for multiplication algorithms, encode numbers as their prime factors).

- In each case, the formal computational complexity obtained will not be a true representation of the complexity of the problem.
- To obtain uniform results across diverse problems, it is required that problem encodings abide by certain constraints.

(a) The *structured strings* are defined recursively as follows.

(i) Any string of 0's and 1's (possibly preceded by a minus sign) is a structured string which represents the appropriate number in binary.

(ii) If σ is a structured string, so too is $[\sigma]$. This string represents the *name* σ.

(iii) If $\sigma_1, \sigma_2, \ldots, \sigma_n$ are structured strings, then so too is $\langle\sigma_1, \sigma_2, \ldots, \sigma_n\rangle$. This string represents the corresponding n-tuple.

- This is enough to encode most problems of interest.

(b) A *reasonable encoding* of an abstract problem with length $P = (I, \rho, L)$ is a polynomial encoding scheme (A, ρ', e) for P in which $e(i)$ is a structured string for each $i \in I$.

9.5.3 Working convention

- From now on, the theory will be developed with respect to language decision problems.
- It will be assumed that any underlying abstract problem possesses a reasonable encoding, as defined in 9.5.2.

9.6 Tractable Reductions and Their Properties

9.6.1 Computable and tractable reductions

Context:

- Σ is a finite alphabet.
- $P_1 = (I_1, \rho_1), P_2 = (I_2, \rho_2)$ are language problems.
- Σ is a finite alphabet, with $I_1, I_2 \subseteq \Sigma^*$.
- e is a reduction of P_1 to P_2.

$$\begin{tikzcd}
I_1 \arrow[rr, "\rho_1"] \arrow[d, "e"'] & & \{0,1\} \\
I_2 \arrow[urr, "\rho_2"'] & &
\end{tikzcd}$$

(a) e is called a *computable reduction* (resp. a *polynomial reduction* or *tractable reduction*) if there is a computable function (resp. a computable function which is polynomial on I_1)

$$f : \Sigma^* \to \Sigma^*$$

with the property that

$$f(\sigma) = e(\sigma) \qquad \text{for all } \sigma \in I_1.$$

(b) Write

$$P_1 \propto P_2$$

just in case there is a polynomial reduction from P_1 to P_2. In this case, it is also said that P_1 *polynomially reduces* to P_2.

9.6.2 Proposition – preservation of complexity under tractable reduction *Let $P_1 = (I_1, \rho_1)$ and $P_2 = (I_2, \rho_2)$ be language problems, and suppose that $P_1 \propto P_2$.*

$$\begin{tikzcd}
I_1 \arrow[r, "\rho_1"] \arrow[d, "f"'] & \{0,1\} \\
I_2 \arrow[ur, "\rho_2"'] &
\end{tikzcd}$$

(a) *If P_2 is deterministic polynomial, then so too is P_1.*

(b) *If P_2 is nondeterministic polynomial, then so too is P_1.*

PROOF: It suffices to note that the solution of P_1 may be realized by first translating it to P_2 (in deterministic polynomial time), and then to solve the corresponding instance of P_2 (in deterministic or nondeterministic polynomial time, as the case may be). □

9.7 Cook's Theorem – The Existence of $\mathcal{NP}$-Complete Problems

9.7.1 $\mathcal{NP}$-complete problems

(a) A problem $P \in \mathcal{NP}$ is said to be $\mathcal{NP}$-*complete* if, for any $P' \in \mathcal{NP}$,

$$P' \propto P.$$

(b) The class of all $\mathcal{NP}$-complete problems is denoted $\mathcal{NPC}$.

- In effect, the $\mathcal{NP}$-complete problems are the "hardest" problems in $\mathcal{NP}$.
- The seminal theorem of Cook demonstrates the existence of such a problem; namely, the satisfiability of Boolean expressions.
- For all $\mathcal{NP}$-complete problems, the best known *deterministic* algorithms run in worst-case time which is exponential; *i.e.*, $O(q^n)$, with $q > 1$.
- For a few problems, algorithms with q substantially smaller than 2 are known, but even these algorithms are not very efficient.
- If even one of these $\mathcal{NP}$-complete problems could be solved in deterministic polynomial time, then they could all be so solved.

9.7.2 Boolean expressions In that which follows, let $X = \{x_1, x_2, \ldots, x_n\}$ be a finite set of variables.

(a) A *truth assignment* to X is a mapping

$$h : X \to \{0,1\}$$

x_i is said to be *true* for h if $h(x_i) = 1$, and *false* for h if $h(x_i) = 0$.

(b) The *Boolean expressions* over X, denoted $\mathsf{BE}(X)$, are defined as follows.

(i) $X \subseteq \mathsf{BE}(X)$.

(ii) If $\varphi_1, \varphi_2 \in \mathsf{BE}(X)$, then so too are $(\varphi_1 \vee \varphi_2)$ and $(\varphi_1 \wedge \varphi_2)$.

(iii) If $\varphi \in \mathsf{BE}(X)$, then so too is $(\neg\varphi)$.

(c) A truth assignment h is extended to a function

$$\bar{h} : \mathsf{BE}(X) \to \{0,1\}$$

as follows. Let $\varphi \in \mathsf{BE}(X)$.

(i) If $\varphi = x_i \in X$, then $\bar{h}(\varphi) = h(\varphi)$.

(ii) If $\varphi = (\varphi_1 \vee \varphi_2)$ for some $\varphi_1, \varphi_2 \in \mathsf{BE}(X)$, then

$$h(\varphi) = \max(\{\bar{h}(\varphi_1), \bar{h}(\varphi_2)\})$$

(iii) If $\varphi = (\varphi_1 \wedge \varphi_2)$ for some $\varphi_1, \varphi_2 \in \mathsf{BE}(X)$, then

$$h(\varphi) = \min(\{\bar{h}(\varphi_1), \bar{h}(\varphi_2)\})$$

(iv) If $\varphi = (\neg\varphi)$ for some $\varphi \in \mathsf{BE}(X)$, then

$$h(\varphi) = 1 - \bar{h}(\varphi)$$

9.7.3 The problem SAT– satisfiability of Boolean expressions Let $X = \{x_1, x_2, \ldots, x_n\}$ be a finite set of variables, and let $\varphi \in BE(X)$.

(a) The *satisfiability problem* for φ is that of determining whether there exists a truth assignment

$$h : X \to \{0,1\}$$

with the property that $\bar{h}(\varphi) = 1$. Formally, the problem SAT is defined as:

$$\mathsf{SAT} = (\mathsf{BE}(X), \rho_{\mathsf{SAT}})$$

with

$$\rho_{\mathsf{SAT}} : \varphi \mapsto \sup(\{\bar{h}(\varphi) \mid h \text{ is a truth assignment for } \mathsf{BE}(X)\})$$

(b) The *size* of an instance of SAT is the length of the string needed to represent the associated formula.

9.7.4 Theorem –SAT is $\mathcal{NP}$-complete *The problem SAT is in $\mathcal{NPC}$; that is, SAT is $\mathcal{NP}$-complete.*

PROOF: First of all, it will be shown that SAT $\in \mathcal{NP}$. There are two key steps which the deciding NDTM must perform:

(gen) Nondeterministically generate a test truth assignment g.

(test) (Deterministically) test the truth assignment on the input formula φ.

- It is straightforward, but extremely tedious, to describe these steps in detail for a Turing machine.
- Therefore, only a high-level explanation will be provided.
- First of all, it is clear that step (test) may be performed in (deterministic) polynomial time.
- A more detailed description of step (gen) is given on the next slide.

(gen1) Parse the input expression and determine the number of variables. Write this number to the right of the expression, on the tape.

(gen2) If the variable count is zero, go on to the test phase. Otherwise proceed.

(gen3) Move to the right of the tape, to the position where the test sequence will be written.

(gen4) Enter a "generate bit" state. From this state, the next state is fixed. However, there are two possible choices for the value of the tape square, 0 and 1. The next configuration is nondeterministically chosen, so that either a 0 or 1 is written onto the tape.

(gen5) Enter a (deterministic) subroutine which sends the tape head back to the position of the variable count.

(gen6) Decrement the variable count by one and go to (gen2).

- It is straightforward to see that this program for (gen2) may be performed in polynomial time.
- Note that step is (gen4) is inherently nondeterministic; is is not possible on a DTM.
- It is important to recall that for an NDTM to accept, it is sufficient that it halt in an accepting state for *some* sequence of moves.
- Thus, since *every* possible truth assignment is generated by some sequence of moves, it is guaranteed to accept if at least one of these assignments satisfies the formula.

Next, the proof of completeness is presented.

- Let $P = (I, \rho) \in \mathcal{NP}$.
- The strategy is to provide a construction κ which takes an instance $w \in I$, and produces a Boolean expression φ_w which is satisfiable iff $\rho(w) = 1$.
- It must furthermore be shown that this construction may be performed in (deterministic) polynomial time.
- The diagram below illustrates the main entities in the reduction.

$$\begin{tikzcd} I \arrow[rr, "\rho"] \arrow[d, "w"'] & & \{0,1\} \\ \mathsf{BE}(X) \arrow[urr, "\rho_{\mathsf{SAT}}"'] & & \end{tikzcd}$$

- X is a an appropriate set of variables; its size is dependent upon I but not upon any particular instance of I.

- Let $M = (Q, \Sigma, \Gamma, \#, \delta, q_0, F)$ be an NDTM which solves P in polynomial time.
- It is useful to modify M, to yield $M' = (Q', \Sigma, \Gamma, \#, \delta', q_0, F')$, as follows.
    - The new machine M' has only one accepting state, which will be denoted by q_a.
    - To accomplish this, define $F' = \{q_a\}$, and for each q in the F of the old machine, and each $x \in \Gamma$ with $\delta(q,x) = \emptyset$, define

$$\delta'(q,x) = \{(q_a, x, N)\}$$

    - Note that q_a can only be entered if the old machine would have *halted* in an accepting state.
    - Also, instead of halting in an accepting state, M' loops forever, leaving the state, head position, and tape unchanged.
    - To accomplish this, for each $x \in \Gamma$, define

$$\delta'(q_a,x) = \{(q_a, x, N)\}$$

    - Under this convention, it is necessary to redefine halting as looping endlessly in state q_a.
    - This does not contradict the unsolvability of the halting problem; looping in the same configuration is trivial to detect.
    - Note also that the complexity of operations on M' is the same as that of M, up to the insignificant addition of a small constant.

- Now, let p be a polynomial which bounds the complexity of M; *i.e.*, for $w \in I$ and D a computation of M,

$$\mathsf{Length}(D) \leq p(\mathsf{Length}(w))$$

- If M' is used instead of M, then D may be taken to be an infinite computation, and the test condition for acceptance becomes that of asking whether or not

$$D_{p(n)} = (?,?,?,q_a)$$

Here $D_{p(n)}$ is the $p(n)+1^{st}$ element of D, and $(?,?,?,x)$ represents any configuration of M' for which the state is q_a.

- A Boolean expression, depending upon the input w, will now be constructed.
- This expression will have the property that it is satisfiable iff M' accepts w.
- For convenience, assume that the following naming conventions are followed:
  - (i) The tape cells are numbered $1, 2, 3, \ldots$, beginning with the leftmost.
  - (ii) The states are named $q_1, q_2, \ldots, q_s$, with s representing the accepting state.
  - (iii) Time is measured in unit steps, beginning with 0.
- To begin, three families of propositions are introduced.

$$\begin{aligned} C(i, j, t) = 1 &\Leftrightarrow \text{tape cell } i \text{ contains symbol } j \text{ at time } t. \\ S(k, t) = 1 &\Leftrightarrow M' \text{ is in state } q_k \text{ at time } t. \\ H(i, t) = 1 &\Leftrightarrow \text{the tape head is scanning cell } i \text{ at time } t. \end{aligned}$$

- It is important to keep in mind that these are regarded as propositions, and not as first-order predicates which take arguments.
- The following constraints exist on the parameters.
  - $1 \leq i \leq p(\mathsf{Length}(w))$ (The size is bounded by the time used.)
  - $1 \leq j \leq m = \mathsf{Card}(\Gamma)$ (the number of tape symbols.)
  - $1 \leq k \leq s = \mathsf{Card}(Q')$ (the number of states)
  - $0 \leq t \leq p(\mathsf{Length}(w))$

- There are:
  - $\Theta(p(\mathsf{Length}(w))^2)$ propositions of the form $C(i, j, t)$.
  - $\Theta(p(\mathsf{Length}(w)))$ propositions of the form $S(k, t)$.
  - $\Theta(p(\mathsf{Length}(w))^2)$ propositions of the form $H(i, t)$.
- Therefore, there are $\Theta(p(\mathsf{Length}(w))^2)$ propositions total.
- Next, a Boolean expression on r propositional variables is introduced, which asserts that exactly one of the variables is true:

$$U(x_1, \ldots, x_r) \overset{\text{def}}{=} \left(\bigvee_{i=1}^{r} x_i \right) \wedge \left(\bigwedge_{1 \leq i < j \leq r} (\neg(x_i \wedge x_j)) \right)$$

- Observe that $U(x_1, \ldots, x_r)$ may be written using at most $k' \cdot r^2$ symbols, for some constant k'.
- Using the families $U(-)$, $C(-)$, $S(-)$, and $H(-)$, a Boolean expression

$$X_1 \wedge X_2 \wedge X_3 \wedge X_4 \wedge X_5 \wedge X_6 \wedge X_7$$

  will be constructed which is true iff M accepts w.
- It will furthermore be shown that each X_i may be constructed in (deterministic) polynomial time.

1. X_1 asserts that, at each point in time, the tape head is scanning exactly one cell.

$$\begin{aligned} X_1^t &\overset{\text{def}}{=} U(H(1,t),H(2,t),\ldots,H(p(\mathsf{Length}(w)),t)) \\ X_1 &\overset{\text{def}}{=} \bigwedge_{t=0}^{p(\mathsf{Length}(w))} X_1^t \end{aligned}$$

- $\mathsf{Length}(X_1) \in \Theta(p(\mathsf{Length}(w))^3)$

2. X_2 asserts that, at each point in time, each tape cell contains exactly one symbol. (Only the cells from 1 through $p(\mathsf{Length}(w))$ are considered.

$$\begin{aligned} X_2^{i,t} &\overset{\text{def}}{=} U(C(i,1,t),C(i,2,t),\ldots,C(i,m,t)) \\ X_2 &\overset{\text{def}}{=} \bigwedge_{\substack{1\leq i\leq p(\mathsf{Length}(w)) \\ 1\leq t\leq p(\mathsf{Length}(w))}} X_2^{i,t} \end{aligned}$$

- $\mathsf{Length}(X_2) \in \Theta(p(\mathsf{Length}(w))^2)$

3. X_3 asserts that, at each point in time, M' is in exactly one state.

$$\begin{aligned} X_3^t &\overset{\text{def}}{=} U(S(1,t),S(2,t),\ldots,S(s,t)) \\ X_3 &\overset{\text{def}}{=} \bigwedge_{t=0}^{p(\mathsf{Length}(w))} X_3^t \end{aligned}$$

- $\mathsf{Length}(X_3) \in \Theta(p(\mathsf{Length}(w)))$

4. X_4 asserts that, at each point in time, at most one tape cell can change value, and that cell must be the one which is currently being scanned.

$$X_4^{i,j,t} \overset{\text{def}}{=} ((C(i,j,t+1) \Leftrightarrow C(i,j,t)) \vee H(i,t))$$

$$X_4 \overset{\text{def}}{=} \bigwedge_{\substack{1 \le i \le p(\mathsf{Length}(w)) \\ 1 \le j \le \mathsf{Card}(\Gamma) \\ 1 \le t \le p(\mathsf{Length}(w))}} X_4^{i,j,t}$$

- $\mathsf{Length}(X_4) \in \Theta(p(\mathsf{Length}(w))^2)$

5. X_5 asserts that, at each point in time, the new ID of M' is obtained from the old one by an application of the state-transition function δ'.

$$Y_5^{i,j,t} \overset{\text{def}}{=} \bigvee_{\substack{(k',j',\xi) \in \delta'(k,j) \\ (\xi=-1 \Rightarrow L;\ \xi=0 \Rightarrow N;\ \xi=1 \Rightarrow R)}} (C(i,j',t+1) \wedge S(k',t+1) \wedge H(i+\xi,t+1))$$

$$X_5^{i.j,t} \overset{\text{def}}{=} \underset{\substack{\uparrow \\ \text{legal} \\ \text{moves}}}{Y_5^{i,j,t}} \vee \underset{\substack{\uparrow \\ \text{wrong} \\ \text{tape symbol} \\ \text{for transition}}}{(\neg C(i,j,t))} \vee \underset{\substack{\uparrow \\ \text{wrong} \\ \text{tape cell} \\ \text{for transition}}}{(\neg H(i,t))} \vee \underset{\substack{\uparrow \\ \text{wrong} \\ \text{state} \\ \text{for transition}}}{(\neg S(k,t))}$$

$$X_5 \overset{\text{def}}{=} \bigwedge_{\substack{1 \le i \le p(\mathsf{Length}(w)) \\ 1 \le j \le \mathsf{Card}(\Gamma) \\ 1 \le k \le \mathsf{Card}(Q') \\ 1 \le t \le p(\mathsf{Length}(w))}} X_4^{i,j,t}$$

- $\mathsf{Length}(X_5) \in \Theta(p(\mathsf{Length}(w))^2)$

6. X_6 asserts that the initial ID is correct.

$$X_6 \overset{\text{def}}{=} \underset{\substack{\uparrow \\ \text{initial} \\ \text{state}}}{S(1,0)} \wedge \underset{\substack{\uparrow \\ \text{initial} \\ \text{tape} \\ \text{square}}}{H(1,0)} \wedge \left(\bigwedge_{i=1}^{\mathsf{Length}(w)} \underset{\substack{\uparrow \\ w \text{ on tape} \\ (w = w_1, w_2, \ldots, w_{\mathsf{Length}(w)})}}{C(i, w_i, 0)} \right) \wedge \left(\bigwedge_{i=\mathsf{Length}(w)+1}^{p(\mathsf{Length}(w))} \underset{\substack{\uparrow \\ \text{pad} \\ \text{with blanks}}}{C(i, \#, 0)} \right)$$

- $\mathsf{Length}(X_6) \in \Theta(p(\mathsf{Length}(w)))$

7. X_7 states that M' is in state q_s at time $p(\mathsf{Length}(w))$.

$$X_7 = S(s, p(\mathsf{Length}(w)))$$

- $\mathsf{Length}(X_7) \in \Theta(\log(\mathsf{Length}(w)))$.
- Thus,
$$X = X_1 \wedge X_2 \wedge X_3 \wedge X_4 \wedge X_5 \wedge X_6 \wedge X_7$$
can be generated and written in time $O(p(\mathsf{Length}(w))^3)$.
- It follows directly from the construction that X is satisfiable iff M' accepts w.
- The question of whether or not M' accepts w can therefore be solved by first computing X, and then using the SAT routine to test it for satisfiability.
- Thus,
$$P \propto \mathsf{SAT}$$

□

9.8 The Complexity of Other Satisfiability Problems

9.8.1 CNF-SAT Let $X = \{x_1, x_2, \ldots, x_n\}$ be a finite set of variables.

(a) A *literal* is either $x \in X$ or else a negation of a variable; *e.g.*, $(\neg x)$.

- (b) A *clause* is a disjunction of literals; *e.g.*,

$$(\ell_1 \vee \ell_2 \vee \ell_3 \vee \ldots \vee \ell_n)$$

- (c) A Boolean expression is in *conjunctive normal form* (*CNF*) if it is a conjunction of clauses.
- Example of a clause:

$$(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee x_5) \wedge (x_6 \vee x_1) \wedge x_7$$

(b) The decision problem CNF-SAT takes as input a Boolean expression φ in CNF and returns 1 if φ is satisfiable, and 0 otherwise.

- Clearly, CNF-SAT is a special case of SAT.

9.8.2 Proposition – CNF-SAT is $\mathcal{NP}$-complete *The problem CNF-SAT is $\mathcal{NP}$-complete.*

PROOF: The proof makes use of the constructions in the proof of Cook's theorem (9.7.4). The expressions X_1, X_2, X_3, X_6, and X_7 are already in CNF. Note that $U(x_1, \ldots, x_r)$ may be rewritten as

$$U(x_1, \ldots, x_r) \overset{\text{def}}{=} \left(\bigvee_{i=1}^{r} x_i \right) \wedge \left(\bigwedge_{1 \leq i < j \leq r} ((\neg x_i) \vee (\neg x_j)) \right)$$

It remains to show that X_4 and X_5 may be transformed into CNF in polynomial time. First, X_4 consists of conjuncts of the form

$$((\varphi_1 \Leftrightarrow \varphi_2) \vee \varphi_3)$$

which is equivalent to

$$(\varphi_1 \vee (\neg \varphi_2) \vee \varphi_3) \wedge ((\neg \varphi_1) \vee \varphi_2 \vee \varphi_3)$$

This translation may be performed in constant time. Each $X_5^{i,j,k,t}$ in X_5 may be transformed to CNF in linear time as well. Thus, X_5 may be so transformed as well, in time $O(\mathsf{Length}(X_5)) = O(p(\mathsf{Length}(w))^2)$. Hence, the entire transformation may be performed in polynomial time. Since the problem is a subproblem of SAT, it is clearly in $\mathcal{NP}$ which completes the proof. □

- It is important not to assume that *every* normal form for formulas defines a class which is $\mathcal{NP}$-complete.
- The following example illustrates this point.

9.8.3 Disjunctive normal form Let $X = \{x_1, x_2, \ldots, x_n\}$ be a finite set of variables.

(a) A *co-clause* is a conjunction of literals; that is, a formula of the form

$$(\ell_1 \wedge \ell_2 \wedge \ell_3 \wedge \ldots \wedge \ell_n)$$

(b) A formula is in *disjunctive normal form DNF* if it is a disjunction of co-clauses.

- Shown below is a formula in DNF.

$$(x_1 \wedge x_2 \wedge x_3) \vee (x_2 \wedge x_5) \vee (x_6 \wedge x_1) \vee x_7$$

(b) The decision problem DNF-SAT takes as input a Boolean expression φ in CNF and returns 1 if φ is satisfiable, and 0 otherwise.

- Clearly, DNF-SAT is a special case of SAT.

9.8.4 Proposition – the complexity of DNF-SAT

DNF-SAT is solvable in time linear in the size of the formula.

PROOF: Let

$$\varphi = (\ell_{11} \wedge \ldots \wedge \ell_{1n_1}) \vee (\ell_{21} \wedge \ldots \wedge \ell_{2n_2}) \vee \ldots \vee (\ell_{m1} \wedge \ldots \wedge \ell_{mn_m})$$

be a formula in DNF. Note that the formula is satisfiable iff at least one of its co-clauses is. The algorithm simply searches each co-clause for complementary literals; *i.e.*, pairs of the form $\{x_1, (\neg x_i)\}$. A co-clause is satisfiable iff it does not contain such a pair. Clearly, such a search may be conducted in linear time, with one sweep across the formula. □

9.8.5 Remarks

- The above proposition shows that DNF-SAT cannot be $\mathcal{NP}$-complete unless $\mathcal{P}$=$\mathcal{NP}$.
- It also establishes that the transformation of a formula from CNF to DNF cannot be performed in polynomial time, unless $\mathcal{P}$=$\mathcal{NP}$.
- In fact, it is possible to find formulas which are in CNF with the property that any corresponding formula in DNF is exponentially larger in size.
- Thus, even if $\mathcal{P}$=$\mathcal{NP}$, the transformation from CNF to DNF must be $\Theta(2^n)$ in the worst case.

9.8.6 Size-limited CNF Let $X = \{x_1, x_2, \dots, x_n\}$ be a finite set of variables, and let k be a positive integer.

(a) A Boolean expression φ over X is said to be in k-CNF if it is in CNF and each clause of φ contains at most k literals.

(b) The problem k-SAT is that of determining whether a given formula φ which is in k-CNF-SAT is satisfiable.

9.8.7 Proposition *The problem 3-SAT is $\mathcal{NP}$-complete.*

PROOF:

- The approach is to show how to replace a clause with more than three literals with a conjunction of clauses with exactly three literals, in polynomial time.
- Let
$$\varphi = (\ell_1 \vee \ell_2 \vee \dots \vee \ell_k)$$
be a clause with $k \geq 4$.
- Introduce $k-3$ new variables $y_1, \dots, y_{k-3}$, and replace φ by the following conjunction.

$$\begin{aligned}
\varphi' &= (\ell_1 \vee \ell_2 \vee y_1) \\
&\wedge (\ell_3 \vee (\neg y_1) \vee y_2) \\
&\wedge (\ell_4 \vee (\neg y_2) \vee y_3) \\
&\wedge (\ell_5 \vee (\neg y_3) \vee y_4) \\
&\quad \vdots \\
&\wedge (\ell_{k-3} \vee (\neg y_{k-5}) \vee y_{k-4}) \\
&\wedge (\ell_{k-2} \vee (\neg y_{k-4}) \vee y_{k-3}) \\
&\wedge (\ell_{k-1} \vee \ell_k \vee (\neg y_{k-3}))
\end{aligned}$$

- Let f be a truth assignment on the variables of the original clause φ with $\bar{f}(\varphi) = 1$. Then $\bar{f}(\ell_i)$ must be true for some ℓ_i.
- Extend f by
$$f(y_j) = \begin{cases} 1 & \text{if } j \leq i-2 \\ 0 & \text{if } j > i-2 \end{cases}$$
- It is easily verified that $\bar{f}(\varphi') = 1$.
- On the other hand, let f be a truth assignment on the variables of φ with the property that $f(\varphi) = 0$. Then $f(\ell_i) = 0$ for all i.
- The claim is that there is no extension of f to the variables $\{y_1, \ldots, y_{k-3}\}$ with $\bar{f}(\varphi') = 1$.
- If g were such an extension, then
$$\bar{g}(\ell_1 \vee \ell_2 \vee y_1) = 1 \quad \Rightarrow \quad g(y_1) = 1$$
- Then
$$\bar{g}(\ell_2 \vee (\neg y_1) \vee y_2) = 1 \quad \Rightarrow \quad g(y_2) = 1$$
- Continuing on in this fashion, it is easily established that $g(y_i) = 1$ for each y_i.
- The last clause
$$(\ell_{k-1} \vee \ell_k \vee (\neg y_{k-3}))$$
cannot then be satisfied by $\bar{g}$, since all three of its literals are false.
- Thus, φ' is satisfiable iff φ is.
- This transformation may be performed in linear time, and applied to each clause in a CNF formula.
- Thus, CNF-SAT may be reduced 3-CNF-SAT in linear time, whence the result. $\square$

9.8.8 Remarks

- Note that in the previous proof, the problem which was known to be $\mathcal{NP}$-complete (CNF-SAT) was reduced to the problem which was to be shown to be $\mathcal{NP}$-complete (3-CNF-SAT).
- This is always the reduction which is need to show $\mathcal{NP}$-completeness.
- Consider once again the following reduction diagram.

$$\begin{tikzcd} I_1 \arrow[rr, "\rho_1"] \arrow[d, "f"'] & & \{0,1\} \\ I_2 \arrow[urr, "\rho_2"'] & & \end{tikzcd}$$

- The diagram states that, up to deterministic polynomial equivalence, $P_2 = (I_2, \rho_2)$ is at least a difficult as $P_1 = (I_1, \rho_1)$, since P_2 may be used as a subroutine to solve P_1.
- Relative to the above example:

$$\begin{aligned} P_1 &= \text{CNF-SAT} \\ P_2 &= \text{3-CNF-SAT} \end{aligned}$$

- It is also worth noting that 2-CNF-SAT is solvable in deterministic polynomial time, but that not all Boolean expressions have a representation in 2-CNF.
- It also bears repeating that the best known algorithms for all of these $\mathcal{NP}$-complete problems run in worst-case time $\Omega_2(k^n)$ for some $k > 1$.

9.9 Other $\mathcal{NP}$-complete Problems

- In this section, a number of other problems which are known to be $\mathcal{NP}$-complete are presented.
- This selection is far from complete; there are thousands of problems known to be $\mathcal{NP}$-complete.
- The proofs that these problems are $\mathcal{NP}$-complete will not be given.

9.9.1 The discrete knapsack decision problem

- This problem is the decision version of the discrete knapsack problem.

The setting:

- A knapsack with weight capacity M.
- n objects $\{\mathsf{obj}_1, \mathsf{obj}_2, \ldots, \mathsf{obj}_n\}$, each with a weight w_i and a value v_i.
- M, the w_i's, and the v_i's may be taken to be positive integers.
- A *target profit P* is also given.

The problem:

- Determine whether or not there is a vector $(x_1, x_2, \ldots, x_n) \in \{0,1\}^N$ such that:
  (a) $\sum_{i=1}^{n} x_i \cdot \mathsf{v}_i \geq P$, subject to the constraint that
  (b) $\sum_{i=1}^{n} x_i \cdot \mathsf{w}_i \leq M$.

9.9.2 The travelling salesman decision problem

- This problem is the decision version of the travelling salesman problem.

The setting:

- A directed graph $G = (V, E, g)$, together with a weighting function $d : E \to \mathbb{N}$.
- A *target tour cost* C.

The problem: Determine whether or not there is a tour whose cost is no greater than C.

9.9.3 The coin changing decision problem

Given:

- A finite sequence $(c_1, c_2, \ldots, c_n)$ of positive integers, with $c_i = 1$ and $c_i \leq c_{i+1}$ for all i.
- A target change amount $C > 0$.
- A maximum number of coins K.

The problem: Determine whether or not there is a vector $(x_1, x_2, \ldots, x_n) \in \mathbb{N}^N$ such that:

(a) $\sum_{i=1}^{n} x_i \cdot c_i = C$, subject to the constraint that

(b) $\sum_{i=1}^{n} x_i \leq K$.

9.9.4 The clique decision problem

Given:

- An undirected graph $G = (V, E, g)$
- A positive integer $K \leq \mathsf{Card}(V)$.

The problem: Determine whether G has a clique with exactly K nodes.

- A *clique* of G is a subgraph in which there is an edge between every pair of distinct nodes.

9.9.5 The Hamiltonian circuit decision problem

Given:

- An undirected graph $G = (V, E, g)$

The problem: Determine whether G has a Hamiltonian circuit.

- A *Hamiltonian circuit* of G is a cycle which passes through each vertex exactly once.
- Note that a tour (in the travelling salesman sense) is a Hamiltonian circuit.

9.9.6 The partition problem

Given:

- n objects $\{\mathsf{obj}_1, \mathsf{obj}_2, \ldots, \mathsf{obj}_n\}$, each with a weight w_i.

The problem: Determine whether there is a partition {A,B} of $\{\mathsf{obj}_1, \mathsf{obj}_2, \ldots, \mathsf{obj}_n\}$ such that

$$\sum_{x \in A} \mathsf{w}_x = \sum_{x \in B} \mathsf{w}_x$$

- This problem may be viewed as one of placing weights on a balance scale such that the two sides balance exactly.

9.9.7 *n*-dimensional matching

Given:

- n disjoint sets $X_1, X_2, \ldots X_n$, each of the same cardinality c.
- A subset $A \subseteq X_1 \times X_2 \times \ldots \times \ldots X_n$.

The problem: Determine whether there is a subset $B \subseteq A$ with the property that each element of each X_i occurs in exactly one tuple of B.

- Note that B must also be of cardinality c.
- n dimensional matching is $\mathcal{NP}$-complete for $n \geq 3$, but not for $n = 2$.

9.9.8 A remark on two-dimensional matching

- The two-dimensional matching problem is sometimes called the *marriage problem*.
- Let
  - (i) $X_1 =$ set of males;
  - (ii) $X_2 =$ set of females; with
  - (iii) $\mathsf{Card}(X_1) = \mathsf{Card}(X_2)$.
  - (iv) $A \subseteq X_1 \times X_2$.
- Think of A as representing “acceptable” pairings.
- A match B (in the sense of 9.9.7 above), represents a pairing of each individual with an acceptable mate.
- This problem is solvable in deterministic polynomial time; it is not $\mathcal{NP}$-complete.

9.10 Complements of problems

- There is an inherent asymmetry in the nature of acceptance for an NDTM, as described in 9.2.5.
- This asymmetry has substantial implications for the nature of $\mathcal{NP}$-problems.

9.10.1 The complement of a problem

- Let $P = (I, \rho)$ be a decision problem.

(a) The problem *complementary to P* is

$$P^c = (I, 1 - \rho)$$

- Here
$$1 - \rho : x \mapsto 1 - \rho(x)$$
- In working with discussions of solutions using nondeterministic algorithms, it is very important to distinguish between the solution of a problem and the solution of its complement.
- This will be illustrated via several examples.

9.10.2 UnSAT– The complement of SAT Let $X = \{x_1, x_2, \ldots, x_n\}$ be a finite set of variables, and let $\varphi \in BE(X)$.

(a) The *unsatisfiability problem* for φ is that of determining whether there is no truth assignment

$$h : X \to \{0,1\}$$

with the property that $\bar{h}(\varphi) = 1$. Formally, the problem UnSAT is defined as:

$$\mathsf{UnSAT} = (\mathsf{BE}(X), \rho_{\mathsf{UnSAT}})$$

with

$$\rho_{\mathsf{UnSAT}} : \varphi \mapsto \inf(\{\bar{h}(\varphi) \mid h \text{ is a truth assignment for } \mathsf{BE}(X)\})$$

(b) The *size* of an instance of SAT is the size of the underlying set X of variables.

- Note that with respect to nondeterministic solutions, this problem appears to have a very different flavor than does SAT.
- With SAT, a nondeterministic algorithm can nondeterministically generate a test solution and then test it in linear time.
- With UnSAT, the nondeterminism would not appear to be of any help, since *all* truth assignment must be false. There is no notion of a verifier which can test a proposed solution.
- It would thus appear that UnSAT is strictly more difficult than SAT.
- This question will now be examined more closely in a short while.
- First, however, one more complementary problem will be presented.

9.10.3 The complement of the discrete knapsack decision problem

The setting:

- A knapsack with weight capacity M.
- n objects $\{\mathsf{obj}_1, \mathsf{obj}_2, \ldots, \mathsf{obj}_n\}$, each with a weight w_i and a value v_i.
- M, the w_i's, and the v_i's may be taken to be positive integers.
- A *target profit* P is also given.

The problem:

- Determine whether no vector $(x_1, x_2, \ldots, x_n) \in \{0,1\}^N$ has the properties that:
  - (a) $\sum_{i=1}^{n} x_i \cdot \mathsf{v}_i \geq P$, and
  - (b) $\sum_{i=1}^{n} x_i \cdot \mathsf{w}_i \leq M$.

9.10.4 Proposition *Let $P \in \mathcal{P}$. Then $P^c \in \mathcal{P}$ as well.*

PROOF: This is immediate; if $M = (Q, \Sigma, \Gamma, \#, \delta, q_0, F)$ is an accepting machine for P, then $M = (Q, \Sigma, \Gamma, \#, \delta, q_0, Q \setminus F)$ is an accepting machine for P^c. □

9.10.5 Theorem

- *If there is a decision problem P which has the following two properties:*

  (i) *P is $\mathcal{NP}$-complete; and*

  (ii) *$P^c \in \mathcal{NP}$;*

  then for every $Q \in \mathcal{NP}$, $Q^c \in \mathcal{NP}$ as well.

- *In other words, if the complement of some $\mathcal{NP}$-complete problem is in $\mathcal{NP}$, then the complement of every problem in $\mathcal{NP}$ is also in $\mathcal{NP}$.*

PROOF: Let:

$$\begin{array}{rcccl} P &=& (I_1, \rho_1) &\in& \mathcal{NPC} \\ P^C &=& (I_1, 1-\rho_1) &\in& \mathcal{NPC} \\ Q &=& (I_2, \rho_2) &\in& \mathcal{NPC} \end{array}$$

Then there is a function $f : I_1 \to I_2$, which is computable in polynomial time, such that the following diagram commutes.

$$\begin{tikzcd} I_2 \arrow[r, "\rho_2"] \arrow[d, "f"'] & \{0,1\} \\ I_1 \arrow[ur, "\rho_1"] & \end{tikzcd}$$

Since ρ_1 and ρ_2 are total functions, this immediately implies that the following diagram also commutes.

$$\begin{tikzcd}
I_2 \arrow[r, "1-\rho_2"] \arrow[d, "f"'] & \{0,1\} \\
I_1 \arrow[ur, "1-\rho_1"'] &
\end{tikzcd}$$

Then, since $P^C \in \mathcal{NP}$, so too is Q^c. □

9.10.6 Definition

$$\text{co-}\mathcal{NP} \stackrel{\text{def}}{=} \{P^c \mid P \in \mathcal{NP}\}$$

9.10.7 Theorem (restatement)

$$\mathcal{NPC} \cap co\text{-}\mathcal{NP} \neq \emptyset \quad \Longrightarrow \quad \mathcal{NP} = co\text{-}\mathcal{NP} \ \square$$

9.10.8 Corollary

$$\mathcal{P} = \mathcal{NP} \quad \Longrightarrow \quad \mathcal{NP} = co\text{-}\mathcal{NP} \ \square$$