Introduction to the Theory of Computation
Some Notes for CIS262

Jean Gallier
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104, USA
e-mail: jean@cis.upenn.edu

© Jean Gallier
Please, do not reproduce **without permission** of the author

December 26, 2017

Contents

1 Introduction **7**

2 Basics of Formal Language Theory **9**
- 2.1 Alphabets, Strings, Languages 9
- 2.2 Operations on Languages 15

3 DFA's, NFA's, Regular Languages **19**
- 3.1 Deterministic Finite Automata (DFA's) 20
- 3.2 The "Cross-product" Construction 25
- 3.3 Nondeteterministic Finite Automata (NFA's) 27
- 3.4 ϵ-Closure 30
- 3.5 Converting an NFA into a DFA 32
- 3.6 Finite State Automata With Output: Transducers 36
- 3.7 An Application of NFA's: Text Search 40

4 Hidden Markov Models (HMMs) **45**
- 4.1 Hidden Markov Models (HMMs) 45
- 4.2 The Viterbi Algorithm and the Forward Algorithm 58

5 Regular Languages, Minimization of DFA's **67**
- 5.1 Directed Graphs and Paths 67
- 5.2 Labeled Graphs and Automata 70
- 5.3 The Closure Definition of the Regular Languages 72
- 5.4 Regular Expressions 75
- 5.5 Regular Expressions and Regular Languages 76
- 5.6 Regular Expressions and NFA's 78
- 5.7 Applications of Regular Expressions 86
- 5.8 Summary of Closure Properties of the Regular Languages 87
- 5.9 Right-Invariant Equivalence Relations on Σ^* 88
- 5.10 Finding minimal DFA's 98
- 5.11 State Equivalence and Minimal DFA's 100
- 5.12 The Pumping Lemma 111
- 5.13 A Fast Algorithm for Checking State Equivalence 115

6 Context-Free Grammars And Languages 127
6.1 Context-Free Grammars 127
6.2 Derivations and Context-Free Languages 128
6.3 Normal Forms for Context-Free Grammars 134
6.4 Regular Languages are Context-Free 141
6.5 Useless Productions in Context-Free Grammars 142
6.6 The Greibach Normal Form 144
6.7 Least Fixed-Points 145
6.8 Context-Free Languages as Least Fixed-Points 147
6.9 Least Fixed-Points and the Greibach Normal Form 151
6.10 Tree Domains and Gorn Trees 156
6.11 Derivations Trees 160
6.12 Ogden's Lemma 162
6.13 Pushdown Automata 168
6.14 From Context-Free Grammars To PDA's 172
6.15 From PDA's To Context-Free Grammars 173
6.16 The Chomsky-Schutzenberger Theorem 175

7 A Survey of LR-Parsing Methods 177
7.1 $LR(0)$-Characteristic Automata 177
7.2 Shift/Reduce Parsers 186
7.3 Computation of FIRST 188
7.4 The Intuition Behind the Shift/Reduce Algorithm 189
7.5 The Graph Method for Computing Fixed Points 190
7.6 Computation of FOLLOW 192
7.7 Algorithm $Traverse$ 193
7.8 More on $LR(0)$-Characteristic Automata 195
7.9 LALR(1)-Lookahead Sets 195
7.10 Computing FIRST, FOLLOW, etc. in the Presence of ϵ-Rules 197
7.11 $LR(1)$-Characteristic Automata 204

8 RAM Programs, Turing Machines 209
8.1 Partial Functions and RAM Programs 209
8.2 Definition of a Turing Machine 215
8.3 Computations of Turing Machines 217
8.4 RAM-computable functions are Turing-computable 220
8.5 Turing-computable functions are RAM-computable 221
8.6 Computably Enumerable and Computable Languages 222
8.7 The Primitive Recursive Functions 223
8.8 The Partial Computable Functions 229

9 Universal RAM Programs and the Halting Problem 235
9.1 Pairing Functions 235

9.2 Equivalence of Alphabets . . . 238
9.3 Coding of RAM Programs . . . 242
9.4 Kleene's T-Predicate . . . 250
9.5 A Simple Function Not Known to be Computable . . . 252
9.6 A Non-Computable Function; Busy Beavers . . . 254

10 Elementary Recursive Function Theory 259
10.1 Acceptable Indexings . . . 259
10.2 Undecidable Problems . . . 262
10.3 Listable (Recursively Enumerable) Sets . . . 267
10.4 Reducibility and Complete Sets . . . 272
10.5 The Recursion Theorem . . . 276
10.6 Extended Rice Theorem . . . 280
10.7 Creative and Productive Sets . . . 283

11 Listable and Diophantine Sets; Hilbert's Tenth 287
11.1 Diophantine Equations; Hilbert's Tenth Problem . . . 287
11.2 Diophantine Sets and Listable Sets . . . 290
11.3 Some Applications of the DPRM Theorem . . . 294

12 The Post Correspondence Problem; Applications 299
12.1 The Post Correspondence Problem . . . 299
12.2 Some Undecidability Results for CFG's . . . 300
12.3 More Undecidable Properties of Languages . . . 303

13 Computational Complexity; $\mathcal{P}$ and $\mathcal{NP}$ 305
13.1 The Class $\mathcal{P}$ . . . 305
13.2 Directed Graphs, Paths . . . 307
13.3 Eulerian Cycles . . . 308
13.4 Hamiltonian Cycles . . . 309
13.5 Propositional Logic and Satisfiability . . . 310
13.6 The Class $\mathcal{NP}$, $\mathcal{NP}$-Completeness . . . 314
13.7 The Cook-Levin Theorem . . . 319

14 Some $\mathcal{NP}$-Complete Problems 331
14.1 Statements of the Problems . . . 331
14.2 Proofs of $\mathcal{NP}$-Completeness . . . 342
14.3 Succinct Certificates, co$\mathcal{NP}$, and $\mathcal{EXP}$ . . . 355

15 Primality Testing is in $\mathcal{NP}$ 361
15.1 Prime Numbers and Composite Numbers . . . 361
15.2 Methods for Primality Testing . . . 362
15.3 Modular Arithmetic, the Groups $\mathbb{Z}/n\mathbb{Z}$, $(\mathbb{Z}/n\mathbb{Z})^*$ . . . 365

15.4 The Lucas Theorem; Lucas Trees 374
15.5 Algorithms for Computing Powers Modulo m 379
15.6 PRIMES is in $\mathcal{NP}$ 381

Chapter 1

Introduction

The theory of computation is concerned with algorithms and algorithmic systems: their design and representation, their completeness, and their complexity.

The purpose of these notes is to introduce some of the basic notions of the theory of computation, including concepts from formal languages and automata theory, the theory of computability, some basics of recursive function theory, and an introduction to complexity theory. Other topics such as correctness of programs will not be treated here (there just isn't enough time!).

The notes are divided into three parts. The first part is devoted to formal languages and automata. The second part deals with models of computation, recursive functions, and undecidability. The third part deals with computational complexity, in particular the classes $\mathcal{P}$ and $\mathcal{NP}$.

Chapter 2

Basics of Formal Language Theory

2.1 Alphabets, Strings, Languages

Our view of languages is that *a language is a set of strings*. In turn, a string is a finite sequence of letters from some alphabet. These concepts are defined rigorously as follows.

Definition 2.1. An *alphabet* Σ is any **finite** set.

We often write $\Sigma = \{a_1, \ldots, a_k\}$. The a_i are called the *symbols* of the alphabet.

Examples:

$\Sigma = \{a\}$

$\Sigma = \{a, b, c\}$

$\Sigma = \{0, 1\}$

$\Sigma = \{\alpha, \beta, \gamma, \delta, \epsilon, \lambda, \varphi, \psi, \omega, \mu, \nu, \rho, \sigma, \eta, \xi, \zeta\}$

A string is a finite sequence of symbols. Technically, it is convenient to define strings as functions. For any integer $n \geq 1$, let

$$[n] = \{1, 2, \ldots, n\},$$

and for $n = 0$, let

$$[0] = \emptyset.$$

Definition 2.2. Given an alphabet Σ, a *string over Σ (or simply a string) of length n* is any function

$$u \colon [n] \to \Sigma.$$

The integer n is the *length* of the string u, and it is denoted as $|u|$. When $n = 0$, the special string
$u \colon [0] \to \Sigma$ of length 0 is called the *empty string, or null string*, and is denoted as ϵ.

Given a string $u\colon [n] \to \Sigma$ of length $n \geq 1$, $u(i)$ is the i-th letter in the string u. For simplicity of notation, we denote the string u as

$$u = u_1 u_2 \dots u_n,$$

with each $u_i \in \Sigma$.

For example, if $\Sigma = \{a, b\}$ and $u\colon [3] \to \Sigma$ is defined such that $u(1) = a$, $u(2) = b$, and $u(3) = a$, we write

$$u = aba.$$

Other examples of strings are

$$work, \ \ fun, \ \ gabuzomeuh$$

Strings of length 1 are functions $u\colon [1] \to \Sigma$ simply picking some element $u(1) = a_i$ in Σ. Thus, we will identify every symbol $a_i \in \Sigma$ with the corresponding string of length 1.

The set of all strings over an alphabet Σ, including the empty string, is denoted as Σ^*.

Observe that when $\Sigma = \emptyset$, then

$$\emptyset^* = \{\epsilon\}.$$

When $\Sigma \neq \emptyset$, the set Σ^* is countably infinite. Later on, we will see ways of ordering and enumerating strings.

Strings can be juxtaposed, or concatenated.

Definition 2.3. Given an alphabet Σ, given any two strings $u\colon [m] \to \Sigma$ and $v\colon [n] \to \Sigma$, the *concatenation $u \cdot v$ (also written uv) of u and v* is the string $uv\colon [m + n] \to \Sigma$, defined such that

$$uv(i) = \begin{cases} u(i) & \text{if } 1 \leq i \leq m, \\ v(i - m) & \text{if } m + 1 \leq i \leq m + n. \end{cases}$$

In particular, $u\epsilon = \epsilon u = u$. Observe that

$$|uv| = |u| + |v|.$$

For example, if $u = ga$, and $v = buzo$, then

$$uv = gabuzo$$

It is immediately verified that

$$u(vw) = (uv)w.$$

Thus, concatenation is a binary operation on Σ^* which is associative and has ϵ as an identity.

Note that generally, $uv \neq vu$, for example for $u = a$ and $v = b$.

Given a string $u \in \Sigma^*$ and $n \geq 0$, we define u^n recursively as follows:

$$\begin{aligned} u^0 &= \epsilon \\ u^{n+1} &= u^n u \quad (n \geq 0). \end{aligned}$$

Clearly, $u^1 = u$, and it is an easy exercise to show that

$$u^n u = u u^n, \quad \text{for all } n \geq 0.$$

For the induction step, we have

$$\begin{aligned} u^{n+1} u &= (u^n u) u && \text{by definition of } u^{n+1} \\ &= (u u^n) u && \text{by the induction hypothesis} \\ &= u(u^n u) && \text{by associativity} \\ &= u u^{n+1} && \text{by definition of } u^{n+1}. \end{aligned}$$

Definition 2.4. Given an alphabet Σ, given any two strings $u, v \in \Sigma^*$ we define the following notions as follows:

u is a prefix of v iff there is some $y \in \Sigma^*$ such that

$$v = uy.$$

u is a suffix of v iff there is some $x \in \Sigma^*$ such that

$$v = xu.$$

u is a substring of v iff there are some $x, y \in \Sigma^*$ such that

$$v = xuy.$$

We say that *u is a proper prefix (suffix, substring) of v* iff u is a prefix (suffix, substring) of v and $u \neq v$.

For example, *ga* is a prefix of *gabuzo*,

zo is a suffix of *gabuzo* and

buz is a substring of *gabuzo*.

Recall that a partial ordering $\leq$ on a set S is a binary relation $\leq \subseteq S \times S$ which is reflexive, transitive, and antisymmetric.

The concepts of prefix, suffix, and substring, define binary relations on Σ^* in the obvious way. It can be shown that these relations are partial orderings.

Another important ordering on strings is the lexicographic (or dictionary) ordering.

Definition 2.5. Given an alphabet $\Sigma = \{a_1, \ldots, a_k\}$ assumed totally ordered such that $a_1 < a_2 < \cdots < a_k$, given any two strings $u, v \in \Sigma^*$, we define the *lexicographic ordering* $\preceq$ as follows:

$$u \preceq v \quad \begin{cases} \text{(1) if } v = uy, \text{ for some } y \in \Sigma^*, \text{ or} \\ \text{(2) if } u = xa_iy,\ v = xa_jz,\ a_i < a_j, \\ \text{with } a_i, a_j \in \Sigma, \text{ and for some } x, y, z \in \Sigma^*. \end{cases}$$

Note that cases (1) and (2) are mutually exclusive. In case (1) u is a prefix of v. In case (2) $v \not\preceq u$ and $u \neq v$.

For example

$$ab \preceq b, \quad gallhager \preceq gallier.$$

It is fairly tedious to prove that the lexicographic ordering is in fact a partial ordering.

In fact, it is a *total ordering*, which means that for any two strings $u, v \in \Sigma^*$, either $u \preceq v$, or $v \preceq u$.

The *reversal* w^R of a string w is defined inductively as follows:

$$\begin{aligned} \epsilon^R &= \epsilon, \\ (ua)^R &= au^R, \end{aligned}$$

where $a \in \Sigma$ and $u \in \Sigma^*$.

For example

$$reillag = gallier^R.$$

It can be shown that

$$(uv)^R = v^R u^R.$$

Thus,

$$(u_1 \ldots u_n)^R = u_n^R \ldots u_1^R,$$

and when $u_i \in \Sigma$, we have

$$(u_1 \ldots u_n)^R = u_n \ldots u_1.$$

We can now define languages.

Definition 2.6. Given an alphabet Σ, a *language over* Σ *(or simply a language)* is any subset L of Σ^*.

If $\Sigma \neq \emptyset$, there are uncountably many languages.

A Quick Review of Finite, Infinite, Countable, and Uncountable Sets

For details and proofs, see *Discrete Mathematics*, by Gallier.

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of *natural numbers*.

Recall that a set X is *finite* if there is some natural number $n \in \mathbb{N}$ and a bijection between X and the set $[n] = \{1, 2, \ldots, n\}$. (When $n = 0$, $X = \emptyset$, the empty set.)

The number n is uniquely determined. It is called the *cardinality (or size)* of X and is denoted by $|X|$.

A set is *infinite* iff it is not finite.

Recall that any injection or surjection of a finite set to itself is in fact a *bijection*.

The above fails for infinite sets.

The *pigeonhole principle* asserts that *there is no bijection between a finite set X and any proper subset Y of X*.

Consequence: If we think of X as a set of n pigeons and if there are only $m < n$ boxes (corresponding to the elements of Y), then at least two of the pigeons must share the same box.

As a consequence of the pigeonhole principle, a set X is infinite iff it is in bijection with a proper subset of itself.

For example, we have a bijection $n \mapsto 2n$ between $\mathbb{N}$ and the set $2\mathbb{N}$ of even natural numbers, a proper subset of $\mathbb{N}$, so $\mathbb{N}$ is infinite.

A set X is *countable (or denumerable)* if there is an *injection* from X into $\mathbb{N}$.

If X is not the empty set, then X is countable iff there is a *surjection* from $\mathbb{N}$ onto X.

It can be shown that a set X is countable if either it is finite or if it is in bijection with $\mathbb{N}$.

We will see later that $\mathbb{N} \times \mathbb{N}$ is countable. As a consequence, the set $\mathbb{Q}$ of rational numbers is countable.

A set is *uncountable* if it is not countable.

For example, $\mathbb{R}$ (the set of real numbers) is uncountable.

Similarly

$$(0, 1) = \{x \in \mathbb{R} \mid 0 < x < 1\}$$

is uncountable. However, there is a bijection between $(0, 1)$ and $\mathbb{R}$ (find one!)

The set $2^{\mathbb{N}}$ of all subsets of $\mathbb{N}$ is uncountable.

If $\Sigma \neq \emptyset$, then the set Σ^* of all strings over Σ is infinite and countable.

Suppose $|\Sigma| = k$ with $\Sigma = \{a_1, \ldots, a_k\}$.

If $k = 1$ write $a = a_1$, and then

$$\{a\}^* = \{\epsilon, a, aa, aaa, \ldots, a^n, \ldots\}.$$

We have the bijection $n \mapsto a^n$ from $\mathbb{N}$ to $\{a\}^*$.

If $k \geq 2$, then we can think of the string

$$u = a_{i_1} \cdots a_{i_n}$$

as a representation of the integer $\nu(u)$ in base k shifted by $(k^n - 1)/(k - 1)$,

$$\nu(u) = i_1 k^{n-1} + i_2 k^{n-2} + \cdots + i_{n-1} k + i_n$$
$$= \frac{k^n - 1}{k - 1} + (i_1 - 1)k^{n-1} + \cdots + (i_{n-1} - 1)k + i_n - 1.$$

(with $\nu(\epsilon) = 0$).

We leave it as an exercise to show that $\nu\colon \Sigma^* \to \mathbb{N}$ is a bijection.

In fact, ν correspond to the enumeration of Σ^* where u precedes v if $|u| < |v|$, and u precedes v in the lexicographic ordering if $|u| = |v|$.

For example, if $k = 2$ and if we write $\Sigma = \{a, b\}$, then the enumeration begins with

$$\epsilon,\ a,\ b,\ aa,\ ab,\ ba,\ bb.$$

On the other hand, if $\Sigma \neq \emptyset$, the set 2^{Σ^*} of all subsets of Σ^* (all languages) is *uncountable*.

Indeed, we can show that there is no surjection from $\mathbb{N}$ onto 2^{Σ^*}.

First, we show that there is no surjection from Σ^* onto 2^{Σ^*}.

We claim that if there is no surjection from Σ^* onto 2^{Σ^*}, then there is no surjection from $\mathbb{N}$ onto 2^{Σ^*} either.

Assume by contradiction that there is a surjection $g\colon \mathbb{N} \to 2^{\Sigma^*}$. But, if $\Sigma \neq \emptyset$, then Σ^* is infinite and countable, thus we have the bijection $\nu\colon \Sigma^* \to \mathbb{N}$. Then the composition

$$\Sigma^* \xrightarrow{\ \nu\ } \mathbb{N} \xrightarrow{\ g\ } 2^{\Sigma^*}$$

is a surjection, because the bijection ν is a surjection, g is a surjection, and the composition of surjections is a surjection, contradicting the hypothesis that there is no surjection from Σ^* onto 2^{Σ^*}.

To prove that that there is no surjection Σ^* onto 2^{Σ^*}. We use a *diagonalization* argument. This is an instance of *Cantor's Theorem*.

Theorem 2.1. *(Cantor) There is no surjection from Σ^* onto 2^{Σ^*}.*

Proof. Assume there is a surjection $h\colon \Sigma^* \to 2^{\Sigma^*}$, and consider the set

$$D = \{u \in \Sigma^* \mid u \notin h(u)\}.$$

By definition, for any u we have $u \in D$ iff $u \notin h(u)$. Since h is surjective, there is some $w \in \Sigma^*$ such that $h(w) = D$. Then, since by definition of D and since $D = h(w)$, we have

$w \in D$ iff $w \notin h(w) = D$,

a contradiction. Therefore g is not surjective. □

Therefore, if $\Sigma \neq \emptyset$, then 2^{Σ^*} is uncountable.

We will try to single out countable "tractable" families of languages.

We will begin with the family of *regular languages*, and then proceed to the *context-free languages*.

We now turn to operations on languages.

2.2 Operations on Languages

A way of building more complex languages from simpler ones is to combine them using various operations. First, we review the set-theoretic operations of union, intersection, and complementation.

Given some alphabet Σ, for any two languages L_1, L_2 over Σ, the *union* $L_1 \cup L_2$ of L_1 and L_2 is the language

$$L_1 \cup L_2 = \{w \in \Sigma^* \mid w \in L_1 \quad \text{or} \quad w \in L_2\}.$$

The *intersection* $L_1 \cap L_2$ of L_1 and L_2 is the language

$$L_1 \cap L_2 = \{w \in \Sigma^* \mid w \in L_1 \quad \text{and} \quad w \in L_2\}.$$

The *difference* $L_1 - L_2$ of L_1 and L_2 is the language

$$L_1 - L_2 = \{w \in \Sigma^* \mid w \in L_1 \quad \text{and} \quad w \notin L_2\}.$$

The difference is also called the *relative complement*.

A special case of the difference is obtained when $L_1 = \Sigma^*$, in which case we define the *complement* $\overline{L}$ of a language L as

$$\overline{L} = \{w \in \Sigma^* \mid w \notin L\}.$$

The above operations do not use the structure of strings. The following operations use concatenation.

Definition 2.7. Given an alphabet Σ, for any two languages L_1, L_2 over Σ, the *concatenation L_1L_2 of L_1 and L_2* is the language

$$L_1L_2 = \{w \in \Sigma^* \mid \exists u \in L_1,\ \exists v \in L_2,\ w = uv\}.$$

For any language L, we define L^n as follows:

$$\begin{aligned} L^0 &= \{\epsilon\}, \\ L^{n+1} &= L^nL \quad (n \geq 0). \end{aligned}$$

The following properties are easily verified:

$$\begin{aligned} L\emptyset &= \emptyset, \\ \emptyset L &= \emptyset, \\ L\{\epsilon\} &= L, \\ \{\epsilon\}L &= L, \\ (L_1 \cup \{\epsilon\})L_2 &= L_1L_2 \cup L_2, \\ L_1(L_2 \cup \{\epsilon\}) &= L_1L_2 \cup L_1, \\ L^nL &= LL^n. \end{aligned}$$

In general, $L_1L_2 \neq L_2L_1$.

So far, the operations that we have introduced, except complementation (since $\overline{L} = \Sigma^* - L$ is infinite if L is finite and Σ is nonempty), preserve the finiteness of languages. This is not the case for the next two operations.

Definition 2.8. Given an alphabet Σ, for any language L over Σ, the *Kleene $*$-closure L^* of L* is the language

$$L^* = \bigcup_{n \geq 0} L^n.$$

The *Kleene $+$-closure L^+ of L* is the language

$$L^+ = \bigcup_{n \geq 1} L^n.$$

Thus, L^* is the infinite union

$$L^* = L^0 \cup L^1 \cup L^2 \cup \ldots \cup L^n \cup \ldots,$$

and L^+ is the infinite union

$$L^+ = L^1 \cup L^2 \cup \ldots \cup L^n \cup \ldots.$$

Since $L^1 = L$, both L^* and L^+ contain L.

In fact,

$$\begin{aligned} L^+ = \{&w \in \Sigma^*, \exists n \geq 1, \\ &\exists u_1 \in L \cdots \exists u_n \in L, w = u_1 \cdots u_n\}, \end{aligned}$$

and since $L^0 = \{\epsilon\}$,

$$\begin{aligned} L^* = \{\epsilon\} \cup \{&w \in \Sigma^*, \exists n \geq 1, \\ &\exists u_1 \in L \cdots \exists u_n \in L, w = u_1 \cdots u_n\}. \end{aligned}$$

Thus, the language L^* always contains ϵ, and we have

$$L^* = L^+ \cup \{\epsilon\}.$$

However, if $\epsilon \notin L$, then $\epsilon \notin L^+$. The following is easily shown:

$$\begin{aligned} \emptyset^* &= \{\epsilon\}, \\ L^+ &= L^* L, \\ L^{**} &= L^*, \\ L^* L^* &= L^*. \end{aligned}$$

The Kleene closures have many other interesting properties.

Homomorphisms are also very useful.

Given two alphabets Σ, Δ, a *homomorphism* *$h \colon \Sigma^* \to \Delta^*$ between Σ^* and Δ^** is a function $h \colon \Sigma^* \to \Delta^*$ such that

$$h(uv) = h(u)h(v) \quad \text{for all } u, v \in \Sigma^*.$$

Letting $u = v = \epsilon$, we get

$$h(\epsilon) = h(\epsilon)h(\epsilon),$$

which implies that (why?)

$$h(\epsilon) = \epsilon.$$

If $\Sigma = \{a_1, \ldots, a_k\}$, it is easily seen that h is completely determined by $h(a_1), \ldots, h(a_k)$ (why?)

Example: $\Sigma = \{a, b, c\}$, $\Delta = \{0, 1\}$, and

$$h(a) = 01, \quad h(b) = 011, \quad h(c) = 0111.$$

For example

$$h(abbc) = 010110110111.$$

Given any language $L_1 \subseteq \Sigma^*$, we define the *image $h(L_1)$ of L_1* as

$$h(L_1) = \{h(u) \in \Delta^* \mid u \in L_1\}.$$

Given any language $L_2 \subseteq \Delta^*$, we define the
inverse image $h^{-1}(L_2)$ of L_2 as

$$h^{-1}(L_2) = \{u \in \Sigma^* \mid h(u) \in L_2\}.$$

We now turn to the first formalism for defining languages, Deterministic Finite Automata (DFA's)

Chapter 3

DFA's, NFA's, Regular Languages

The family of regular languages is the simplest, yet interesting family of languages.

We give six definitions of the regular languages.

1. Using *deterministic finite automata (DFAs)*.
2. Using *nondeterministic finite automata (NFAs)*.
3. Using a *closure definition* involving, union, concatenation, and Kleene $*$.
4. Using *regular expressions*.
5. Using *right-invariant equivalence relations of finite index* (the Myhill-Nerode characterization).
6. Using *right-linear context-free grammars*.

We prove the equivalence of these definitions, often by providing an *algorithm* for converting one formulation into another.

We find that the introduction of NFA's is motivated by the conversion of regular expressions into DFA's.

To finish this conversion, we also show that every NFA can be converted into a DFA (using the *subset construction*).

So, although NFA's often allow for more concise descriptions, they do not have more expressive power than DFA's.

NFA's operate according to the paradigm: *guess a successful path, and check it in polynomial time*.

This is the essence of an important class of hard problems known as $\mathcal{NP}$, which will be investigated later.

We will also discuss methods for proving that certain languages are not regular (Myhill-Nerode, pumping lemma).

We present algorithms to convert a DFA to an equivalent one with a minimal number of states.

3.1 Deterministic Finite Automata (DFA's)

First we define what DFA's are, and then we explain how they are used to accept or reject strings. Roughly speaking, a DFA is a finite transition graph whose edges are labeled with letters from an alphabet Σ.

The graph also satisfies certain properties that make it deterministic. Basically, this means that given any string w, starting from any node, *there is a unique path in the graph "parsing" the string w*.

Example 1. A DFA for the language

$$L_1 = \{ab\}^+ = \{ab\}^*\{ab\},$$

i.e.,

$$L_1 = \{ab, abab, ababab, \ldots, (ab)^n, \ldots\}.$$

Input alphabet: $\Sigma = \{a, b\}$.

State set $Q_1 = \{0, 1, 2, 3\}$.

Start state: 0.

Set of accepting states: $F_1 = \{2\}$.

Transition table (function) δ_1:

	a	b
0	1	3
1	3	2
2	1	3
3	3	3

Note that state 3 is a *trap state* or *dead state*.

Here is a graph representation of the DFA specified by the transition function shown above:

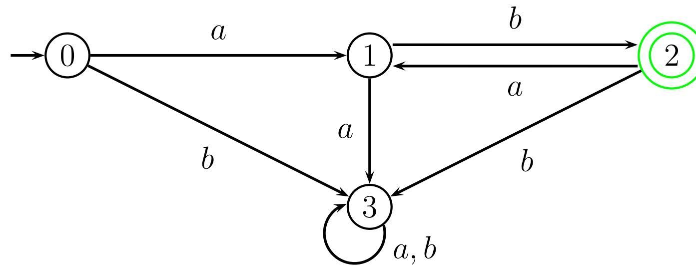


Figure 3.1: DFA for $\{ab\}^+$

Example 2. A DFA for the language

$$L_2 = \{ab\}^* = L_1 \cup \{\epsilon\}$$

i.e.,

$$L_2 = \{\epsilon, ab, abab, ababab, \ldots, (ab)^n, \ldots\}.$$

Input alphabet: $\Sigma = \{a, b\}$.

State set $Q_2 = \{0, 1, 2\}$.

Start state: 0.

Set of accepting states: $F_2 = \{0\}$.

Transition table (function) δ_2:

	a	b
0	1	2
1	2	0
2	2	2

State 2 is a *trap state* or *dead state*.

Here is a graph representation of the DFA specified by the transition function shown above:

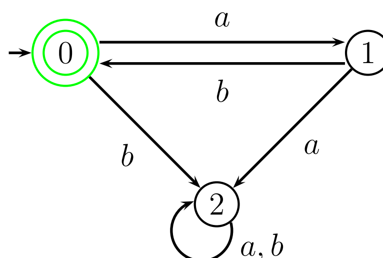


Figure 3.2: DFA for $\{ab\}^*$

Example 3. A DFA for the language

$$L_3 = \{a, b\}^*\{abb\}.$$

Note that L_3 consists of all strings of a's and b's ending in abb.

Input alphabet: $\Sigma = \{a, b\}$.

State set $Q_3 = \{0, 1, 2, 3\}$.

Start state: 0.

Set of accepting states: $F_3 = \{3\}$.

Transition table (function) δ_3:

	a	b
0	1	0
1	1	2
2	1	3
3	1	0

Here is a graph representation of the DFA specified by the transition function shown above:

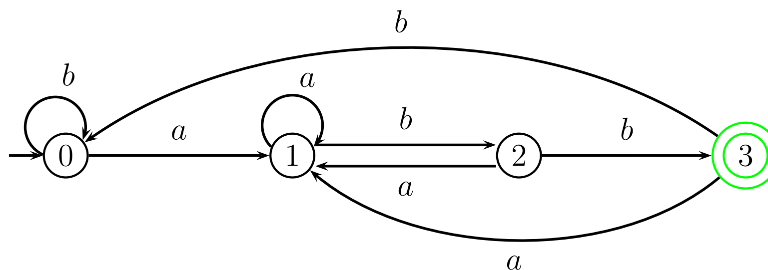


Figure 3.3: DFA for $\{a, b\}^*\{abb\}$

Is this a minimal DFA?

Definition 3.1. A *deterministic finite automaton (or DFA)* is a quintuple $D = (Q, \Sigma, \delta, q_0, F)$, where

- Σ is a finite *input alphabet*;
- Q is a finite set of *states*;

- F is a subset of Q of *final (or accepting) states*;
- $q_0 \in Q$ is the *start state (or initial state)*;
- δ is the *transition function*, a function

$$\delta \colon Q \times \Sigma \to Q.$$

For any state $p \in Q$ and any input $a \in \Sigma$, the state $q = \delta(p, a)$ is uniquely determined.

Thus, it is possible to define the state reached from a given state $p \in Q$ on input $w \in \Sigma^*$, following the path specified by w.

Technically, this is done by defining the extended transition function $\delta^* \colon Q \times \Sigma^* \to Q$.

Definition 3.2. Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, the *extended transition function $\delta^* \colon Q \times \Sigma^* \to Q$* is defined as follows:

$$\begin{aligned} \delta^*(p, \epsilon) &= p, \\ \delta^*(p, ua) &= \delta(\delta^*(p, u), a), \end{aligned}$$

where $a \in \Sigma$ and $u \in \Sigma^*$.

It is immediate that $\delta^*(p, a) = \delta(p, a)$ for $a \in \Sigma$.

The meaning of $\delta^*(p, w)$ is that it is the state reached from state p following the path from p specified by w.

We can show (by induction on the length of v) that

$$\delta^*(p, uv) = \delta^*(\delta^*(p, u), v) \quad \text{for all } p \in Q \text{ and all } u, v \in \Sigma^*$$

For the induction step, for $u \in \Sigma^*$, and all $v = ya$ with $y \in \Sigma^*$ and $a \in \Sigma$,

$$\begin{aligned} \delta^*(p, uya) &= \delta(\delta^*(p, uy), a) && \text{by definition of } \delta^* \\ &= \delta(\delta^*(\delta^*(p, u), y), a) && \text{by induction} \\ &= \delta^*(\delta^*(p, u), ya) && \text{by definition of } \delta^*. \end{aligned}$$

We can now define how a DFA accepts or rejects a string.

Definition 3.3. Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, the *language $L(D)$ accepted (or recognized) by D* is the language

$$L(D) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}.$$

Thus, a string $w \in \Sigma^*$ is accepted iff the path from q_0 on input w ends in a final state.

The definition of a DFA does not prevent the possibility that a DFA may have states that are not reachable from the start state q_0, which means that there is no path from q_0 to such states.

For example, in the DFA D_1 defined by the transition table below and the set of final states $F = \{1, 2, 3\}$, the states in the set $\{0, 1\}$ are reachable from the start state 0, but the states in the set $\{2, 3, 4\}$ are not (even though there are transitions from $2, 3, 4$ to 0, but they go in the wrong direction).

	a	b
0	1	0
1	0	1
2	3	0
3	4	0
4	2	0

Since there is no path from the start state 0 to any of the states in $\{2, 3, 4\}$, the states $2, 3, 4$ are useless as far as acceptance of strings, so they should be deleted as well as the transitions from them.

Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, the above suggests defining the set Q_r of *reachable* (or *accessible*) states as

$$Q_r = \{p \in Q \mid (\exists u \in \Sigma^*)(p = \delta^*(q_0, u))\}.$$

The set Q_r consists of those states $p \in Q$ such that there is some path from q_0 to p (along some string u).

Computing the set Q_r is a reachability problem in a directed graph. There are various algorithms to solve this problem, including breadth-first search or depth-first search.

Once the set Q_r has been computed, we can clean up the DFA D by deleting all redundant states in $Q - Q_r$ and all transitions from these states.

More precisely, we form the DFA $D_r = (Q_r, \Sigma, \delta_r, q_0, Q_r \cap F)$, where $\delta_r \colon Q_r \times \Sigma \to Q_r$ is the restriction of $\delta \colon Q \times \Sigma \to Q$ to Q_r.

If D_1 is the DFA of the previous example, then the DFA $(D_1)_r$ is obtained by deleting the states $2, 3, 4$:

	a	b
0	1	0
1	0	1

It can be shown that $L(D_r) = L(D)$ (see the homework problems).

A DFA D such that $Q = Q_r$ is said to be *trim* (or *reduced*).

Observe that the DFA D_r is trim. A minimal DFA must be trim.

Computing Q_r gives us a method to test whether a DFA D accepts a nonempty language. Indeed

$$L(D) \neq \emptyset \quad \text{iff} \quad Q_r \cap F \neq \emptyset$$

We now come to the first of several equivalent definitions of the regular languages.

Regular Languages, Version 1

Definition 3.4. A language L is a *regular language* if it is accepted by some DFA.

Note that a regular language may be accepted by many different DFAs. Later on, we will investigate how to find minimal DFA's.

For a given regular language L, a minimal DFA for L is a DFA with the smallest number of states among all DFA's accepting L. A minimal DFA for L must exist since every nonempty subset of natural numbers has a smallest element.

In order to understand how complex the regular languages are, we will investigate the closure properties of the regular languages under union, intersection, complementation, concatenation, and Kleene $*$.

It turns out that the family of regular languages is closed under all these operations. For union, intersection, and complementation, we can use the cross-product construction which preserves determinism.

However, for concatenation and Kleene $*$, there does not appear to be any method involving DFA's only. The way to do it is to introduce nondeterministic finite automata (NFA's), which we do a little later.

3.2 The “Cross-product” Construction

Let $\Sigma = \{a_1, \ldots, a_m\}$ be an alphabet.

Given any two DFA's $D_1 = (Q_1, \Sigma, \delta_1, q_{0,1}, F_1)$ and
$D_2 = (Q_2, \Sigma, \delta_2, q_{0,2}, F_2)$, there is a very useful construction for showing that the union, the intersection, or the relative complement of regular languages, is a regular language.

Given any two languages L_1, L_2 over Σ, recall that

$$\begin{aligned} L_1 \cup L_2 &= \{w \in \Sigma^* \mid w \in L_1 \quad \text{or} \quad w \in L_2\}, \\ L_1 \cap L_2 &= \{w \in \Sigma^* \mid w \in L_1 \quad \text{and} \quad w \in L_2\}, \\ L_1 - L_2 &= \{w \in \Sigma^* \mid w \in L_1 \quad \text{and} \quad w \notin L_2\}. \end{aligned}$$

Let us first explain how to constuct a DFA accepting the intersection $L_1 \cap L_2$. Let D_1 and D_2 be DFA's such that $L_1 = L(D_1)$ and $L_2 = L(D_2)$.

The idea is to construct a DFA *simulating D_1 and D_2 in parallel*. This can be done by using states which are pairs $(p_1, p_2) \in Q_1 \times Q_2$.

Thus, we define the DFA D as follows:

$$D = (Q_1 \times Q_2, \Sigma, \delta, (q_{0,1}, q_{0,2}), F_1 \times F_2),$$

where the transition function $\delta\colon (Q_1 \times Q_2) \times \Sigma \to Q_1 \times Q_2$ is defined as follows:

$$\delta((p_1, p_2),\, a) = (\delta_1(p_1, a),\, \delta_2(p_2, a)),$$

for all $p_1 \in Q_1$, $p_2 \in Q_2$, and $a \in \Sigma$.

Clearly, D is a DFA, since D_1 and D_2 are. Also, by the definition of δ, we have

$$\delta^*((p_1, p_2),\, w) = (\delta_1^*(p_1, w),\, \delta_2^*(p_2, w)),$$

for all $p_1 \in Q_1$, $p_2 \in Q_2$, and $w \in \Sigma^*$.

Now, we have $w \in L(D_1) \cap L(D_2)$

$$\begin{aligned}
&\text{iff} \quad w \in L(D_1) \text{ and } w \in L(D_2),\\
&\text{iff} \quad \delta_1^*(q_{0,1}, w) \in F_1 \text{ and } \delta_2^*(q_{0,2}, w) \in F_2,\\
&\text{iff} \quad (\delta_1^*(q_{0,1}, w), \delta_2^*(q_{0,2}, w)) \in F_1 \times F_2,\\
&\text{iff} \quad \delta^*((q_{0,1}, q_{0,2}),\, w) \in F_1 \times F_2,\\
&\text{iff} \quad w \in L(D).
\end{aligned}$$

Thus, $L(D) = L(D_1) \cap L(D_2)$.

We can now modify D very easily to accept $L(D_1) \cup L(D_2)$.

We change the set of final states so that it becomes $(F_1 \times Q_2) \cup (Q_1 \times F_2)$.

Indeed, $w \in L(D_1) \cup L(D_2)$

$$\begin{aligned}
&\text{iff} \quad w \in L(D_1) \text{ or } w \in L(D_2),\\
&\text{iff} \quad \delta_1^*(q_{0,1}, w) \in F_1 \text{ or } \delta_2^*(q_{0,2}, w) \in F_2,\\
&\text{iff} \quad (\delta_1^*(q_{0,1}, w), \delta_2^*(q_{0,2}, w)) \in (F_1 \times Q_2) \cup (Q_1 \times F_2),\\
&\text{iff} \quad \delta^*((q_{0,1}, q_{0,2}),\, w) \in (F_1 \times Q_2) \cup (Q_1 \times F_2),\\
&\text{iff} \quad w \in L(D).
\end{aligned}$$

Thus, $L(D) = L(D_1) \cup L(D_2)$.

We can also modify D very easily to accept $L(D_1) - L(D_2)$.

We change the set of final states so that it becomes $F_1 \times (Q_2 - F_2)$.

Indeed, $w \in L(D_1) - L(D_2)$

$$\begin{aligned}
&\text{iff} \quad w \in L(D_1) \text{ and } w \notin L(D_2),\\
&\text{iff} \quad \delta_1^*(q_{0,1}, w) \in F_1 \text{ and } \delta_2^*(q_{0,2}, w) \notin F_2,\\
&\text{iff} \quad (\delta_1^*(q_{0,1}, w), \delta_2^*(q_{0,2}, w)) \in F_1 \times (Q_2 - F_2),\\
&\text{iff} \quad \delta^*((q_{0,1}, q_{0,2}), w) \in F_1 \times (Q_2 - F_2),\\
&\text{iff} \quad w \in L(D).
\end{aligned}$$

Thus, $L(D) = L(D_1) - L(D_2)$.

In all cases, if D_1 has n_1 states and D_2 has n_2 states, the DFA D has $n_1 n_2$ states.

3.3 Nondeteterministic Finite Automata (NFA's)

NFA's are obtained from DFA's by allowing multiple transitions from a given state on a given input. This can be done by defining $\delta(p, a)$ as a **subset** of Q rather than a single state. It will also be convenient to allow transitions on input ϵ.

We let 2^Q denote the set of all subsets of Q, including the empty set. The set 2^Q is the *power set* of Q.

Example 4. A NFA for the language

$$L_3 = \{a, b\}^*\{abb\}.$$

Input alphabet: $\Sigma = \{a, b\}$.

State set $Q_4 = \{0, 1, 2, 3\}$.

Start state: 0.

Set of accepting states: $F_4 = \{3\}$.

Transition table δ_4:

	a	b
0	$\{0, 1\}$	$\{0\}$
1	$\emptyset$	$\{2\}$
2	$\emptyset$	$\{3\}$
3	$\emptyset$	$\emptyset$

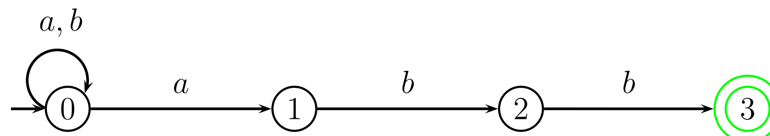


Figure 3.4: NFA for $\{a, b\}^*\{abb\}$

Example 5. Let $\Sigma = \{a_1, \ldots, a_n\}$, let

$$L_n^i = \{w \in \Sigma^* \mid w \text{ contains an odd number of } a_i\text{'s}\},$$

and let

$$L_n = L_n^1 \cup L_n^2 \cup \cdots \cup L_n^n.$$

The language L_n consists of those strings in Σ^* that contain an odd number of some letter $a_i \in \Sigma$.

Equivalently $\Sigma^* - L_n$ consists of those strings in Σ^* with an even number of *every* letter $a_i \in \Sigma$.

It can be shown that every DFA accepting L_n has at least 2^n states.

However, there is an NFA with $2n + 1$ states accepting L_n.

We define NFA's as follows.

Definition 3.5. A *nondeterministic finite automaton (or NFA)* is a quintuple $N = (Q, \Sigma, \delta, q_0, F)$, where

- Σ is a finite *input alphabet*;
- Q is a finite set of *states*;
- F is a subset of Q of *final (or accepting) states*;
- $q_0 \in Q$ is the *start state (or initial state)*;
- δ is the *transition function*, a function

$$\delta \colon Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q.$$

For any state $p \in Q$ and any input $a \in \Sigma \cup \{\epsilon\}$, the set of states $\delta(p, a)$ is uniquely determined. We write $q \in \delta(p, a)$.

Given an NFA $N = (Q, \Sigma, \delta, q_0, F)$, we would like to define the language accepted by N.

However, given an NFA N, unlike the situation for DFA's, given a state $p \in Q$ and some input $w \in \Sigma^*$, in general *there is no unique path from p on input w, but instead a tree of computation paths*.

For example, given the NFA shown below,

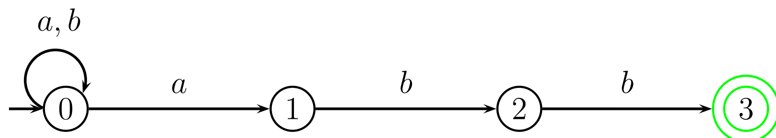


Figure 3.5: NFA for $\{a, b\}^*\{abb\}$

from state 0 on input $w = ababb$ we obtain the following tree of computation paths:

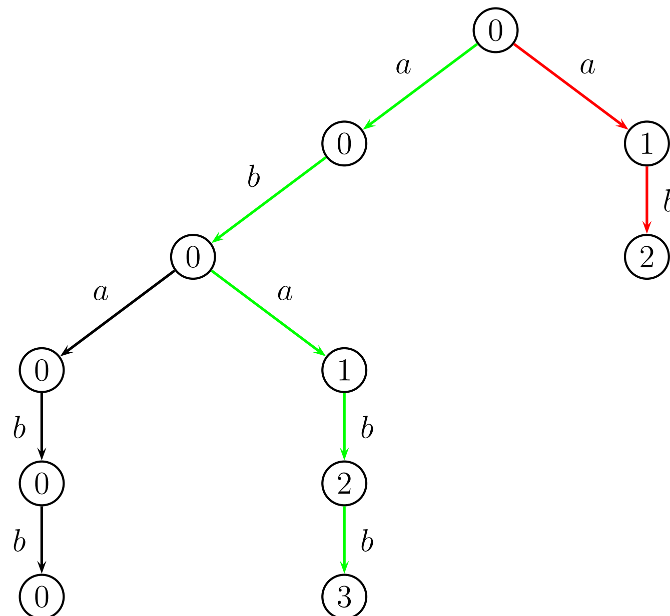


Figure 3.6: A tree of computation paths on input *ababb*

Observe that there are three kinds of computation paths:

1. A path on input w ending in a rejecting state (for example, the lefmost path).

2. A path on some proper prefix of w, along which the computation gets stuck (for example, the rightmost path).

3. A path on input w ending in an accepting state (such as the path ending in state 3).

The acceptance criterion for NFA is *very lenient*: a string w is accepted iff the tree of computation paths contains *some accepting path* (of type (3)).

Thus, all failed paths of type (1) and (2) are ignored. Furthermore, there is *no charge* for failed paths.

A string w is rejected iff all computation paths are failed paths of type (1) or (2).

The "philosophy" of nondeterminism is that an NFA
"guesses" an accepting path and then checks it in polynomial time by following this path. We are only charged for one accepting path (even if there are several accepting paths).

A way to capture this acceptance policy if to extend the transition function $\delta\colon Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$ to a function

$$\delta^* \colon Q \times \Sigma^* \to 2^Q.$$

The presence of ϵ-transitions (i.e., when $q \in \delta(p, \epsilon)$) causes technical problems, and to overcome these problems, we introduce the notion of ϵ-closure.

3.4 ϵ-Closure

Definition 3.6. Given an NFA $N = (Q, \Sigma, \delta, q_0, F)$ (with ϵ-transitions) for every state $p \in Q$, the *ϵ-closure of p* is set ϵ-closure(p) consisting of all states q such that there is a path from p to q whose spelling is ϵ (an *ϵ-path*).

This means that either $q = p$, or that all the edges on the path from p to q have the label ϵ.

We can compute ϵ-closure(p) using a sequence of approximations as follows. Define the sequence of sets of states $(\epsilon\text{-clo}_i(p))_{i\geq 0}$ as follows:

$$\begin{aligned}\epsilon\text{-clo}_0(p) &= \{p\},\\ \epsilon\text{-clo}_{i+1}(p) &= \epsilon\text{-clo}_i(p) \cup\\ &\qquad \{q \in Q \mid \exists s \in \epsilon\text{-clo}_i(p),\ q \in \delta(s, \epsilon)\}.\end{aligned}$$

Since $\epsilon\text{-clo}_i(p) \subseteq \epsilon\text{-clo}_{i+1}(p)$, $\epsilon\text{-clo}_i(p) \subseteq Q$, for all $i \geq 0$, and Q is finite, it can be shown that there is a smallest i, say i_0, such that

$$\epsilon\text{-clo}_{i_0}(p) = \epsilon\text{-clo}_{i_0+1}(p).$$

It suffices to show that there is some $i \geq 0$ such that $\epsilon\text{-clo}_i(p) = \epsilon\text{-clo}_{i+1}(p)$, because then there is a smallest such i (since every nonempty subset of $\mathbb{N}$ has a smallest element).

Assume by contradiction that

$$\epsilon\text{-clo}_i(p) \subset \epsilon\text{-clo}_{i+1}(p) \quad \text{for all } i \geq 0.$$

Then, I claim that $|\epsilon\text{-clo}_i(p)| \geq i + 1$ for all $i \geq 0$.

This is true for $i = 0$ since $\epsilon\text{-clo}_0(p) = \{p\}$.

Since $\epsilon\text{-clo}_i(p) \subset \epsilon\text{-clo}_{i+1}(p)$, there is some $q \in \epsilon\text{-clo}_{i+1}(p)$ that does not belong to $\epsilon\text{-clo}_i(p)$, and since by induction $|\epsilon\text{-clo}_i(p)| \geq i + 1$, we get

$$|\epsilon\text{-clo}_{i+1}(p)| \geq |\epsilon\text{-clo}_i(p)| + 1 \geq i + 1 + 1 = i + 2,$$

establishing the induction hypothesis.

If $n = |Q|$, then $|\epsilon\text{-clo}_n(p)| \geq n + 1$, a contradiction.

Therefore, there is indeed some $i \geq 0$ such that $\epsilon\text{-clo}_i(p) = \epsilon\text{-clo}_{i+1}(p)$, and for the least such $i = i_0$, we have $i_0 \leq n - 1$.

It can also be shown that

$$\epsilon\text{-closure}(p) = \epsilon\text{-clo}_{i_0}(p),$$

by proving that

1. $\epsilon\text{-clo}_i(p) \subseteq \epsilon\text{-closure}(p)$, for all $i \geq 0$.
2. $\epsilon\text{-closure}(p)_i \subseteq \epsilon\text{-clo}_{i_0}(p)$, for all $i \geq 0$.

where $\epsilon\text{-closure}(p)_i$ is the set of states reachable from p by an ϵ-path of length $\leq i$.

When N has no ϵ-transitions, i.e., when $\delta(p, \epsilon) = \emptyset$ for all $p \in Q$ (which means that δ can be viewed as a function $\delta\colon Q \times \Sigma \to 2^Q$), we have

$$\epsilon\text{-closure}(p) = \{p\}.$$

It should be noted that there are more efficient ways of computing ϵ-closure(p), for example, using a stack (basically, a kind of depth-first search).

We present such an algorithm below. It is assumed that the types *NFA* and *stack* are defined. If n is the number of states of an NFA N, we let

```
eclotype = array[1..n] of boolean
function eclosure[N: NFA, p: integer]: eclotype;
  begin
    var eclo: eclotype, q, s: integer, st: stack;
    for each q ∈ setstates(N) do
      eclo[q] := false;
    endfor
    eclo[p] := true; st := empty;
    trans := deltatable(N);
    st := push(st, p);
    while st ≠ emptystack do
      q = pop(st);
      for each s ∈ trans(q, ϵ) do
        if eclo[s] = false then
          eclo[s] := true; st := push(st, s)
```

endif
endfor
endwhile;
eclosure := *eclo*
end

This algorithm can be easily adapted to compute the set of states reachable from a given state p (in a DFA or an NFA).

Given a subset S of Q, we define ϵ-closure(S) as

$$\epsilon\text{-closure}(S) = \bigcup_{s \in S} \epsilon\text{-closure}(s),$$

with

$$\epsilon\text{-closure}(\emptyset) = \emptyset.$$

When N has no ϵ-transitions, we have

$$\epsilon\text{-closure}(S) = S.$$

We are now ready to define the extension $\delta^* \colon Q \times \Sigma^* \to 2^Q$ of the transition function $\delta \colon Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$.

3.5 Converting an NFA into a DFA

The intuition behind the definition of the extended transition function is that $\delta^*(p, w)$ is the set of all states reachable from p by a path whose spelling is w.

Definition 3.7. Given an NFA $N = (Q, \Sigma, \delta, q_0, F)$ (with ϵ-transitions), the *extended transition function* $\delta^* \colon Q \times \Sigma^* \to 2^Q$ is defined as follows: for every $p \in Q$, every $u \in \Sigma^*$, and every $a \in \Sigma$,

$$\begin{aligned} \delta^*(p, \epsilon) &= \epsilon\text{-closure}(\{p\}), \\ \delta^*(p, ua) &= \epsilon\text{-closure}\Bigg(\bigcup_{s \in \delta^*(p,u)} \delta(s, a)\Bigg). \end{aligned}$$

In the second equation, if $\delta^*(p, u) = \emptyset$ then

$$\delta^*(p, ua) = \emptyset.$$

The *language $L(N)$ accepted by an NFA N* is the set

$$L(N) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \cap F \neq \emptyset\}.$$

Observe that the definition of $L(N)$ conforms to the lenient acceptance policy: a string w is accepted iff $\delta^*(q_0, w)$ contains *some final state*.

We can also extend $\delta^*\colon Q \times \Sigma^* \to 2^Q$ to a function

$$\widehat{\delta}\colon 2^Q \times \Sigma^* \to 2^Q$$

defined as follows: for every subset S of Q, for every $w \in \Sigma^*$,

$$\widehat{\delta}(S, w) = \bigcup_{s \in S} \delta^*(s, w),$$

with

$$\widehat{\delta}(\emptyset, w) = \emptyset.$$

Let $\mathcal{Q}$ be the subset of 2^Q consisting of those subsets S of Q that are ϵ-closed, i.e., such that

$$S = \epsilon\text{-closure}(S).$$

If we consider the restriction

$$\Delta\colon \mathcal{Q} \times \Sigma \to \mathcal{Q}$$

of $\widehat{\delta}\colon 2^Q \times \Sigma^* \to 2^Q$ to $\mathcal{Q}$ and Σ, we observe that Δ is the transition function of a DFA.

Indeed, this is the transition function of a DFA accepting $L(N)$. It is easy to show that Δ is defined directly as follows (on subsets S in $\mathcal{Q}$):

$$\Delta(S, a) = \epsilon\text{-closure}\left(\bigcup_{s \in S} \delta(s, a)\right),$$

with

$$\Delta(\emptyset, a) = \emptyset.$$

Then, the DFA D is defined as follows:

$$D = (\mathcal{Q}, \Sigma, \Delta, \epsilon\text{-closure}(\{q_0\}), \mathcal{F}),$$

where $\mathcal{F} = \{S \in \mathcal{Q} \mid S \cap F \neq \emptyset\}$.

It is not difficult to show that $L(D) = L(N)$, that is, D is a DFA accepting $L(N)$. For this, we show that

$$\Delta^*(S, w) = \widehat{\delta}(S, w).$$

Thus, we have converted the NFA N into a DFA D (and gotten rid of ϵ-transitions).

Since DFA's are special NFA's, the subset construction shows that DFA's and NFA's accept *the same* family of languages, the *regular languages, version 1* (although not with the same complexity).

The states of the DFA D equivalent to N are ϵ-closed subsets of Q. For this reason, the above construction is often called the *subset construction*.

This construction is due to Rabin and Scott.

Although theoretically fine, the method may construct useless sets S that are not reachable from the start state ϵ-closure($\{q_0\}$). A more economical construction is given next.

An Algorithm to convert an NFA into a DFA:
The "subset construction"

Given an input NFA $N = (Q, \Sigma, \delta, q_0, F)$, a DFA $D = (K, \Sigma, \Delta, S_0, \mathcal{F})$ is constructed. It is assumed that K is a linear array of sets of states $S \subseteq Q$, and Δ is a 2-dimensional array, where $\Delta[i, a]$ is the index of the target state of the transition from $K[i] = S$ on input a, with $S \in K$, and $a \in \Sigma$.

$S_0 := \epsilon\text{-closure}(\{q_0\});\ \mathit{total} := 1;\ K[1] := S_0;$

$\mathit{marked} := 0;$

while $\mathit{marked} < \mathit{total}$ **do**;

  $\mathit{marked} := \mathit{marked} + 1;\ S := K[\mathit{marked}];$

  for each $a \in \Sigma$ **do**

    $U := \bigcup_{s \in S} \delta(s, a);\ T := \epsilon\text{-closure}(U);$

    if $T \notin K$ **then**

      $\mathit{total} := \mathit{total} + 1;\ K[\mathit{total}] := T$

    endif;

    $\Delta[\mathit{marked}, a] := \text{index}(T)$

  endfor

endwhile;

$\mathcal{F} := \{S \in K \mid S \cap F \neq \emptyset\}$

Let us illustrate the subset construction on the NFA of Example 4.

A NFA for the language

$$L_3 = \{a, b\}^* \{abb\}.$$

Transition table δ_4:

	a	b
0	$\{0,1\}$	$\{0\}$
1	$\emptyset$	$\{2\}$
2	$\emptyset$	$\{3\}$
3	$\emptyset$	$\emptyset$

Set of accepting states: $F_4 = \{3\}$.

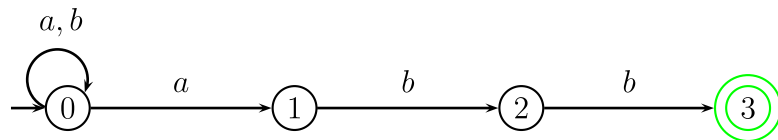


Figure 3.7: NFA for $\{a, b\}^*\{abb\}$

The pointer $\Rightarrow$ corresponds to *marked* and the pointer $\rightarrow$ to *total*.

Initial transition table Δ.

	index	states	a	b
$\Rightarrow$				
$\rightarrow$	A	$\{0\}$		

Just after entering the while loop

	index	states	a	b
$\Rightarrow\rightarrow$	A	$\{0\}$		

After the first round through the while loop.

	index	states	a	b
$\Rightarrow$	A	$\{0\}$	B	A
$\rightarrow$	B	$\{0,1\}$		

After just reentering the while loop.

	index	states	a	b
	A	$\{0\}$	B	A
$\Rightarrow\rightarrow$	B	$\{0,1\}$		

After the second round through the while loop.

	index	states	a	b
	A	$\{0\}$	B	A
$\Rightarrow$	B	$\{0,1\}$	B	C
$\rightarrow$	C	$\{0,2\}$		

After the third round through the while loop.

	index	states	a	b
	A	$\{0\}$	B	A
	B	$\{0,1\}$	B	C
$\Rightarrow$	C	$\{0,2\}$	B	D
$\rightarrow$	D	$\{0,3\}$		

After the fourth round through the while loop.

	index	states	a	b
	A	$\{0\}$	B	A
	B	$\{0,1\}$	B	C
	C	$\{0,2\}$	B	D
$\Rightarrow\rightarrow$	D	$\{0,3\}$	B	A

This is the DFA of Figure 3.3, except that in that example A, B, C, D are renamed $0, 1, 2, 3$.

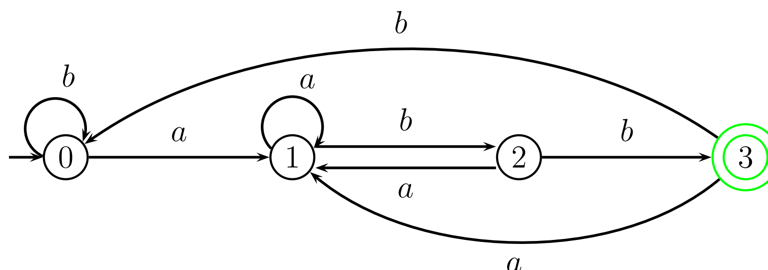


Figure 3.8: DFA for $\{a, b\}^*\{abb\}$

3.6 Finite State Automata With Output: Transducers

So far, we have only considered automata that recognize languages, i.e., automata that do not produce any output on any input (except "accept" or "reject").

It is interesting and useful to consider input/output finite state machines. Such automata are called *transducers*. They compute functions or relations. First, we define a deterministic kind of transducer.

Definition 3.8. A *general sequential machine (gsm)* is a sextuple $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where

(1) Q is a finite set of *states*,

(2) Σ is a finite *input alphabet*,

(3) Δ is a finite *output alphabet*,

(4) $\delta\colon Q \times \Sigma \to Q$ is the *transition function*,

(5) $\lambda\colon Q \times \Sigma \to \Delta^*$ is the *output function* and

(6) q_0 is the *initial* (or *start*) *state*.

If $\lambda(p, a) \neq \epsilon$, for all $p \in Q$ and all $a \in \Sigma$, then M is *nonerasing*. If $\lambda(p, a) \in \Delta$ for all $p \in Q$ and all $a \in \Sigma$, we say that M is a *complete sequential machine (csm)*.

An example of a gsm for which $\Sigma = \{a, b\}$ and $\Delta = \{0, 1, 2\}$ is shown in Figure 3.9. For example aab is converted to 102001.

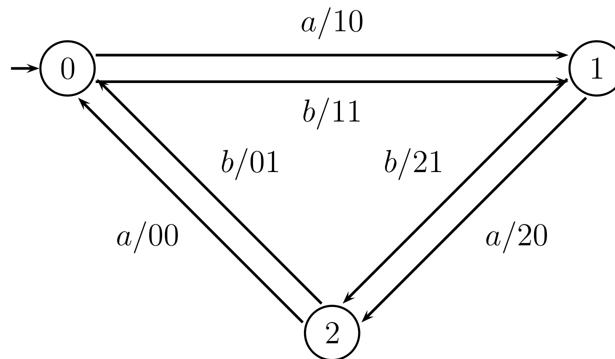


Figure 3.9: Example of a gsm

In order to define how a gsm works, we extend the transition and the output functions. We define $\delta^*\colon Q \times \Sigma^* \to Q$ and $\lambda^*\colon Q \times \Sigma^* \to \Delta^*$ recursively as follows: For all $p \in Q$, all $u \in \Sigma^*$ and all $a \in \Sigma$

$$\begin{aligned}
\delta^*(p, \epsilon) &= p \\
\delta^*(p, ua) &= \delta(\delta^*(p, u), a) \\
\lambda^*(p, \epsilon) &= \epsilon \\
\lambda^*(p, ua) &= \lambda^*(p, u)\lambda(\delta^*(p, u), a).
\end{aligned}$$

For any $w \in \Sigma^*$, we let

$$M(w) = \lambda^*(q_0, w)$$

and for any $L \subseteq \Sigma^*$ and $L' \subseteq \Delta^*$, let

$$M(L) = \{\lambda^*(q_0, w) \mid w \in L\}$$

and

$$M^{-1}(L') = \{w \in \Sigma^* \mid \lambda^*(q_0, w) \in L'\}.$$

Note that if M is a csm, then $|M(w)| = |w|$ for all $w \in \Sigma^*$. Also, a homomorphism is a special kind of gsm—it can be realized by a gsm with one state.

We can use gsm's and csm's to compute certain kinds of functions.

Definition 3.9. A function $f\colon \Sigma^* \to \Delta^*$ is a *gsm* (resp. *csm*) *mapping* iff there is a gsm (resp. csm) M so that $M(w) = f(w)$, for all $w \in \Sigma^*$.

Remark: Ginsburg and Rose (1966) characterized gsm mappings as follows:

A function $f\colon \Sigma^* \to \Delta^*$ is a gsm mapping iff

(a) f preserves prefixes, i.e., $f(x)$ is a prefix of $f(xy)$;

(b) There is an integer, m, such that for all $w \in \Sigma^*$ and all $a \in \Sigma$, we have $|f(wa)| - |f(w)| \leq m$;

(c) $f(\epsilon) = \epsilon$;

(d) For every regular language, $R \subseteq \Delta^*$, the language $f^{-1}(R) = \{w \in \Sigma^* \mid f(w) \in R\}$ is regular.

A function $f\colon \Sigma^* \to \Delta^*$ is a csm mapping iff f satisfies (a) and (d), and for all $w \in \Sigma^*$, $|f(w)| = |w|$.

The following proposition is left as a homework problem.

Proposition 3.1. *The family of regular languages (over an alphabet Σ) is closed under both gsm and inverse gsm mappings.*

We can generalize the gsm model so that

(1) the device is nondeterministic,

(2) the device has a set of accepting states,

(3) transitions are allowed to occur without new input being processed,

(4) transitions are defined for input strings instead of individual letters.

Here is the definition of such a model, the *a-transducer*. A much more powerful model of transducer will be investigated later: the *Turing machine*.

Definition 3.10. An *a-transducer* (or *nondeterministic sequential transducer with accepting states*) is a sextuple $M = (K, \Sigma, \Delta, \lambda, q_0, F)$, where

(1) K is a finite set of *states*,

(2) Σ is a finite *input alphabet*,

(3) Δ is a finite *output alphabet*,

(4) $q_0 \in K$ is the *start* (or *initial*) *state*,

(5) $F \subseteq K$ is the set of *accepting* (of *final*) *states* and

(6) $\lambda \subseteq K \times \Sigma^* \times \Delta^* \times K$ is a finite set of quadruples called the *transition function* of M.

If $\lambda \subseteq K \times \Sigma^* \times \Delta^+ \times K$, then M is *ϵ-free*

Clearly, a gsm is a special kind of a-transducer.

An a-transducer defines a binary relation between Σ^* and Δ^*, or equivalently, a function $M\colon \Sigma^* \to 2^{\Delta^*}$.

We can explain what this function is by describing how an a-transducer makes a sequence of moves from configurations to configurations.

The current *configuration* of an a-transducer is described by a triple

$$(p, u, v) \in K \times \Sigma^* \times \Delta^*,$$

where p is the current state, u is the remaining input, and v is some ouput produced so far.

We define the binary relation $\vdash_M$ on $K \times \Sigma^* \times \Delta^*$ as follows: For all $p, q \in K$, $u, \alpha \in \Sigma^*$, $\beta, v \in \Delta^*$, if $(p, u, v, q) \in \lambda$, then

$$(p,\, u\alpha,\, \beta) \vdash_M (q,\, \alpha,\, \beta v).$$

Let $\vdash_M^*$ be the transitive and reflexive closure of $\vdash_M$.

The function $M\colon \Sigma^* \to 2^{\Delta^*}$ is defined such that for every $w \in \Sigma^*$,

$$M(w) = \{y \in \Delta^* \mid (q_0,\, w,\, \epsilon) \vdash_M^* (f,\, \epsilon,\, y),\ f \in F\}.$$

For any language $L \subseteq \Sigma^*$ let

$$M(L) = \bigcup_{w \in L} M(w).$$

For any $y \in \Delta^*$, let

$$M^{-1}(y) = \{w \in \Sigma^* \mid y \in M(w)\}$$

and for any language $L' \subseteq \Delta^*$, let

$$M^{-1}(L') = \bigcup_{y \in L'} M^{-1}(y).$$

Remark: Notice that if $w \in M^{-1}(L')$, then there exists some $y \in L'$ such that $w \in M^{-1}(y)$, i.e.,
$y \in M(w)$. This **does not** imply that $M(w) \subseteq L'$, only that $M(w) \cap L' \neq \emptyset$.

One should realize that for any $L' \subseteq \Delta^*$ and any a-transducer, M, there is some a-transducer, M', (from Δ^* to 2^{Σ^*}) so that $M'(L') = M^{-1}(L')$.

The following proposition is left as a homework problem:

Proposition 3.2. *The family of regular languages (over an alphabet Σ) is closed under both a-transductions and inverse a-transductions.*

3.7 An Application of NFA's: Text Search

A common problem in the age of the Web (and on-line text repositories) is the following:

Given a set of words, called the *keywords*, find all the documents that contain one (or all) of those words.

Search engines are a popular example of this process. Search engines use *inverted indexes* (for each word appearing on the Web, a list of all the places where that word occurs is stored).

However, there are applications that are unsuited for inverted indexes, but are good for automaton-based techniques.

Some text-processing programs, such as advanced forms of the UNIX `grep` command (such as `egrep` or `fgrep`) are based on automaton-based techniques.

The characteristics that make an application suitable for searches that use automata are:

(1) The repository on which the search is conducted is rapidly changing.

(2) The documents to be searched cannot be catalogued. For example, Amazon.com creates pages "on the fly" in response to queries.

We can use an NFA to find occurrences of a set of keywords in a text. This NFA signals by entering a final state that it has seen one of the keywords. The form of such an NFA is special.

(1) There is a start state, q_0, with a transition to itself on every input symbol from the alphabet, Σ.

(2) For each keyword, $w = w_1 \cdots w_k$ (with $w_i \in \Sigma$), there are k states, $q_1^{(w)}, \ldots, q_k^{(w)}$, and there is a transition from q_0 to $q_1^{(w)}$ on input w_1, a transition from $q_1^{(w)}$ to $q_2^{(w)}$ on input w_2, and so on, until a transition from $q_{k-1}^{(w)}$ to $q_k^{(w)}$ on input w_k. The state $q_k^{(w)}$ is an accepting state and indicates that the keyword $w = w_1 \cdots w_k$ has been found.

The NFA constructed above can then be converted to a DFA using the subset construction.

Here is an example where $\Sigma = \{a, b\}$ and the set of keywords is

$$\{aba,\ \ ab,\ \ ba\}.$$

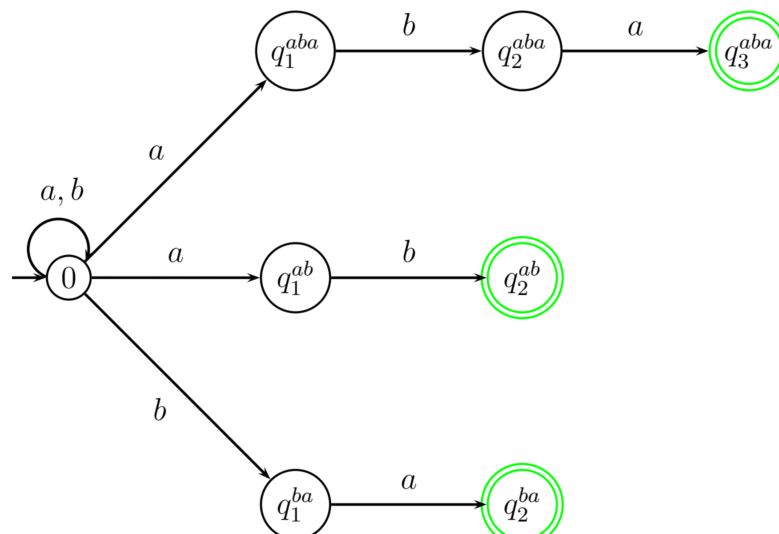


Figure 3.10: NFA for the keywords aba, ab, ba.

Applying the subset construction to the NFA, we obtain the DFA whose transition table is:

		a	b
0	0	1	2
1	$0, q_1^{aba}, q_1^{ab}$	1	3
2	$0, q_1^{ba}$	4	2
3	$0, q_1^{ba}, q_2^{aba}, q_2^{ab}$	5	2
4	$0, q_1^{aba}, q_1^{ab}, q_2^{ba}$	1	3
5	$0, q_1^{aba}, q_1^{ab}, q_2^{ba}, q_3^{aba}$	1	3

The final states are: 3, 4, 5.

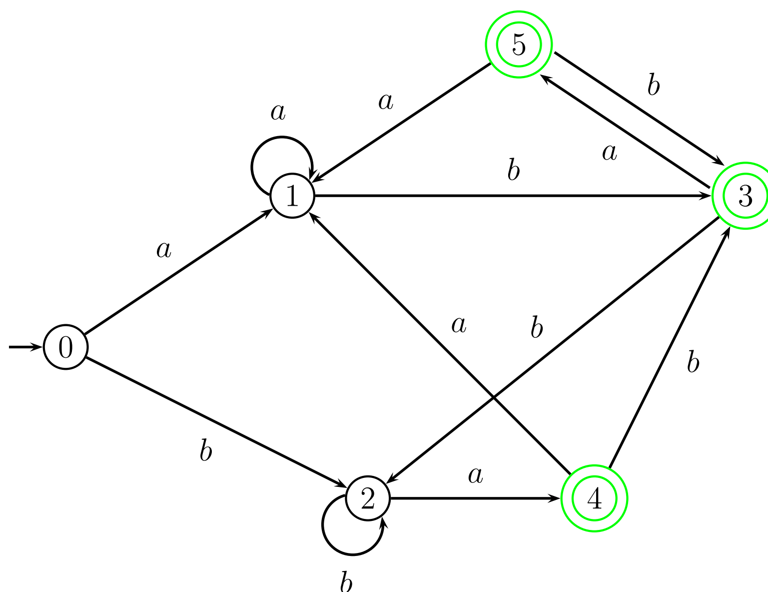


Figure 3.11: DFA for the keywords aba, ab, ba.

The good news news is that, due to the very special structure of the NFA, the number of states of the corresponding DFA is *at most* the number of states of the original NFA!

We find that the states of the DFA are (check it yourself!):

(1) The set $\{q_0\}$, associated with the start state q_0 of the NFA.

(2) For any state $p \neq q_0$ of the NFA reached from q_0 along a path corresponding to a string $u = u_1 \cdots u_m$, the set consisting of:

(a) q_0

(b) p

(c) The set of all states q of the NFA reachable from q_0 by following a path whose symbols form a nonempty suffix of u, i.e., a string of the form
$u_j u_{j+1} \cdots u_m$.

As a consequence, we get an efficient (w.r.t. time and space) method to recognize a set of keywords. In fact, this DFA recognizes leftmost occurrences of keywords in a text (we can stop as soon as we enter a final state).

Chapter 4

Hidden Markov Models (HMMs)

4.1 Hidden Markov Models (HMMs)

There is a variant of the notion of DFA with ouput, for example a transducer such as a gsm (generalized sequential machine), which is widely used in machine learning. This machine model is known as *hidden Markov model*, for short *HMM*. These notes are only an *introduction* to HMMs and are by no means complete. For more comprehensive presentations of HMMs, see the references at the end of this chapter.

There are three new twists compared to traditional gsm models:

(1) There is a finite set of states Q with n elements, a bijection $\sigma\colon Q \to \{1, \ldots, n\}$, and the transitions between states are labeled with probabilities rather that symbols from an alphabet. For any two states p and q in Q, the edge from p to q is labeled with a probability $A(i, j)$, with $i = \sigma(p)$ and $j = \sigma(q)$. The probabilities $A(i, j)$ form an $n \times n$ matrix $A = (A(i, j))$.

(2) There is a finite set $\mathbb{O}$ of size m (called the *observation space*) of possible outputs that can be emitted, a bijection $\omega\colon \mathbb{O} \to \{1, \ldots, m\}$, and for every state $q \in Q$, there is a probability $B(i, j)$ that output $O \in \mathbb{O}$ is emitted (produced), with $i = \sigma(q)$ and $j = \omega(O)$. The probabilities $B(i, j)$ form an $n \times m$ matrix $B = (B(i, j))$.

(3) *Sequences of outputs* $\mathcal{O} = (O_1, \ldots, O_T)$ (with $O_t \in \mathbb{O}$ for $t = 1, \ldots, T$) emitted by the model are *directly observable*, but the sequences of states $\mathcal{S} = (q_1, \ldots, q_T)$ (with $q_t \in Q$ for $t = 1, \ldots, T$) that caused some sequence of output to be emitted are *not observable*. In this sense the states are hidden, and this is the reason for calling this model a *hidden Markov model.*

Remark: We could define a state transition probability function $\mathbb{A}\colon Q \times Q \to [0, 1]$ by $\mathbb{A}(p, q) = A(\sigma(p), \sigma(q))$, and a state observation probability function $\mathbb{B}\colon Q \times \mathbb{O} \to [0, 1]$ by $\mathbb{B}(p, O) = B(\sigma(p), \omega(O))$. The function $\mathbb{A}$ conveys exactly the same amount of information

as the matrix A, and the function $\mathbb{B}$ conveys exactly the same amount of information as the matrix B. The only difference is that the arguments of $\mathbb{A}$ are states rather than integers, so in that sense it is perhaps more natural. We can think of A as an implementation of $\mathbb{A}$. Similarly, the arguments of $\mathbb{B}$ are states and outputs rather than integers. Again, we can think of B as an implementation of $\mathbb{B}$. Most of the literature is rather sloppy about this. We will use matrices.

Before going any further, we wish to address a notational issue that everyone who writes about state-processes faces. This issue is a bit of a headache which needs to be resolved to avoid a lot of confusion.

The issue is how to denote the states, the ouputs, as well as (ordered) sequences of states and sequences of output. In most problems, states and outputs have "meaningful" names. For example, if we wish to describe the evolution of the temperature from day to day, it makes sense to use two states "Cold" and "Hot," and to describe whether a given individual has a drink by "D," and no drink by "N." Thus our set of states is $Q = \{\mathrm{Cold}, \mathrm{Hot}\}$, and our set of outputs is $\mathbb{O} = \{\mathrm{N}, \mathrm{D}\}$.

However, when computing probabilities, we need to use matrices whose rows and columns are indexed by positive integers, so we need a mechanism to associate a *numerical index* to every state and to every output, and this is the purpose of the bijections $\sigma\colon Q \to \{1, \ldots, n\}$ and $\omega\colon \mathbb{O} \to \{1, \ldots, m\}$. In our example, we define σ by $\sigma(\mathrm{Cold}) = 1$ and $\sigma(\mathrm{Hot}) = 2$, and ω by $\omega(\mathrm{N}) = 1$ and $\omega(\mathrm{D}) = 2$.

Some author circumvent (or do they?) this notational issue by assuming that the set of outputs is $\mathbb{O} = \{1, 2, \ldots, m\}$, and that the set of states is $Q = \{1, 2, \ldots, n\}$. The disadvantage of doing this is that in "real" situations, it is often more convenient to name the outputs and the states with more meaningful names than $1, 2, 3$ *etc.* With respect to this, Mitch Marcus pointed out to me that the task of naming the elements of the output alphabet can be challenging, for example in speech recognition.

Let us now turn to sequences. For example, consider the sequence of six states (from the set $Q = \{\mathrm{Cold}, \mathrm{Hot}\}$),

$$\mathcal{S} = (\mathrm{Cold}, \mathrm{Cold}, \mathrm{Hot}, \mathrm{Cold}, \mathrm{Hot}, \mathrm{Hot}).$$

Using the bijection $\sigma\colon \{\mathrm{Cold}, \mathrm{Hot}\} \to \{1, 2\}$ defined above, the sequence $\mathcal{S}$ is completely determined by the sequence of indices

$$\sigma(\mathcal{S}) = (\sigma(\mathrm{Cold}), \sigma(\mathrm{Cold}), \sigma(\mathrm{Hot}), \sigma(\mathrm{Cold}), \sigma(\mathrm{Hot}), \sigma(\mathrm{Hot})) = (1, 1, 2, 1, 2, 2).$$

More generally, we will denote a sequence of length $T \geq 1$ of states from a set Q of size n by

$$\mathcal{S} = (q_1, q_2, \ldots, q_T),$$

with $q_t \in Q$ for $t = 1, \ldots, T$. Using the bijection $\sigma\colon Q \to \{1, \ldots, n\}$, the sequence $\mathcal{S}$ is completely determined by the sequence of indices

$$\sigma(\mathcal{S}) = (\sigma(q_1), \sigma(q_2), \ldots, \sigma(q_T)),$$

where $\sigma(q_t)$ is some index from the set $\{1, \ldots, n\}$, for $t = 1, \ldots, T$. The problem now is, *what is a better notation for the index denoted by $\sigma(q_t)$?*

Of course, we could use $\sigma(q_t)$, but this is a heavy notation, so *we adopt the notational convention to denote the index $\sigma(q_t)$ by i_t.*[1]

Going back to our example

$$\mathcal{S} = (q_1, q_2, q_3, q_4, q_4, q_6) = (\mathrm{Cold}, \mathrm{Cold}, \mathrm{Hot}, \mathrm{Cold}, \mathrm{Hot}, \mathrm{Hot}),$$

we have

$$\sigma(\mathcal{S}) = (\sigma(q_1), \sigma(q_2), \sigma(q_3), \sigma(q_4), \sigma(q_5), \sigma(q_6)) = (1, 1, 2, 1, 2, 2),$$

so the sequence of indices $(i_1, i_2, i_3, i_4, i_5, i_6) = (\sigma(q_1), \sigma(q_2), \sigma(q_3), \sigma(q_4), \sigma(q_5), \sigma(q_6))$ is given by

$$\sigma(\mathcal{S}) = (i_1, i_2, i_3, i_4, i_5, i_6) = (1, 1, 2, 1, 2, 2).$$

So, the fourth index i_4 is has the value 1.

We apply a similar convention to sequences of outputs. For example, consider the sequence of six outputs (from the set $\mathbb{O} = \{\mathrm{N}, \mathrm{D}\}$),

$$\mathcal{O} = (\mathrm{N}, \mathrm{D}, \mathrm{N}, \mathrm{N}, \mathrm{N}, \mathrm{D}).$$

Using the bijection $\omega\colon \{\mathrm{N}, \mathrm{D}\} \to \{1, 2\}$ defined above, the sequence $\mathcal{O}$ is completely determined by the sequence of indices

$$\omega(\mathcal{O}) = (\omega(\mathrm{N}), \omega(\mathrm{D}), \omega(\mathrm{N}), \omega(\mathrm{N}), \omega(\mathrm{N}), \omega(\mathrm{D})) = (1, 2, 1, 1, 1, 2).$$

More generally, we will denote a sequence of length $T \geq 1$ of outputs from a set $\mathbb{O}$ of size m by

$$\mathcal{O} = (O_1, O_2, \ldots, O_T),$$

with $O_t \in \mathbb{O}$ for $t = 1, \ldots, T$. Using the bijection $\omega\colon \mathbb{O} \to \{1, \ldots, m\}$, the sequence $\mathcal{O}$ is completely determined by the sequence of indices

$$\omega(\mathcal{O}) = (\omega(O_1), \omega(O_2), \ldots, \omega(O_T)),$$

where $\omega(O_t)$ is some index from the set $\{1, \ldots, m\}$, for $t = 1, \ldots, T$. This time, *we adopt the notational convention to denote the index $\omega(O_t)$ by ω_t.*

Going back to our example

$$\mathcal{O} = (O_1, O_2, O_3, O_4, O_5, O_6) = (\mathrm{N}, \mathrm{D}, \mathrm{N}, \mathrm{N}, \mathrm{N}, \mathrm{D}),$$

[1] We contemplated using the notation σ_t for $\sigma(q_t)$ instead of i_t. However, we feel that this would deviate too much from the common practice found in the literature, which uses the notation i_t. This is not to say that the literature is free of horribly confusing notation!

we have

$$\omega(\mathcal{O}) = (\omega(O_1), \omega(O_2), \omega(O_3), \omega(O_4), \omega(O_5), \omega(O_6)) = (1, 2, 1, 1, 1, 2),$$

so the sequence of indices $(\omega_1, \omega_2, \omega_3, \omega_4, \omega_5, \omega_6) = (\omega(O_1), \omega(O_2), \omega(O_3), \omega(O_4), \omega(O_5), \omega(O_6))$ is given by

$$\omega(\mathcal{O}) = (\omega_1, \omega_2, \omega_3, \omega_4, \omega_5, \omega_6) = (1, 2, 1, 1, 1, 2).$$

Remark: What is very confusing is this: to assume that our state set is $Q = \{q_1, \ldots, q_n\}$, and to denote a sequence of states of length T as $\mathcal{S} = (q_1, q_2, \ldots, q_T)$. The symbol q_1 in the sequence $\mathcal{S}$ may actually refer to q_3 in Q, *etc.*

We feel that the explicit introduction of the bijections $\sigma\colon Q \to \{1, \ldots, n\}$ and $\omega\colon \mathbb{O} \to \{1, \ldots, m\}$, although not standard in the literature, yields a mathematically clean way to deal with sequences which is not too cumbersome, although this latter point is a matter of taste.

HMM's are among the most effective tools to solve the following types of problems:

(1) **DNA and protein sequence alignment** in the face of mutations and other kinds of evolutionary change.

(2) **Speech understanding**, also called **Automatic speech recognition**. When we talk, our mouths produce sequences of sounds from the sentences that we want to say. This process is complex. Multiple words may map to the same sound, words are pronounced differently as a function of the word before and after them, we all form sounds slightly differently, and so on. All a listener can hear (perhaps a computer system) is the sequence of sounds, and the listener would like to reconstruct the mapping (backward) in order to determine what words we were attempting to say. For example, when you "talk to your TV" to pick a program, say *game of thrones*, you don't want to get *Jessica Jones*.

(3) **Optical character recognition (OCR)**. When we write, our hands map from an idealized symbol to some set of marks on a page (or screen). The marks are observable, but the process that generates them isn't. A system performing OCR, such as a system used by the post office to read addresses, must discover which word is most likely to correspond to the mark it reads.

Here is an example illustrating the notion of HMM.

Example 4.1. Say we consider the following behavior of some professor at some university. On a hot day (denoted by Hot), the professor comes to class with a drink (denoted D) with probability 0.7, and with no drink (denoted N) with probability 0.3. On the other hand, on

a cold day (denoted Cold), the professor comes to class with a drink with probability 0.2, and with no drink with probability 0.8.

Suppose a student intrigued by this behavior recorded a sequence showing whether the professor came to class with a drink or not, say NNND. Several months later, the student would like to know whether the weather was hot or cold the days he recorded the drinking behavior of the professor.

Now the student heard about machine learning, so he constructs a probabilistic (hidden Markov) model of the weather. Based on some experiments, he determines the probability of going from a hot day to another hot day to be 0.75, the probability of going from a hot to a cold day to be 0.25, the probability of going from a cold day to another cold day to be 0.7, and the probability of going from a cold day to a hot day to be 0.3. He also knows that when he started his observations, it was a cold day with probability 0.45, and a hot day with probability 0.55.

In this example, the set of states is $Q = \{\text{Cold}, \text{Hot}\}$, and the set of outputs is $\mathbb{O} = \{\text{N}, \text{D}\}$. We have the bijection $\sigma\colon \{\text{Cold}, \text{Hot}\} \to \{1, 2\}$ given by $\sigma(\text{Cold}) = 1$ and $\sigma(\text{Hot}) = 2$, and the bijection $\omega\colon \{\text{N}, \text{D}\} \to \{1, 2\}$ given by $\omega(\text{N}) = 1$ and $\omega(\text{D}) = 2$

The above data determine an HMM depicted in Figure 4.1.

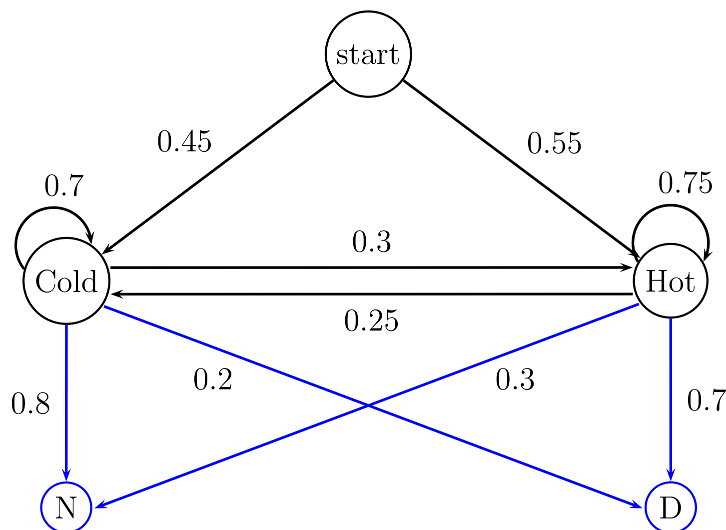


Figure 4.1: Example of an HMM modeling the “drinking behavior” of a professor at the University of Pennsylvania.

The portion of the state diagram involving the states Cold, Hot, is analogous to an NFA in which the transition labels are probabilities; it is the underlying Markov model of the HMM. For any given state, the probabilities on the outgoing edges sum to 1. The start state is a convenient way to express the probabilities of starting either in state Cold or in state

Hot. Also, from each of the states Cold and Hot, we have emission probabilities of producing the ouput N or D, and these probabilities also sum to 1.

We can also express these data using matrices. The matrix

$$A = \begin{pmatrix} 0.7 & 0.3 \\ 0.25 & 0.75 \end{pmatrix}$$

describes the transitions of the Makov model, the vector

$$\pi = \begin{pmatrix} 0.45 \\ 0.55 \end{pmatrix}$$

describes the probabilities of starting either in state Cold or in state Hot, and the matrix

$$B = \begin{pmatrix} 0.8 & 0.2 \\ 0.3 & 0.7 \end{pmatrix}$$

describes the emission probabilities. Observe that the rows of the matrices A and B sum to 1. Such matrices are called *row-stochastic matrices*. The entries in the vector π also sum to 1.

The student would like to solve what is known as the *decoding problem*. Namely, given the output sequence NNND, find the *most likely state sequence* of the Markov model that produces the output sequence NNND. Is it $(\text{Cold}, \text{Cold}, \text{Cold}, \text{Cold})$, or $(\text{Hot}, \text{Hot}, \text{Hot}, \text{Hot})$, or $(\text{Hot}, \text{Cold}, \text{Cold}, \text{Hot})$, or $(\text{Cold}, \text{Cold}, \text{Cold}, \text{Hot})$? Given the probabilities of the HMM, it seems unlikely that it is $(\text{Hot}, \text{Hot}, \text{Hot}, \text{Hot})$, but how can we find the most likely one?

Let us consider another example taken from Stamp [19].

Example 4.2. Suppose we want to determine the average annual temperature at a particular location over a series of years in a distant past where thermometers did not exist. Since we can't go back in time, we look for indirect evidence of the temperature, say in terms of the size of tree growth rings. For simplicity, assume that we consider the two temperatures Cold and Hot, and three different sizes of tree rings: small, medium and large, which we denote by S, M, L.

In this example, the set of states is $Q = \{\text{Cold}, \text{Hot}\}$, and the set of outputs is $\mathbb{O} = \{\text{S}, \text{M}, \text{L}\}$. We have the bijection $\sigma\colon \{\text{Cold}, \text{Hot}\} \to \{1, 2\}$ given by $\sigma(\text{Cold}) = 1$ and $\sigma(\text{Hot}) = 2$, and the bijection $\omega\colon \{\text{S}, \text{M}, \text{L}\} \to \{1, 2, 3\}$ given by $\omega(\text{S}) = 1$, $\omega(\text{M}) = 2$, and $\omega(\text{L}) = 3$. The HMM shown in Figure 4.2 is a model of the situation.

Suppose we observe the sequence of tree growth rings (S, M, S, L). What is the most likely sequence of temperatures over a four-year period which yields the observations (S, M, S, L)?

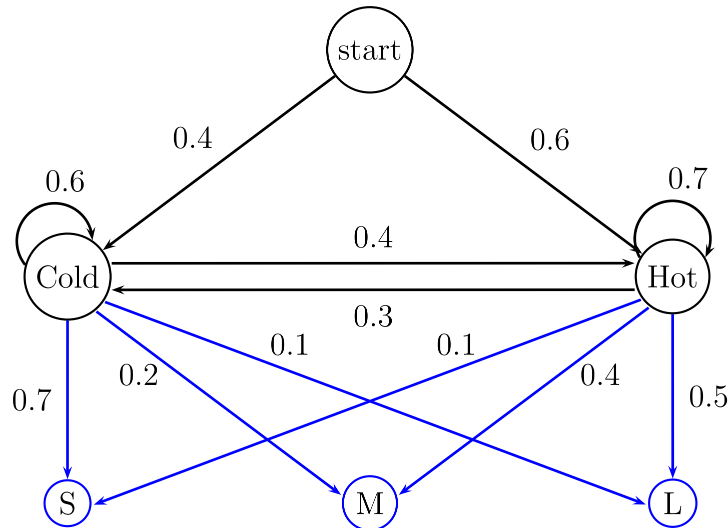


Figure 4.2: Example of an HMM modeling the temperature in terms of tree growth rings.

Going back to Example 4.1, we need to figure out the probability that a sequence of states $\mathcal{S} = (q_1, q_2, \ldots, q_T)$ produces the output sequence $\mathcal{O} = (O_1, O_2, \ldots, O_T)$. Then the probability that we want is just the product of the probability that we begin with state q_1, times the product of the probabilities of each of the transitions, times the product of the emission probabilities. With our notational conventions, $\sigma(q_t) = i_t$ and $\omega(O_t) = \omega_t$, so we have

$$\mathsf{Pr}(\mathcal{S}, \mathcal{O}) = \pi(i_1) B(i_1, \omega_1) \prod_{t=2}^{T} A(i_{t-1}, i_t) B(i_t, \omega_t).$$

In our example, $\omega(\mathcal{O}) = (\omega_1, \omega_2, \omega_3, \omega_4) = (1, 1, 1, 2)$, which corresponds to NNND. The brute-force method is to compute these probabilities for all $2^4 = 16$ sequences of states of length 4 (in general, there are n^T sequences of length T). For example, for the sequence $\mathcal{S} = (\text{Cold}, \text{Cold}, \text{Cold}, \text{Hot})$, associated with the sequence of indices $\sigma(\mathcal{S}) = (i_1, i_2, i_3, i_4) = (1, 1, 1, 2)$, we find that

$$\begin{aligned}\mathsf{Pr}(\mathcal{S}, \text{NNND}) &= \pi(1) B(1,1) A(1,1) B(1,1) A(1,1) B(1,1) A(1,2) B(2,2) \\ &= 0.45 \times 0.8 \times 0.7 \times 0.8 \times 0.7 \times 0.8 \times 0.3 \times 0.7 = 0.0237.\end{aligned}$$

A much more efficient way to proceed is to use a method based on *dynamic programming*. Recall the bijection $\sigma \colon \{\text{Cold}, \text{Hot}\} \to \{1, 2\}$, so that we will refer to the state Cold as 1, and to the state Hot as 2. For $t = 1, 2, 3, 4$, for every state $i = 1, 2$, we compute $\mathit{score}(i, t)$ to be the highest probability that a sequence of length t ending in state i produces the output sequence $(O_1, \ldots, O_t)$, and for $t \geq 2$, we let $\mathit{pred}(i, t)$ be the state that precedes state i in a best sequence of length t ending in i.

Recall that in our example, $\omega(\mathcal{O}) = (\omega_1, \omega_2, \omega_3, \omega_4) = (1, 1, 1, 2)$, which corresponds to NNND. Initially, we set

$$score(j, 1) = \pi(j)B(j, \omega_1), \quad j = 1, 2,$$

and since $\omega_1 = 1$ we get $score(1, 1) = 0.45 \times 0.8 = 0.36$, which is the probability of starting in state Cold and emitting N, and $score(2, 1) = 0.55 \times 0.3 = 0.165$, which is the probability of starting in state Hot and emitting N.

Next we compute $score(1, 2)$ and $score(2, 2)$ as follows. For $j = 1, 2$, for $i = 1, 2$, compute temporary scores

$$tscore(i, j) = score(i, 1)A(i, j)B(j, \omega_2);$$

then pick the best of the temporary scores,

$$score(j, 2) = \max_i tscore(i, j).$$

Since $\omega_2 = 1$, we get $tscore(1, 1) = 0.36\times 0.7\times 0.8 = 0.2016$, $tscore(2, 1) = 0.165\times 0.25\times 0.8 = 0.0330$, and $tscore(1, 2) = 0.36\times 0.3\times 0.3 = 0.0324$, $tscore(2, 2) = 0.165\times 0.75\times 0.3 = 0.0371$. Then

$$score(1, 2) = \max\{tscore(1, 1), tscore(2, 1)\} = \max\{0.2016, 0.0330\} = 0.2016,$$

which is the largest probability that a sequence of two states emitting the output (N, N) ends in state Cold, and

$$score(2, 2) = \max\{tscore(1, 2), tscore(2, 2)\} = \max\{0.0324, 0.0371\} = 0.0371.$$

which is the largest probability that a sequence of two states emitting the output (N, N) ends in state Hot. Since the state that leads to the optimal score score$(1, 2)$ is 1, we let $pred(1, 2) = 1$, and since the state that leads to the optimal score score$(2, 2)$ is 2, we let $pred(2, 2) = 2$.

We compute $score(1, 3)$ and $score(2, 3)$ in a similar way. For $j = 1, 2$, for $i = 1, 2$, compute

$$tscore(i, j) = score(i, 2)A(i, j)B(j, \omega_3);$$

then pick the best of the temporary scores,

$$score(j, 3) = \max_i tscore(i, j).$$

Since $\omega_3 = 1$, we get $tscore(1, 1) = 0.2016 \times 0.7 \times 0.8 = 0.1129$, $tscore(2, 1) = 0.0371 \times 0.25 \times 0.8 = 0.0074$, and $tscore(1, 2) = 0.2016 \times 0.3 \times 0.3 = 0.0181$, $tscore(2, 2) = 0.0371 \times 0.75 \times 0.3 = 0.0083$. Then

$$score(1, 3) = \max\{tscore(1, 1), tscore(2, 1)\} = \max\{0.1129, 0.074\} = 0.1129,$$

which is the largest probability that a sequence of three states emitting the output (N, N, N) ends in state Cold, and

$$score(2,3) = \max\{tscore(1,2), tscore(2,2)\} = \max\{0.0181, 0.0083\} = 0.0181,$$

which is the largest probability that a sequence of three states emitting the output (N, N, N) ends in state Hot. We also get $pred(1,3) = 1$ and $pred(2,3) = 1$. Finally, we compute $score(1,4)$ and $score(2,4)$ in a similar way. For $j = 1, 2$, for $i = 1, 2$, compute

$$tscore(i,j) = score(i,3)A(i,j)B(j,\omega_4);$$

then pick the best of the temporary scores,

$$score(j,4) = \max_i tscore(i,j).$$

Since $\omega_4 = 2$, we get $tscore(1,1) = 0.1129 \times 0.7 \times 0.2 = 0.0158$, $tscore(2,1) = 0.0181 \times 0.25 \times 0.2 = 0.0009$, and $tscore(1,2) = 0.1129 \times 0.3 \times 0.7 = 0.0237$, $tscore(2,2) = 0.0181 \times 0.75 \times 0.7 = 0.0095$. Then

$$score(1,4) = \max\{tscore(1,1), tscore(2,1)\} = \max\{0.0158, 0.0009\} = 0.0158,$$

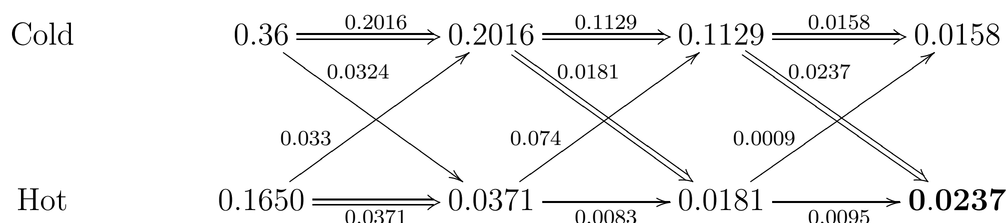
which is the largest probability that a sequence of four states emitting the output (N, N, N, D) ends in state Cold, and

$$score(2,4) = \max\{tscore(1,2), tscore(2,2)\} = \max\{0.0237, 0.0095\} = 0.0237,$$

which is the largest probability that a sequence of four states emitting the output (N, N, N, D) ends in state Hot, and $pred(1,4) = 1$ and $pred(2,4) = 1$

Since $\max\{score(1,4), score(2,4)\} = \max\{0.0158, 0.0237\} = 0.0237$, the state with the maximum score is Hot, and by following the predecessor list (also called backpointer list), we find that the most likely state sequence to produce the output sequence NNND is (Cold, Cold, Cold, Hot).

The stages of the computations of $score(j,t)$ for $i = 1, 2$ and $t = 1, 2, 3, 4$ can be recorded in the following diagram called a *lattice*, or a *trellis* (which means lattice in French!):



Double arrows represent the predecessor edges. For example, the predecessor $pred(2,3)$ of the third node on the bottom row labeled with the score 0.0181 (which corresponds to

Hot), is the second node on the first row labeled with the score 0.2016 (which corresponds to Cold). The two incoming arrows to the third node on the bottom row are labeled with the temporary scores 0.0181 and 0.0083. The node with the highest score at time $t = 4$ is Hot, with score 0.0237 (showed in bold), and by following the double arrows backward from this node, we obtain the most likely state sequence (Cold, Cold, Cold, Hot).

The method we just described is known as the *Viterbi algorithm*. We now define HHM's in general, and then present the Viterbi algorithm.

Definition 4.1. A *hidden Markov model*, for short *HMM*, is a quintuple $M = (Q, \mathbb{O}, \pi, A, B)$ where

- Q is a finite set of *states* with n elements, and there is a bijection $\sigma\colon Q \to \{1, \ldots, n\}$.
- $\mathbb{O}$ is a finite *output alphabet* (also called *set of possible observations*) with m observations, and there is a bijection $\omega\colon \mathbb{O} \to \{1, \ldots, m\}$.
- $A = (A(i, j))$ is an $n \times n$ matrix called the *state transition probability matrix*, with

$$A(i, j) \geq 0, \quad 1 \leq i, j \leq n, \quad \text{and} \quad \sum_{j=1}^{n} A(i, j) = 1, \quad i = 1, \ldots, n.$$

- $B = (B(i, j))$ is an $n \times m$ matrix called the *state observation probability matrix* (also called *confusion matrix*), with

$$B(i, j) \geq 0, \quad 1 \leq i, j \leq n, \quad \text{and} \quad \sum_{j=1}^{m} B(i, j) = 1, \quad i = 1, \ldots, n.$$

  A matrix satisfying the above conditions is said to be *row stochastic*. Both A and B are row-stochastic.

We also need to state the conditions that make M a Markov model. To do this rigorously requires the notion of random variable and is a bit tricky (see the remark below), so we will cheat as follows:

(a) Given any sequence of states $(q_0, \ldots, q_{t-2}, p, q)$, the conditional probability that q is the tth state given that the previous states were $q_0, \ldots, q_{t-2}, p$ is equal to the conditional probability that q is the tth state given that the previous state at time $t - 1$ is p:

$$\mathsf{Pr}(q \mid q_0, \ldots, q_{t-2}, p) = \mathsf{Pr}(q \mid p).$$

This is the *Markov property*. Informally, the "next" state q of the process at time t is independent of the "past" states $q_0, \ldots, q_{t-2}$, provided that the "present" state p at time $t - 1$ is known.

(b) Given any sequence of states $(q_0, \ldots, q_i, \ldots, q_t)$, and given any sequence of outputs $(O_0, \ldots, O_i, \ldots, O_t)$, the conditional probability that the output O_i is emitted depends only on the state q_i, and not any other states or any other observations:

$$\mathsf{Pr}(O_i \mid q_0, \ldots, q_i, \ldots, q_t, O_0, \ldots, O_i, \ldots, O_t) = \mathsf{Pr}(O_i \mid q_i).$$

This is the *output independence* condition. Informally, the output function is near-sighted.

Examples of HMMs are shown in Figure 4.1, Figure 4.2, and Figure 4.3. Note that an ouput is emitted when visiting a state, not when making a transition, as in the case of a gsm. So the analogy with the gsm model is only partial; it is meant as a motivation for HMMs.

The hidden Markov model was developed by L. E. Baum and colleagues at the Institue for Defence Analysis at Princeton (including Petrie, Eagon, Sell, Soules, and Weiss) starting in 1966.

If we ignore the output components $\mathbb{O}$ and B, then we have what is called a *Markov chain*. A good interpretation of a Markov chain is the evolution over (discrete) time of the populations of n species that may change from one species to another. The probability $A(i, j)$ is the fraction of the population of the ith species that changes to the jth species. If we denote the populations at time t by the row vector $x = (x_1, \ldots, x_n)$, and the populations at time $t + 1$ by $y = (y_1, \ldots, y_n)$, then

$$y_j = A(1, j)x_1 + \cdots + A(i, j)x_i + \cdots + A(n, j)x_n, \quad 1 \leq j \leq n,$$

in matrix form, $y = xA$. The condition $\sum_{j=1}^{n} A(i, j) = 1$ expresses that the total population is preserved, namely $y_1 + \cdots + y_n = x_1 + \cdots + x_n$.

Remark: This remark is intended for the reader who knows some probability theory, and it can be skipped without any negative effect on understanding the rest of this chapter. Given a probability space $(\Omega, \mathcal{F}, \mu)$ and any countable set Q (for simplicity we may assume Q is finite), a *stochastic discrete-parameter process with state space* Q is a countable family $(X_t)_{t \in \mathbb{N}}$ of random variables $X_t \colon \Omega \to Q$. We can think of t as time, and for any $q \in Q$, of $\mathsf{Pr}(X_t = q)$ as the probability that the process X is in state q at time t. If

$$\mathsf{Pr}(X_t = q \mid X_0 = q_0, \ldots, X_{t-2} = q_{t-2}, X_{t-1} = p) = \mathsf{Pr}(X_t = q \mid X_{t-1} = p)$$

for all $q_0, , \ldots, q_{t-2}, p, q \in Q$ and for all $t \geq 1$, and if the probability on the right-hand side is independent of t, then we say that $X = (X_t)_{t \in \mathbb{N}}$ is a *time-homogeneous Markov chain*, for short, *Markov chain*. Informally, the "next" state X_t of the process is independent of the "past" states $X_0, \ldots, X_{t-2}$, provided that the "present" state X_{t-1} is known.

Since for simplicity Q is assumed to be finite, there is a bijection $\sigma \colon Q \to \{1, \ldots, n\}$, and then, the process X is completely determined by the probabilities

$$a_{ij} = \mathsf{Pr}(X_t = q \mid X_{t-1} = p), \quad i = \sigma(p),\ j = \sigma(q),\ p, q \in Q,$$

and if Q is a finite state space of size n, these form an $n \times n$ matrix $A = (a_{ij})$ called the *Markov matrix* of the process X. It is a row-stochastic matrix.

The beauty of Markov chains is that if we write

$$\pi(i) = \mathsf{Pr}(X_0 = i)$$

for the initial probability distribution, then the joint probability distribution of $X_0, X_1, \ldots,$ X_t is given by

$$\mathsf{Pr}(X_0 = i_0, X_1 = i_1, \ldots, X_t = i_t) = \pi(i_0)A(i_0, i_1) \cdots A(i_{t-1}, i_t).$$

The above expression only involves π and the matrix A, and makes no mention of the original measure space. Therefore, it *doesn't matter what the probability space is!*

Conversely, given an $n \times n$ row-stochastic matrix A, let Ω be the set of all countable sequences $\omega = (\omega_0, \omega_1, \ldots, \omega_t, \ldots)$ with $\omega_t \in Q = \{1, \ldots, n\}$ for all $t \in \mathbb{N}$, and let $X_t \colon \Omega \to Q$ be the projection on the tth component, namely $X_t(\omega) = \omega_t$.[2] Then it is possible to define a σ-algebra (also called a σ-field) $\mathcal{B}$ and a measure μ on $\mathcal{B}$ such that $(\Omega, \mathcal{B}, \mu)$ is a probability space, and $X = (X_t)_{t\in\mathbb{N}}$ is a Markov chain with corresponding Markov matrix A.

To define $\mathcal{B}$, proceed as follows. For every $t \in \mathbb{N}$, let $\mathcal{F}_t$ be the family of all unions of subsets of Ω of the form

$$\{\omega \in \Omega \mid (X_0(\omega) \in S_0) \wedge (X_1(\omega) \in S_1) \wedge \cdots \wedge (X_t(\omega) \in S_t)\},$$

where $S_0, S_1, \ldots, S_t$ are subsets of the state space $Q = \{1, \ldots, n\}$. It is not hard to show that each $\mathcal{F}_t$ is a σ-algebra. Then let

$$\mathcal{F} = \bigcup_{t\geq 0} \mathcal{F}_t.$$

Each set in $\mathcal{F}$ is a set of paths for which a finite number of outcomes are restricted to lie in certain subsets of $Q = \{1, \ldots, n\}$. All other outcomes are unrestricted. In fact, every subset C in $\mathcal{F}$ is a countable union

$$C = \bigcup_{i\in\mathbb{N}} B_i^{(t)}$$

of sets of the form

$$\begin{aligned} B_i^{(t)} &= \{\omega \in \Omega \mid \omega = (q_0, q_1, \ldots, q_t, s_{t+1}, \ldots. s_j, \ldots,) \mid q_0, q_1, \ldots, q_t \in Q\} \\ &= \{\omega \in \Omega \mid X_0(\omega) = q_0, X_1(\omega) = q_1, \ldots, X_t(\omega) = q_t\}. \end{aligned}$$

[2]It is customary in probability theory to denote events by the letter ω. In the present case, ω denotes a countable sequence of elements from Q. This notation has nothing do with the bijection $\omega \colon \mathbb{O} \to \{1, \ldots, m\}$ occurring in Definition 4.1.

The sequences in $B_i^{(t)}$ are those beginning with the fixed sequence $(q_0, q_1, \ldots, q_t)$. One can show that $\mathcal{F}$ is a field of sets, but not necessarily a σ-algebra, so we form the smallest σ-algebra $\mathcal{G}$ containing $\mathcal{F}$.

Using the matrix A we can define the measure $\nu(B_i^{(t)})$ as the product of the probabilities along the sequence $(q_0, q_1, \ldots, q_t)$. Then it can be shown that ν can be extended to a measure μ on $\mathcal{G}$, and we let $\mathcal{B}$ be the σ-algebra obtained by adding to $\mathcal{G}$ all subsets of sets of measure zero. The resulting probability space $(\Omega, \mathcal{B}, \mu)$ is usually called the *sequence space*, and the measure μ is called the *tree measure*. Then it is easy to show that the family of random variables $X_t \colon \Omega \to Q$ on the probability space$(\Omega, \mathcal{B}, \mu)$ is a time-homogeneous Markov chain whose Markov matrix is the orginal matrix A. The above construction is presented in full detail in Kemeny, Snell, and Knapp[10] (Chapter 2, Sections 1 and 2).

Most presentations of Markov chains do not even mention the probability space over which the random variables X_t are defined. This makes the whole thing quite mysterious, since the probabilities $\mathsf{Pr}(X_t = q)$ are by definition given by

$$\mathsf{Pr}(X_t = q) = \mu(\{\omega \in \Omega \mid X_t(\omega) = q\}),$$

which requires knowing the measure μ. This is more problematic if we start with a stochastic matrix. What are the random variables X_t, what are they defined on? The above construction puts things on firm grounds.

There are three types of problems that can be solved using HMMs:

(1) **The decoding problem**: Given an HMM $M = (Q, \mathbb{O}, \pi, A, B)$, for any observed output sequence $\mathcal{O} = (O_1, O_2, \ldots, O_T)$ of length $T \geq 1$, find a most likely sequence of states $\mathcal{S} = (q_1, q_2, \ldots, q_T)$ that produces the output sequence $\mathcal{O}$. More precisely, with our notational convention that $\sigma(q_t) = i_t$ and $\omega(O_t) = \omega_t$, this means finding a sequence $\mathcal{S}$ such that the probability

$$\mathsf{Pr}(\mathcal{S}, \mathcal{O}) = \pi(i_1) B(i_1, \omega_1) \prod_{t=2}^{T} A(i_{t-1}, i_t) B(i_t, \omega_t)$$

is maximal. This problem is solved effectively by the *Viterbi algorithm* that we outlined before.

(2) **The evaluation problem**, also called **the likelyhood problem**: Given a finite collection $\{M_1, \ldots, M_L\}$ of HMM's with the same output alphabet O, for any output sequence $\mathcal{O} = (O_1, O_2, \ldots, O_T)$ of length $T \geq 1$, find which model M_ℓ is most likely to have generated $\mathcal{O}$. More precisely, given any model M_k, we compute the probability $tprob_k$ that M_k could have produced $\mathcal{O}$ along any path. Then we pick an HMM M_ℓ for which $tprob_\ell$ is maximal. We will return to this point after having described the Viterbi algoritm. A variation of the Viterbi algorithm called the *forward algorithm* effectively solves the evaluation problem.

(3) **The training problem**, also called **the learning problem**: Given a set $\{\mathcal{O}_1, \ldots, \mathcal{O}_r\}$ of output sequences on the same output alpabet O, usually called a set of *training data*, given Q, find the "best" π, A, and B for an HMM M that produces all the sequences in the training set, in the sense that the HMM $M = (Q, \mathbb{O}, \pi, A, B)$ is the most likely to have produced the sequences in the training set. The technique used here is called *expectation maximization*, or *EM*. It is an iterative method that starts with an initial triple π, A, B, and tries to impove it. There is such an algorithm known as the *Baum-Welch* or *forward-backward algorithm*, but it is beyond the scope of this introduction.

Let us now describe the Viterbi algorithm in more details.

4.2 The Viterbi Algorithm and the Forward Algorithm

Given an HMM $M = (Q, \mathbb{O}, \pi, A, B)$, for any observed output sequence $\mathcal{O} = (O_1, O_2, \ldots, O_T)$ of length $T \geq 1$, we want to find a most likely sequence of states $\mathcal{S} = (q_1, q_2, \ldots, q_T)$ that produces the output sequence $\mathcal{O}$.

Using the bijections $\sigma\colon Q \to \{1, \ldots, n\}$ and $\omega\colon \mathbb{O} \to \{1, \ldots, m\}$, we can work with sequences of indices, and recall that we denote the index $\sigma(q_t)$ associated with the tth state q_t in the sequence $\mathcal{S}$ by i_t, and the index $\omega(O_t)$ associated with the tth output O_t in the sequence $\mathcal{O}$ by ω_t. Then we need to find a sequence $\mathcal{S}$ such that the probability

$$\mathsf{Pr}(\mathcal{S}, \mathcal{O}) = \pi(i_1)B(i_1, \omega_1)\prod_{t=2}^{T} A(i_{t-1}, i_t)B(i_t, \omega_t)$$

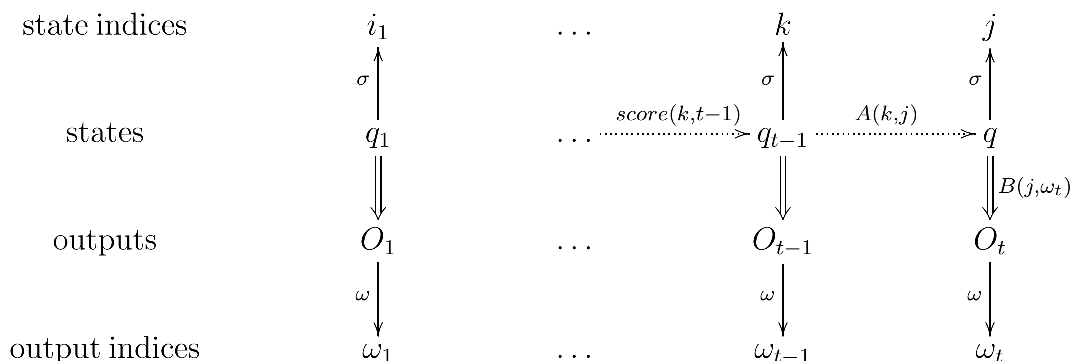
is maximal.

In general, there are n^T sequences of length T. This problem can be solved efficiently by a method based on *dynamic programming*. For any t, $1 \leq t \leq T$, for any state $q \in Q$, if $\sigma(q) = j$, then we compute $score(j, t)$, which is the largest probability that a sequence $(q_1, \ldots, q_{t-1}, q)$ of length t ending with q has produced the output sequence $(O_1, \ldots, O_{t-1}, O_t)$.

The point is that if we know $score(k, t-1)$ for $k = 1, \ldots, n$ (with $t \geq 2$), then we can find $score(j, t)$ for $j = 1, \ldots, n$, because if we write $k = \sigma(q_{t-1})$ and $j = \sigma(q)$ (recall that $\omega_t = \omega(O_t)$), then the probability associated with the path $(q_1, \ldots, q_{t-1}, q)$ is

$$tscore(k, j) = score(k, t-1)A(k, j)B(j, \omega_t).$$

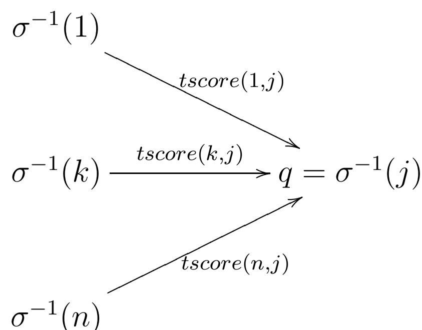
See the illustration below:



So to maximize this probability, we just have to find the maximum of the probabilities $tscore(k, j)$ over all k, that is, we must have

$$score(j, t) = \max_k tscore(k, j).$$

See the illustration below:



To get started, we set $score(j, 1) = \pi(j)B(j, \omega_1)$ for $j = 1, \ldots, n$.

The algorithm goes through a forward phase for $t = 1, \ldots, T$, during which it computes the probabilities $score(j, t)$ for $j = 1, \ldots, n$. When $t = T$, we pick an index j such that $score(j, T)$ is maximal. The machine learning community is fond of the notation

$$j = \arg\max_k score(k, T)$$

to express the above fact. Typically, the *smallest index* j corresponding the maximum element in the list of probabilities

$$(score(1, T), score(2, T), \ldots, score(n, T))$$

is returned. This gives us the last state $q_T = \sigma^{-1}(j)$ in an optimal sequence that yields the output sequence $\mathcal{O}$.

The algorithm then goes through a path retrieval phase. To do this, when we compute

$$score(j,t) = \max_k tscore(k,j),$$

we also record the index $k = \sigma(q_{t-1})$ of the state q_{t-1} in the best sequence $(q_1, \ldots, q_{t-1}, q_t)$ for which $tscore(k,j)$ is maximal (with $j = \sigma(q_t)$), as $pred(j,t) = k$. The index k is often called the *backpointer* of j at time t. This index may not be unique, we just pick one of them. Again, this can be expressed by

$$pred(j,t) = \arg\max_k tscore(k,j).$$

Typically, the *smallest index* k corresponding the maximum element in the list of probabilities

$$(tscore(1,j), tscore(2,j), \ldots, tscore(n,j))$$

is returned.

The predecessors $pred(j,t)$ are only defined for $t = 2, \ldots, T$, but we can let $pred(j,1) = 0$.

Observe that the path retrieval phase of the Viterbi algorithm is very similar to the phase of Dijkstra's algorithm for finding a shortest path that follows the *prev* array. One should not confuse this phase with what is called the *backward algorithm*, which is used in solving the learning problem. The forward phase of the Viterbi algorithm is quite different from the Dijkstra's algorithm, and the Viterbi algorithm is actually simpler (it computes $score(j,t)$ for all states and for $t = 1, \ldots, T$), whereas Dijkstra's algorithm maintains a list of unvisited vertices, and needs to pick the next vertex). The major difference is that the Viterbi algorithm *maximizes a product* of weights along a path, but Dijkstra's algorithm *minimizes a sum* of weights along a path. Also, the Viterbi algorithm knows the length of the path (T) ahead of time, but Dijkstra's algorithm does not.

The Viterbi algorithm, invented by Andrew Viterbi in 1967, is shown below.

The input to the algorithm is $M = (Q, \mathbb{O}, \pi, A, B)$ and the sequence of indices $\omega(\mathcal{O}) = (\omega_1, \ldots, \omega_T)$ associated with the observed sequence $\mathcal{O} = (O_1, O_2, \ldots, O_T)$ of length $T \geq 1$, with $\omega_t = \omega(O_t)$ for $t = 1, \ldots, T$.

The output is a sequence of states $(q_1, \ldots, q_T)$. This sequence is determined by the sequence of indices $(I_1, \ldots, I_T)$; namely, $q_t = \sigma^{-1}(I_t)$.

The Viterbi Algorithm

begin
  for $j = 1$ **to** n **do**
    $score(j,1) = \pi(j)B(j,\omega_1)$
  endfor;

```
(∗ forward phase to find the best (highest) scores ∗)
  for t = 2 to T do
    for j = 1 to n do
      for k = 1 to n do
        tscore(k) = score(k, t − 1)A(k, j)B(j, ω_t)
      endfor;
      score(j, t) = max_k tscore(k);
      pred(j, t) = arg max_k tscore(k)
    endfor
  endfor;
(∗ second phase to retrieve the optimal path ∗)
  I_T = arg max_j score(j, T);
  q_T = σ^{−1}(I_T);
  for t = T to 2 by −1 do
    I_{t−1} = pred(I_t, t);
    q_{t−1} = σ^{−1}(I_{t−1})
  endfor
end
```

An illustration of the Viterbi algorithm applied to Example 4.1 was presented after Example 4.3. If we run the Viterbi algorithm on the output sequence (S, M, S, L) of Example 4.2, we find that the sequence (Cold, Cold, Cold, Hot) has the highest probability, 0.00282, among all sequences of length four.

One may have noticed that the numbers involved, being products of probabilities, become quite small. Indeed, underflow may arise in dynamic programming. Fortunately, there is a simple way to avoid underflow by taking logarithms. We initialize the algorithm by computing

$$score(j, 1) = \log[\pi(j)] + \log[B(j, \omega_1)],$$

and in the step where *tscore* is computed we use the formula

$$tscore(k) = score(k, t-1) + \log[A(k, j)] + \log[B(j, \omega_t)].$$

It immediately verified that the time complexity of the Viterbi algorithm is $O(n^2T)$.

Let us now to turn to the second problem, the evaluation problem (or likelyhood problem).

This time, given a finite collection $\{M_1, \ldots, M_L\}$ of HMM's with the same output alphabet O, for any observed output sequence $\mathcal{O} = (O_1, O_2, \ldots, O_T)$ of length $T \geq 1$, find which model M_ℓ is most likely to have generated $\mathcal{O}$. More precisely, given any model M_k,

we compute the probability $tprob_k$ that M_k could have produced $\mathcal{O}$ along any sequence of states $\mathcal{S} = (q_1, \ldots, q_T)$. Then we pick an HMM M_ℓ for which $tprob_\ell$ is maximal.

The probability $tprob_k$ that we are seeking is given by

$$\begin{aligned} tprob_k &= \mathsf{Pr}(\mathcal{O}) \\ &= \sum_{(i_1,\ldots,i_T)\in\{1,\ldots,n\}^T} \mathsf{Pr}((q_{i_1}, \ldots, q_{i_T}), \mathcal{O}) \\ &= \sum_{(i_1,\ldots,i_T)\in\{1,\ldots,n\}^T} \pi(i_1)B(i_1, \omega_1) \prod_{t=2}^{T} A(i_{t-1}, i_t)B(i_t, \omega_t), \end{aligned}$$

where $\{1, \ldots, n\}^T$ denotes the set of all sequences of length T consisting of elements from the set $\{1, \ldots, n\}$.

It is not hard to see that a brute-force computation requires $2Tn^T$ multiplications. Fortunately, it is easy to adapt the Viterbi algorithm to compute $tprob_k$ efficiently. Since we are not looking for an explicity path, there is no need for the second phase, and during the forward phase, going from $t-1$ to t, rather than finding the maximum of the scores $tscore(k)$ for $k = 1, \ldots, n$, we just set $score(j, t)$ to the sum over k of the temporary scores $tscore(k)$. At the end, $tprob_k$ is the sum over j of the probabilities $score(j, T)$.

The algorithm solving the evaluation problem known as the *forward algorithm* is shown below.

The input to the algorithm is $M = (Q, \mathbb{O}, \pi, A, B)$ and the sequence of indices $\omega(\mathcal{O}) = (\omega_1, \ldots, \omega_T)$ associated with the observed sequence $\mathcal{O} = (O_1, O_2, \ldots, O_T)$ of length $T \geq 1$, with $\omega_t = \omega(O_t)$ for $t = 1, \ldots, T$. The output is the probability *tprob*.

The Foward Algorithm

begin
  for $j = 1$ **to** n **do**
    $score(j, 1) = \pi(j)B(j, \omega_1)$
  endfor;
  for $t = 2$ **to** T **do**
    for $j = 1$ **to** n **do**
      for $k = 1$ **to** n **do**
        $tscore(k) = score(k, t-1)A(k, j)B(j, \omega_t)$
      endfor;
      $score(j, t) = \sum_k tscore(k)$
    endfor

endfor;
$tprob = \sum_j score(j, T)$
end

We can now run the above algorithm on $M_1, \ldots, M_L$ to compute $tprob_1, \ldots, tprob_L$, and we pick the model M_ℓ for which $tprob_\ell$ is maximum.

As for the Viterbi algorithm, the time complexity of the forward algorithm is $O(n^2 T)$.

Underflow is also a problem with the forward algorithm. At first glance it looks like taking logarithms does not help because there is no simple expression for $\log(x_1 + \cdots + x_n)$ in terms of the $\log x_i$. Fortunately, we can use the *log-sum exp trick* (which I learned from Mitch Marcus), namely the identity

$$\log\left(\sum_{i=1}^n e^{x_i}\right) = a + \log\left(\sum_{i=1}^n e^{x_i - a}\right)$$

for all $x_1, \ldots, x_n \in \mathbb{R}$ and $a \in \mathbb{R}$ (take exponentials on both sides). Then, if we pick $a = \max_{1 \leq i \leq n} x_i$, we get

$$1 \leq \sum_{i=1}^n e^{x_i - a} \leq n,$$

so

$$\max_{1 \leq i \leq n} x_i \leq \log\left(\sum_{i=1}^n e^{x_i}\right) \leq \max_{1 \leq i \leq n} x_i + \log n,$$

which shows that $\max_{1 \leq i \leq n} x_i$ is a good approximation for $\log\left(\sum_{i=1}^n e^{x_i}\right)$. For any positive reals $y_1, \ldots, y_n$, if we let $x_i = \log y_i$, then we get

$$\log\left(\sum_{i=1}^n y_i\right) = \max_{1 \leq i \leq n} \log y_i + \log\left(\sum_{i=1}^n e^{\log(y_i) - a}\right), \quad \text{with} \quad a = \max_{1 \leq i \leq n} \log y_i.$$

We will use this trick to compute

$$\log(score(j, k)) = \log\left(\sum_{k=1}^n e^{\log(tscore(k))}\right) = a + \log\left(\sum_{k=1}^n e^{\log(tscore(k)) - a}\right)$$

with $a = \max_{1 \leq k \leq n} \log(tscore(k))$, where $tscore((k)$ could be very small, but $\log(tscore(k))$ is not, so computing $\log(tscore(k)) - a$ does not cause underflow, and

$$1 \leq \sum_{k=1}^n e^{\log(tscore(k)) - a} \leq n,$$

since $\log(tscore(k)) - a \leq 0$ and one of these terms is equal to zero, so even if some of the terms $e^{\log(tscore(k)) - a}$ are very small, this does not cause any trouble. We will also use this trick to compute $\log(tprob) = \log\left(\sum_{j=1}^n score(j, T)\right)$ in terms of the $\log(score(j, T))$.

We leave it as an exercise to the reader to modify the forward algorithm so that it computes $\log(score(j, t))$ and $\log(tprob)$ using the log-sum exp trick. If you use `Matlab`, then this is quite easy because `Matlab` does a lot of the work for you since it can apply operators such as exp or $\sum$ (sum) to vectors.

Example 4.3. To illustrate the forward algorithm, assume that our observant student also recorded the drinking behavior of a professor at Harvard, and that he came up with the HHM shown in Figure 4.3.

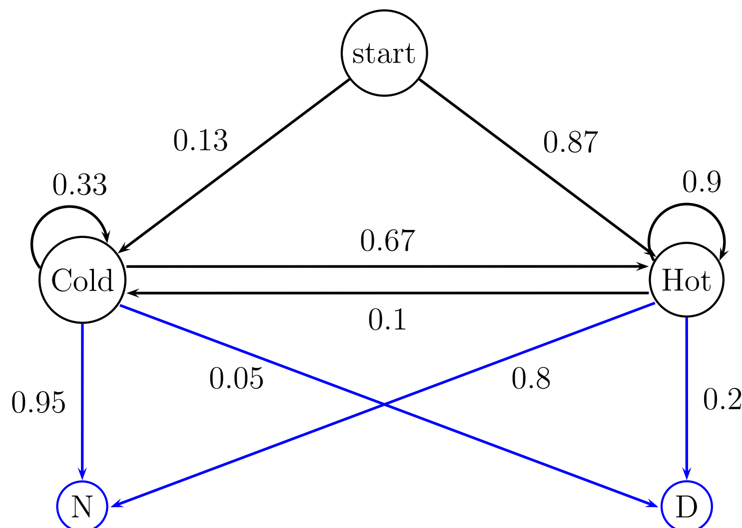


Figure 4.3: Example of an HMM modeling the "drinking behavior" of a professor at Harvard.

However, the student can't remember whether he observed the sequence NNND at Penn or at Harvard. So he runs the forward algorithm on both HMM's to find the most likely model. Do it!

Following Jurafsky, the following chronology shows how of the Viterbi algorithm has had applications in many separate fields.

Citation	**Field**
Viterbi (1967)	information theory
Vintsyuk (1968)	speech processing
Needleman and Wunsch (1970)	molecular biology
Sakoe and Chiba (1971)	speech processing
Sankoff (1972)	molecular biology
Reichert et al. (1973)	molecular biology
Wagner and Fischer (1974)	computer science

Readers who wish to learn more about HMMs should begin with Stamp [19], a great tutorial which contains a very clear and easy to read presentation. Another nice introduction is given in Rich [18] (Chapter 5, Section 5.11). A much more complete, yet accessible, coverage of HMMs is found in Rabiner's tutorial [16]. Jurafsky and Martin's online Chapter 9 (Hidden Markov Models) is also a very good and informal tutorial (see https://web.stanford.edu/˜jurafsky/slp3/9.pdf).

A very clear and quite accessible presentation of Markov chains is given in Cinlar [4]. Another thorough but a bit more advanced presentation is given in Brémaud [3]. Other presentations of Markov chains can be found in Mitzenmacher and Upfal [13], and in Grimmett and Stirzaker [9].

Acknowledgments: I would like to thank Mitch Marcus, Jocelyn Qaintance, and Joao Sedoc, for scrutinizing my work and for many insightful comments.

Chapter 5

Regular Languages and Equivalence Relations, The Myhill-Nerode Characterization, State Equivalence

5.1 Directed Graphs and Paths

It is often useful to view DFA's and NFA's as labeled directed graphs.

Definition 5.1. A *directed graph* is a quadruple $G = (V, E, s, t)$, where V is a set of *vertices, or nodes*, E is a set of *edges, or arcs*, and $s, t \colon E \to V$ are two functions, s being called the *source* function, and t the *target* function. Given an edge $e \in E$, we also call $s(e)$ the *origin* (or *source*) of e, and $t(e)$ the *endpoint* (or *target*) of e.

Remark: the functions s, t need not be injective or surjective. Thus, we allow "isolated vertices."

Example: Let G be the directed graph defined such that

$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$,

$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$, and

$$\begin{array}{llll} s(e_1) = v_1, & s(e_2) = v_2, & s(e_3) = v_3, & s(e_4) = v_4, \\ s(e_5) = v_2, & s(e_6) = v_5, & s(e_7) = v_5, & s(e_8) = v_5, \\ t(e_1) = v_2, & t(e_2) = v_3, & t(e_3) = v_4, & t(e_4) = v_2, \\ t(e_5) = v_5, & t(e_6) = v_5, & t(e_7) = v_6, & t(e_8) = v_6. \end{array}$$

Such a graph can be represented by the following diagram:

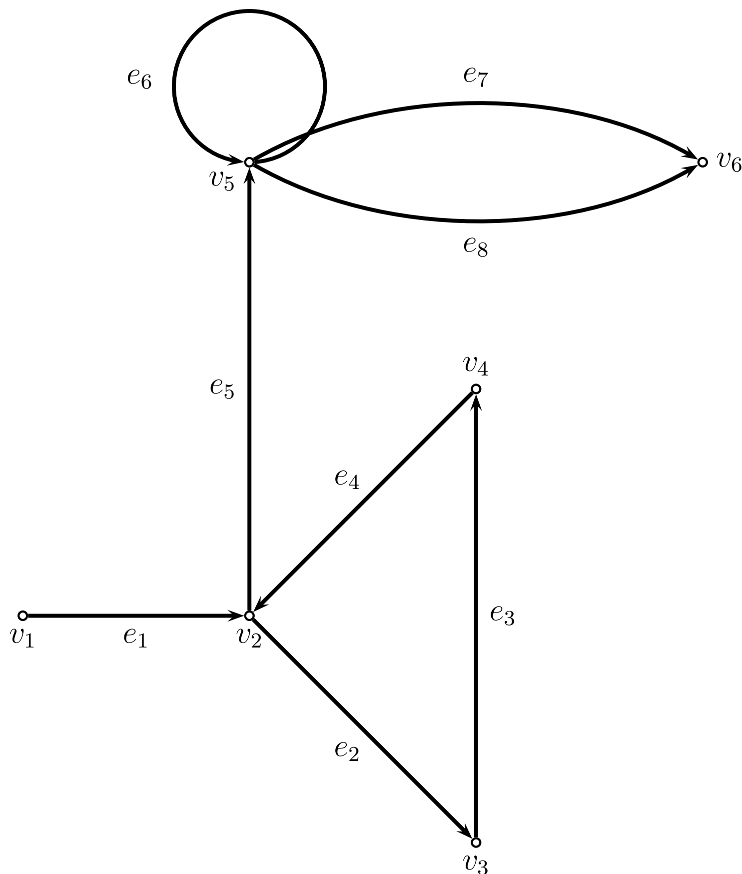


Figure 5.1: A directed graph

In drawing directed graphs, we will usually omit edge names (the e_i), and sometimes even the node names (the v_j).

We now define paths in a directed graph.

Definition 5.2. Given a directed graph $G = (V, E, s, t)$, for any two nodes $u, v \in V$, a *path from u to v* is a triple $\pi = (u, e_1 \ldots e_n, v)$, where $e_1 \ldots e_n$ is a string (sequence) of edges in E such that, $s(e_1) = u$, $t(e_n) = v$, and $t(e_i) = s(e_{i+1})$, for all i such that $1 \leq i \leq n-1$. When $n = 0$, we must have $u = v$, and the path (u, ϵ, u) is called the *null path from u to u*. The number n is the *length* of the path. We also call u the *source* (or *origin*) of the path, and v the *target* (or *endpoint*) of the path. When there is a nonnull path π from u to v, we say that *u and v are connected*.

Remark: In a path $\pi = (u, e_1 \ldots e_n, v)$, the expression $e_1 \ldots e_n$ is a **sequence**, and thus, the e_i are **not** necessarily distinct.

For example, the following are paths:

$$\pi_1 = (v_1, e_1e_5e_7, v_6),$$

$$\pi_2 = (v_2, e_2e_3e_4e_2e_3e_4e_2e_3e_4, v_2),$$

and

$$\pi_3 = (v_1, e_1e_2e_3e_4e_2e_3e_4e_5e_6e_6e_8, v_6).$$

Clearly, π_2 and π_3 are of a different nature from π_1. Indeed, they contain cycles. This is formalized as follows.

Definition 5.3. Given a directed graph $G = (V, E, s, t)$, for any node $u \in V$ a *cycle (or loop) through u* is a nonnull path of the form $\pi = (u, e_1 \ldots e_n, u)$ (equivalently, $t(e_n) = s(e_1)$). More generally, a nonnull path $\pi = (u, e_1 \ldots e_n, v)$ *contains a cycle* iff for some i, j, with $1 \leq i \leq j \leq n$, $t(e_j) = s(e_i)$. In this case, letting $w = t(e_j) = s(e_i)$, the path $(w, e_i \ldots e_j, w)$ is a cycle through w. A path π is *acyclic* iff it does not contain any cycle. Note that each null path (u, ϵ, u) is acyclic.

Obviously, a cycle $\pi = (u, e_1 \ldots e_n, u)$ through u is also a cycle through every node $t(e_i)$. Also, a path π may contain several different cycles.

Paths can be concatenated as follows.

Definition 5.4. Given a directed graph $G = (V, E, s, t)$, two paths $\pi_1 = (u, e_1 \ldots e_m, v)$ and $\pi_2 = (u', e'_1 \ldots e'_n, v')$ can be *concatenated* provided that $v = u'$, in which case their *concatenation* is the path

$$\pi_1\pi_2 = (u, e_1 \ldots e_m e'_1 \ldots e'_n, v').$$

It is immediately verified that the concatenation of paths is associative, and that the concatenation of the path
$\pi = (u, e_1 \ldots e_m, v)$ with the null path (u, ϵ, u) or with the null path (v, ϵ, v) is the path π itself.

The following fact, although almost trivial, is used all the time, and is worth stating in detail.

Proposition 5.1. *Given a directed graph $G = (V, E, s, t)$, if the set of nodes V contains $m \geq 1$ nodes, then every path π of length at least m contains some cycle.*

A consequence of Proposition 5.1 is that in a finite graph with m nodes, given any two nodes $u, v \in V$, in order to find out whether there is a path from u to v, it is enough to consider paths of length $\leq m - 1$.

Indeed, if there is path between u and v, then there is some path π of minimal length (not necessarily unique, but this doesn't matter).

If this minimal path has length at least m, then by the Proposition, it contains a cycle.

However, by deleting this cycle from the path π, we get an even shorter path from u to v, contradicting the minimality of π.

We now turn to labeled graphs.

5.2 Labeled Graphs and Automata

In fact, we only need edge-labeled graphs.

Definition 5.5. A *labeled directed graph* is a tuple $G = (V, E, L, s, t, \lambda)$, where V is a set of *vertices, or nodes*, E is a set of *edges, or arcs*, L is a set of *labels*, $s, t\colon E \to V$ are two functions, s being called the *source* function, and t the *target* function, and $\lambda\colon E \to L$ is the *labeling function*. Given an edge $e \in E$, we also call $s(e)$ the *origin* (or *source*) of e, $t(e)$ the *endpoint* (or *target*) of e, and $\lambda(e)$ the *label* of e.

Note that the function λ need not be injective or surjective. Thus, distinct edges may have the same label.

Example: Let G be the directed graph defined such that

$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$,

$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$, $L = \{a, b\}$,

and

$$
\begin{array}{llll}
s(e_1) = v_1, & s(e_2) = v_2, & s(e_3) = v_3, & s(e_4) = v_4, \\
s(e_5) = v_2, & s(e_6) = v_5, & s(e_7) = v_5, & s(e_8) = v_5, \\
t(e_1) = v_2, & t(e_2) = v_3, & t(e_3) = v_4, & t(e_4) = v_2, \\
t(e_5) = v_5, & t(e_6) = v_5, & t(e_7) = v_6, & t(e_8) = v_6 \\
\lambda(e_1) = a, & \lambda(e_2) = b, & \lambda(e_3) = a, & \lambda(e_4) = a, \\
\lambda(e_5) = b, & \lambda(e_6) = a, & \lambda(e_7) = a, & \lambda(e_8) = b.
\end{array}
$$

Such a labeled graph can be represented by the following diagram:

In drawing labeled graphs, we will usually omit edge names (the e_i), and sometimes even the node names (the v_j).

Paths, cycles, and concatenation of paths are defined just as before (that is, we ignore the labels). However, we can now define the *spelling* of a path.

Definition 5.6. Given a labeled directed graph $G = (V, E, L, s, t, \lambda)$ for any two nodes $u, v \in V$, for any path $\pi = (u, e_1 \ldots e_n, v)$, the *spelling of the path* π is the string of labels

$$\lambda(e_1) \cdots \lambda(e_n).$$

When $n = 0$, the spelling of the null path (u, ϵ, u) is the null string ϵ.

For example, the spelling of the path

$$\pi_3 = (v_1, e_1e_2e_3e_4e_2e_3e_4e_5e_6e_6e_8, v_6)$$

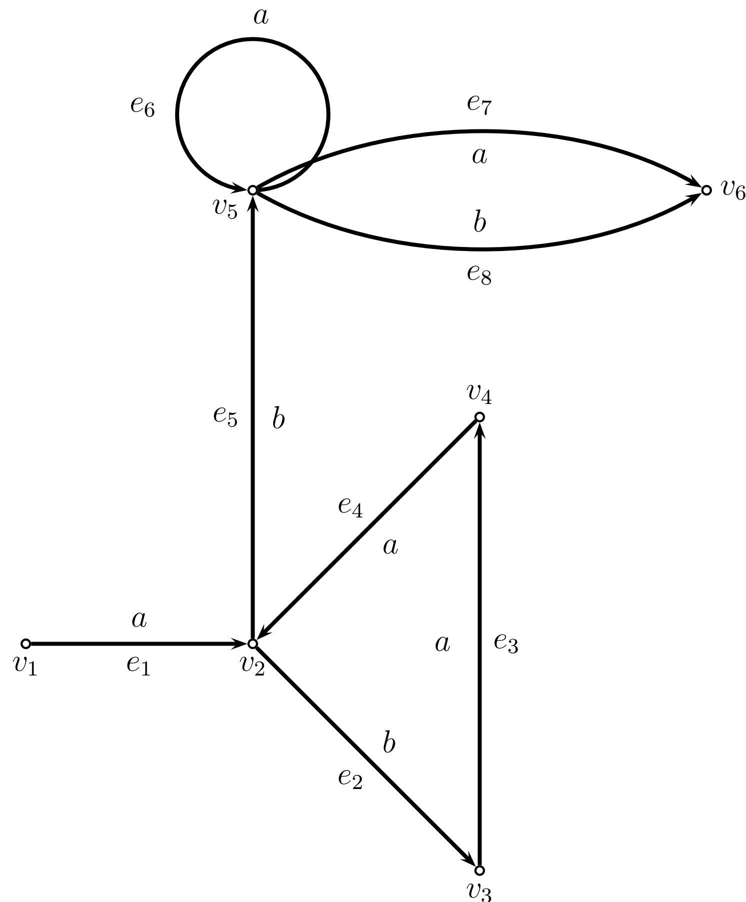


Figure 5.2: A labeled directed graph

is

$$abaabaabaab.$$

Every DFA and every NFA can be viewed as a labeled graph, in such a way that the set of spellings of paths from the start state to some final state is the language accepted by the automaton in question.

Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, where $\delta\colon Q \times \Sigma \to Q$, we associate the labeled directed graph $G_D = (V, E, L, s, t, \lambda)$ defined as follows:

$$V = Q, \quad E = \{(p, a, q) \mid q = \delta(p, a),\ p, q \in Q,\ a \in \Sigma\},$$

$L = \Sigma$, $s((p, a, q)) = p$, $t((p, a, q)) = q$, and $\lambda((p, a, q)) = a$.

Such labeled graphs have a special structure that can easily be characterized.

It is easily shown that a string $w \in \Sigma^*$ is in the language $L(D) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$ iff w is the spelling of some path in G_D from q_0 to some final state.

Similarly, given an NFA $N = (Q, \Sigma, \delta, q_0, F)$, where $\delta \colon Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$, we associate the labeled directed graph $G_N = (V, E, L, s, t, \lambda)$ defined as follows: $V = Q$

$$E = \{(p, a, q) \mid q \in \delta(p, a),\ p, q \in Q,\ a \in \Sigma \cup \{\epsilon\}\},$$

$$L = \Sigma \cup \{\epsilon\},\ \ s((p, a, q)) = p,\ \ t((p, a, q)) = q,$$

$$\lambda((p, a, q)) = a.$$

Remark: When N has no ϵ-transitions, we can let $L = \Sigma$.

Such labeled graphs have also a special structure that can easily be characterized.

Again, a string $w \in \Sigma^*$ is in the language $L(N) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \cap F \neq \emptyset\}$ iff w is the spelling of some path in G_N from q_0 to some final state.

5.3 The Closure Definition of the Regular Languages

Let $\Sigma = \{a_1, \ldots, a_m\}$ be some alphabet. We would like to define a family of languages, $R(\Sigma)$, by singling out some very basic (atomic) languages, namely the languages $\{a_1\}, \ldots, \{a_m\}$, the empty language, and the trivial language, $\{\epsilon\}$, and then forming more complicated languages by repeatedly forming union, concatenation and Kleene $*$ of previously constructed languages. By doing so, we hope to get a family of languages ($R(\Sigma)$) that is closed under union, concatenation, and Kleene $*$. This means that for any two languages, $L_1, L_2 \in R(\Sigma)$, we also have $L_1 \cup L_2 \in R(\Sigma)$ and $L_1 L_2 \in R(\Sigma)$, and for any language $L \in R(\Sigma)$, we have $L^* \in R(\Sigma)$. Furthermore, we would like $R(\Sigma)$ to be the smallest family with these properties. How do we achieve this rigorously?

First, let us look more closely at what we mean by a family of languages. Recall that a language (over Σ) is *any* subset, L, of Σ^*. Thus, the set of all languages is 2^{Σ^*}, the power set of Σ^*. If Σ is nonempty, this is an uncountable set. Next, we define a *family*, $\mathcal{L}$, of languages to be any subset of 2^{Σ^*}. This time, the set of families of languages is $2^{2^{\Sigma^*}}$. This is a huge set. We can use the inclusion relation on $2^{2^{\Sigma^*}}$ to define a partial order on families of languages. So, $\mathcal{L}_1 \subseteq \mathcal{L}_2$ iff for every language, L, if $L \in \mathcal{L}_1$ then $L \in \mathcal{L}_2$.

We can now state more precisely what we are trying to do. Consider the following properties for a family of languages, $\mathcal{L}$:

(1) We have $\{a_1\}, \ldots, \{a_m\}, \emptyset, \{\epsilon\} \in \mathcal{L}$, i.e., $\mathcal{L}$ contains the "atomic" languages.

(2a) For all $L_1, L_2 \in \mathcal{L}$, we also have $L_1 \cup L_2 \in \mathcal{L}$.

(2b) For all $L_1, L_2 \in \mathcal{L}$, we also have $L_1 L_2 \in \mathcal{L}$.

(2c) For all $L \in \mathcal{L}$, we also have $L^* \in \mathcal{L}$.

In other words, $\mathcal{L}$ is closed under union, concatenation and Kleene $*$.

Now, what we want is the smallest (w.r.t. inclusion) family of languages that satisfies properties (1) and (2)(a)(b)(c). We can construct such a family using an *inductive definition*. This inductive definition constructs a sequence of families of languages, $(R(\Sigma)_n)_{n\geq 0}$, called the *stages of the inductive definition*, as follows:

$$\begin{aligned} R(\Sigma)_0 &= \{\{a_1\}, \ldots, \{a_m\}, \emptyset, \{\epsilon\}\}, \\ R(\Sigma)_{n+1} &= R(\Sigma)_n \cup \{L_1 \cup L_2,\ L_1L_2,\ L^* \ \mid L_1, L_2, L \in R(\Sigma)_n\}. \end{aligned}$$

Then, we define $R(\Sigma)$ by

$$R(\Sigma) = \bigcup_{n\geq 0} R(\Sigma)_n.$$

Thus, a language L belongs to $R(\Sigma)$ iff it belongs L_n, for some $n \geq 0$.

For example, if $\Sigma = \{a, b\}$, we have

$$\begin{aligned} R(\Sigma)_1 = \{&\{a\}, \{b\}, \emptyset, \{\epsilon\}, \\ &\{a, b\}, \{a, \epsilon\}, \{b, \epsilon\}, \\ &\{ab\}, \{ba\}, \{aa\}, \{bb\}, \{a\}^*, \{b\}^*\}. \end{aligned}$$

Some of the languages that will appear in $R(\Sigma)_2$ are:

$$\{a, bb\},\ \{ab, ba\},\ \{abb\},\ \{aabb\},\ \{a\}\{a\}^*,\ \{aa\}\{b\}^*,\ \{bb\}^*.$$

Observe that

$$R(\Sigma)_0 \subseteq R(\Sigma)_1 \subseteq R(\Sigma)_2 \subseteq \cdots R(\Sigma)_n \subseteq R(\Sigma)_{n+1} \subseteq \cdots \subseteq R(\Sigma),$$

so that if $L \in R(\Sigma)_n$, then $L \in R(\Sigma)_p$, for all $p \geq n$. Also, there is some smallest n for which $L \in R(\Sigma)_n$ (the *birthdate* of L!). In fact, all these inclusions are strict. Note that each $R(\Sigma)_n$ only contains a finite number of languages (but some of the languages in $R(\Sigma)_n$ are infinite, because of Kleene $*$).

Then we define the *Regular languages, Version 2*, as the family $R(\Sigma)$.

Of course, it is far from obvious that $R(\Sigma)$ coincides with the family of languages accepted by DFA's (or NFA's), what we call the regular languages, version 1. However, this is the case, and this can be demonstrated by giving two algorithms. Actually, it will be slightly more convenient to define a notation system, the *regular expressions*, to denote the languages in $R(\Sigma)$. Then, we will give an algorithm that converts a regular expression, R, into an NFA, N_R, so that $L_R = L(N_R)$, where L_R is the language (in $R(\Sigma)$) denoted by R. We will also give an algorithm that converts an NFA, N, into a regular expression, R_N, so that $L(R_N) = L(N)$.

But before doing all this, we should make sure that $R(\Sigma)$ is indeed the family that we are seeking. This is the content of

Proposition 5.2. *The family, $R(\Sigma)$, is the smallest family of languages which contains the atomic languages $\{a_1\}, \ldots, \{a_m\}$, $\emptyset$, $\{\epsilon\}$, and is closed under union, concatenation, and Kleene $*$.*

Proof. There are two things to prove.

(i) We need to prove that $R(\Sigma)$ has properties (1) and (2)(a)(b)(c).

(ii) We need to prove that $R(\Sigma)$ is the smallest family having properties (1) and (2)(a)(b)(c).

(i) Since

$$R(\Sigma)_0 = \{\{a_1\}, \ldots, \{a_m\}, \emptyset, \{\epsilon\}\},$$

it is obvious that (1) holds. Next, assume that $L_1, L_2 \in R(\Sigma)$. This means that there are some integers $n_1, n_2 \geq 0$, so that $L_1 \in R(\Sigma)_{n_1}$ and $L_2 \in R(\Sigma)_{n_2}$. Now, it is possible that $n_1 \neq n_2$, but if we let $n = \max\{n_1, n_2\}$, as we observed that $R(\Sigma)_p \subseteq R(\Sigma)_q$ whenever $p \leq q$, we are guaranteed that both $L_1, L_2 \in R(\Sigma)_n$. However, by the definition of $R(\Sigma)_{n+1}$ (that's why we defined it this way!), we have $L_1 \cup L_2 \in R(\Sigma)_{n+1} \subseteq R(\Sigma)$. The same argument proves that $L_1L_2 \in R(\Sigma)_{n+1} \subseteq R(\Sigma)$. Also, if $L \in R(\Sigma)_n$, we immediately have $L^* \in R(\Sigma)_{n+1} \subseteq R(\Sigma)$. Therefore, $R(\Sigma)$ has properties (1) and (2)(a)(b)(c).

(ii) Let $\mathcal{L}$ be any family of languages having properties (1) and (2)(a)(b)(c). We need to prove that $R(\Sigma) \subseteq \mathcal{L}$. If we can prove that $R(\Sigma)_n \subseteq \mathcal{L}$, for all $n \geq 0$, we are done (since then, $R(\Sigma) = \bigcup_{n\geq 0} R(\Sigma)_n \subseteq \mathcal{L}$). We prove by induction on n that $R(\Sigma)_n \subseteq \mathcal{L}$, for all $n \geq 0$.

The base case $n = 0$ is trivial, since $\mathcal{L}$ has (1), which says that $R(\Sigma)_0 \subseteq \mathcal{L}$. Assume inductively that $R(\Sigma)_n \subseteq \mathcal{L}$. We need to prove that $R(\Sigma)_{n+1} \subseteq \mathcal{L}$. Pick any $L \in R(\Sigma)_{n+1}$. Recall that

$$R(\Sigma)_{n+1} = R(\Sigma)_n \cup \{L_1 \cup L_2,\ L_1L_2,\ L^* \ \mid L_1, L_2, L \in R(\Sigma)_n\}.$$

If $L \in R(\Sigma)_n$, then $L \in \mathcal{L}$, since $R(\Sigma)_n \subseteq \mathcal{L}$, by the induction hypothesis. Otherwise, there are three cases:

(a) $L = L_1 \cup L_2$, where $L_1, L_2 \in R(\Sigma)_n$. By the induction hypothesis, $R(\Sigma)_n \subseteq \mathcal{L}$, so, we get $L_1, L_2 \in \mathcal{L}$; since $\mathcal{L}$ has 2(a), we have $L_1 \cup L_2 \in \mathcal{L}$.

(b) $L = L_1L_2$, where $L_1, L_2 \in R(\Sigma)_n$. By the induction hypothesis, $R(\Sigma)_n \subseteq \mathcal{L}$, so, we get $L_1, L_2 \in \mathcal{L}$; since $\mathcal{L}$ has 2(b), we have $L_1L_2 \in \mathcal{L}$.

(c) $L = L_1^*$, where $L_1 \in R(\Sigma)_n$. By the induction hypothesis, $R(\Sigma)_n \subseteq \mathcal{L}$, so, we get $L_1 \in \mathcal{L}$; since $\mathcal{L}$ has 2(c), we have $L_1^* \in \mathcal{L}$.

Thus, in all cases, we showed that $L \in \mathcal{L}$, and so, $R(\Sigma)_{n+1} \subseteq \mathcal{L}$, which proves the induction step. ◻

Note: a given language L may be built up in different ways. For example,

$$\{a, b\}^* = (\{a\}^*\{b\}^*)^*.$$

Students should study carefully the above proof. Although simple, it is the prototype of many proofs appearing in the theory of computation.

5.4 Regular Expressions

The definition of the family of languages $R(\Sigma)$ given in the previous section in terms of an inductive definition is good to prove properties of these languages but is it not very convenient to manipulate them in a practical way. To do so, it is better to introduce a symbolic notation system, the *regular expressions*. Regular expressions are certain strings formed according to rules that mimic the inductive rules for constructing the families $R(\Sigma)_n$. The set of regular expressions $\mathcal{R}(\Sigma)$ over an alphabet Σ is a language defined on an alphabet Δ defined as follows.

Given an alphabet $\Sigma = \{a_1, \ldots, a_m\}$, consider the new alphabet

$$\Delta = \Sigma \cup \{+, \cdot, *, (,), \emptyset, \epsilon\}.$$

We define the family $(\mathcal{R}(\Sigma)_n)$ of languages over Δ as follows:

$$\begin{aligned} \mathcal{R}(\Sigma)_0 &= \{a_1, \ldots, a_m, \emptyset, \epsilon\}, \\ \mathcal{R}(\Sigma)_{n+1} &= \mathcal{R}(\Sigma)_n \cup \{(R_1 + R_2),\ (R_1 \cdot R_2),\ R^* \mid \\ &\qquad\qquad R_1, R_2, R \in \mathcal{R}(\Sigma)_n\}. \end{aligned}$$

Then, we define $\mathcal{R}(\Sigma)$ as

$$\mathcal{R}(\Sigma) = \bigcup_{n \geq 0} \mathcal{R}(\Sigma)_n.$$

Note that every language $\mathcal{R}(\Sigma)_n$ is finite.

For example, if $\Sigma = \{a, b\}$, we have

$$\begin{aligned} \mathcal{R}(\Sigma)_1 = \{&a, b, \emptyset, \epsilon, \\ &(a + b), (b + a), (a + a), (b + b), (a + \epsilon), (\epsilon + a), \\ &(b + \epsilon), (\epsilon + b), (a + \emptyset), (\emptyset + a), (b + \emptyset), (\emptyset + b), \\ &(\epsilon + \epsilon), (\epsilon + \emptyset), (\emptyset + \epsilon), (\emptyset + \emptyset), \\ &(a \cdot b), (b \cdot a), (a \cdot a), (b \cdot b), (a \cdot \epsilon), (\epsilon \cdot a), \\ &(b \cdot \epsilon), (\epsilon \cdot b), (\epsilon \cdot \epsilon), (a \cdot \emptyset), (\emptyset \cdot a), \\ &(b \cdot \emptyset), (\emptyset \cdot b), (\epsilon \cdot \emptyset), (\emptyset \cdot \epsilon), (\emptyset \cdot \emptyset), \\ &a^*, b^*, \epsilon^*, \emptyset^*\}. \end{aligned}$$

Some of the regular expressions appearing in $\mathcal{R}(\Sigma)_2$ are:

$$(a + (b \cdot b)),\ ((a \cdot b) + (b \cdot a)),\ ((a \cdot b) \cdot b),$$
$$((a \cdot a) \cdot (b \cdot b)),\ (a \cdot a^*),\ ((a \cdot a) \cdot b^*),\ (b \cdot b)^*.$$

Definition 5.7. The set $\mathcal{R}(\Sigma)$ is the set of *regular expressions* (over Σ).

Proposition 5.3. *The language $\mathcal{R}(\Sigma)$ is the smallest language which contains the symbols $a_1, \ldots, a_m$, $\emptyset$, ϵ, from Δ, and such that $(R_1 + R_2)$, $(R_1 \cdot R_2)$, and R^*, also belong to $\mathcal{R}(\Sigma)$, when $R_1, R_2, R \in \mathcal{R}(\Sigma)$.*

For simplicity of notation, we write

$$(R_1R_2)$$

instead of

$$(R_1 \cdot R_2).$$

Examples: $R = (a + b)^*$, $S = (a^*b^*)^*$.

$$T = ((a + b)^*a)(\underbrace{(a + b) \cdots (a + b)}_{n}).$$

5.5 Regular Expressions and Regular Languages

Every regular expression $R \in \mathcal{R}(\Sigma)$ can be viewed as the *name*, or *denotation*, of some language $L \in R(\Sigma)$. Similarly, every language $L \in R(\Sigma)$ is the *interpretation* (or *meaning*) of some regular expression $R \in \mathcal{R}(\Sigma)$.

Think of a regular expression R as a *program*, and of $\mathcal{L}(R)$ as the result of the *execution* or *evaluation*, of R by $\mathcal{L}$.

This can be made rigorous by defining a function

$$\mathcal{L} \colon \mathcal{R}(\Sigma) \to R(\Sigma).$$

This function is defined recursively as follows:

$$\begin{aligned}
\mathcal{L}[a_i] &= \{a_i\}, \\
\mathcal{L}[\emptyset] &= \emptyset, \\
\mathcal{L}[\epsilon] &= \{\epsilon\}, \\
\mathcal{L}[(R_1 + R_2)] &= \mathcal{L}[R_1] \cup \mathcal{L}[R_2], \\
\mathcal{L}[(R_1R_2)] &= \mathcal{L}[R_1]\mathcal{L}[R_2], \\
\mathcal{L}[R^*] &= \mathcal{L}[R]^*.
\end{aligned}$$

Proposition 5.4. *For every regular expression $R \in \mathcal{R}(\Sigma)$, the language $\mathcal{L}[R]$ is regular (version 2), i.e. $\mathcal{L}[R] \in R(\Sigma)$. Conversely, for every regular (version 2) language $L \in R(\Sigma)$, there is some regular expression $R \in \mathcal{R}(\Sigma)$ such that $L = \mathcal{L}[R]$.*

Proof. To prove that $\mathcal{L}[R] \in R(\Sigma)$ for all $R \in \mathcal{R}(\Sigma)$, we prove by induction on $n \geq 0$ that if $R \in \mathcal{R}(\Sigma)_n$, then $\mathcal{L}[R] \in R(\Sigma)_n$. To prove that $\mathcal{L}$ is surjective, we prove by induction on $n \geq 0$ that if $L \in R(\Sigma)_n$, then there is some $R \in \mathcal{R}(\Sigma)_n$ such that $L = \mathcal{L}[R]$. ☐

Note: the function $\mathcal{L}$ is **not** injective.

Example: If $R = (a+b)^*$, $S = (a^*b^*)^*$, then

$$\mathcal{L}[R] = \mathcal{L}[S] = \{a, b\}^*.$$

For simplicity, we often denote $\mathcal{L}[R]$ as L_R. As examples, we have

$$\begin{aligned}
\mathcal{L}[(((ab)b) + a)] &= \{a, abb\} \\
\mathcal{L}[((((a^*b)a^*)b)a^*)] &= \{w \in \{a,b\}^* \mid w \text{ has} \\
&\qquad \text{two } b\text{'s}\} \\
\mathcal{L}[(((((a^*b)a^*)b)a^*)^*a^*)] &= \{w \in \{a,b\}^* \mid w \text{ has an} \\
&\qquad \text{even \# of } b\text{'s}\} \\
\mathcal{L}[(((((((a^*b)a^*)b)a^*)^*a^*)b)a^*)] &= \{w \in \{a,b\}^* \mid w \text{ has an} \\
&\qquad \text{odd \# of } b\text{'s}\}
\end{aligned}$$

Remark. If

$$R = ((a+b)^*a)(\underbrace{(a+b)\cdots(a+b)}_{n}),$$

it can be shown that any minimal DFA accepting L_R has 2^{n+1} states. Yet, both $((a+b)^*a)$ and $(\underbrace{(a+b)\cdots(a+b)}_{n})$ denote languages that can be accepted by "small" DFA's (of size 2 and $n+2$).

Definition 5.8. Two regular expressions $R, S \in \mathcal{R}(\Sigma)$ are *equivalent*, denoted as $R \cong S$, iff $\mathcal{L}[R] = \mathcal{L}[S]$.

It is immediate that $\cong$ is an equivalence relation. The relation $\cong$ satisfies some (nice) identities. For example:

$$\begin{aligned}
(((aa) + b) + c) &\cong ((aa) + (b + c)) \\
((aa)(b(cc))) &\cong (((aa)b)(cc)) \\
(a^*a^*) &\cong a^*,
\end{aligned}$$

and more generally

$$\begin{aligned} ((R_1 + R_2) + R_3) &\cong (R_1 + (R_2 + R_3)), \\ ((R_1 R_2) R_3) &\cong (R_1 (R_2 R_3)), \\ (R_1 + R_2) &\cong (R_2 + R_1), \\ (R^* R^*) &\cong R^*, \\ R^{**} &\cong R^*. \end{aligned}$$

There is an algorithm to test the equivalence of regular expressions, but its complexity is exponential. Such an algorithm uses the conversion of a regular expression to an NFA, and the subset construction for converting an NFA to a DFA. Then the problem of deciding whether two regular expressions R and S are equivalent is reduced to testing whether two DFA D_1 and D_2 accept the same languages (the *equivalence problem for DFA's*). This last problem is equivalent to testing whether

$$L(D_1) - L(D_2) = \emptyset \quad \text{and} \quad L(D_2) - L(D_1) = \emptyset.$$

But $L(D_1) - L(D_2)$ (and similarly $L(D_2) - L(D_1)$) is accepted by a DFA obtained by the cross-product construction for the relative complement (with final states $F_1 \times \overline{F_2}$ and $\overline{F_1} \times F_2$). Thus, in the end, the equivalence problem for regular expressions reduces to the problem of testing whether a DFA $D = (Q, \Sigma, \delta, q_0, F)$ accepts the empty language, which is equivalent to $Q_r \cap F = \emptyset$. This last problem is a reachability problem in a directed graph which is easily solved in polynomial time.

It is an *open problem* to prove that the problem of testing the equivalence of regular expressions cannot be decided in polynomial time.

In the next two sections we show the equivalence of NFA's and regular expressions, by providing an algorithm to construct an NFA from a regular expression, and an algorithm for constructing a regular expression from an NFA. This will show that the regular languages Version 1 coincide with the regular languages Version 2.

5.6 Regular Expressions and NFA's

Proposition 5.5. *There is an algorithm, which, given any regular expression $R \in \mathcal{R}(\Sigma)$, constructs an NFA N_R accepting L_R, i.e., such that $L_R = L(N_R)$.*

In order to ensure the correctness of the construction as well as to simplify the description of the algorithm it is convenient to assume that our NFA's satisfy the following conditions:

1. Each NFA has a *single* final state, t, distinct from the start state, s.
2. There are *no incoming transitions* into the the start state, s, and *no outgoing transitions* from the final state, t.

3. Every state has at most two incoming and two outgoing transitions.

Here is the algorithm.

For the base case, either

(a) $R = a_i$, in which case, N_R is the following NFA:

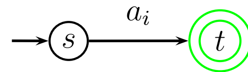


Figure 5.3: NFA for a_i

(b) $R = \epsilon$, in which case, N_R is the following NFA:

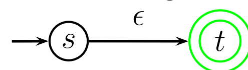


Figure 5.4: NFA for ϵ

(c) $R = \emptyset$, in which case, N_R is the following NFA:

Figure 5.5: NFA for $\emptyset$

The recursive clauses are as follows:

(i) If our expression is $(R+S)$, the algorithm is applied recursively to R and S, generating NFA's N_R and N_S, and then these two NFA's are combined in parallel as shown in Figure 5.6:

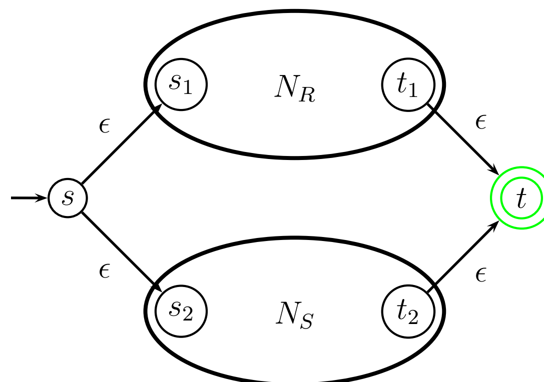


Figure 5.6: NFA for $(R + S)$

(ii) If our expression is $(R \cdot S)$, the algorithm is applied recursively to R and S, generating NFA's N_R and N_S, and then these NFA's are combined sequentially as shown in Figure 5.7 by merging the "old" final state, t_1, of N_R, with the "old" start state, s_2, of N_S:

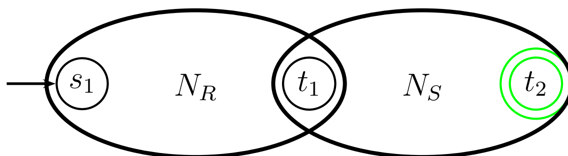


Figure 5.7: NFA for $(R \cdot S)$

Note that since there are no incoming transitions into s_2 in N_S, once we enter N_S, there is no way of reentering N_R, and so the construction is correct (it yields the concatenation $L_R L_S$).

(iii) If our expression is R^*, the algorithm is applied recursively to R, generating the NFA N_R. Then we construct the NFA shown in Figure 5.8 by adding an ϵ-transition from the "old" final state, t_1, of N_R to the "old" start state, s_1, of N_R and, as ϵ is not necessarily accepted by N_R, we add an ϵ-transition from s to t:

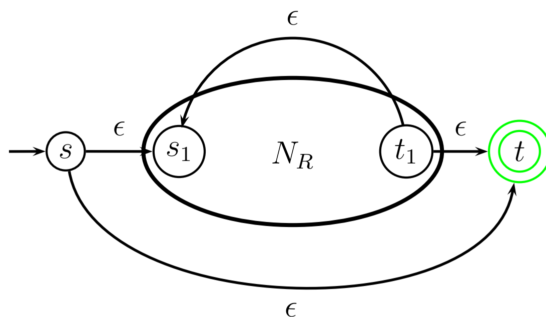


Figure 5.8: NFA for R^*

Since there are no outgoing transitions from t_1 in N_R, we can only loop back to s_1 from t_1 using the new ϵ-transition from t_1 to s_1 and so the NFA of Figure 5.8 does accept N_R^*.

The algorithm that we just described is sometimes called the "sombrero construction."

As a corollary of this construction, we get

Reg. languages version 2 $\subseteq$ Reg. languages, version 1.

The reader should check that if one constructs the NFA corresponding to the regular expression $(a + b)^*abb$ and then applies the subset construction, one get the following DFA:

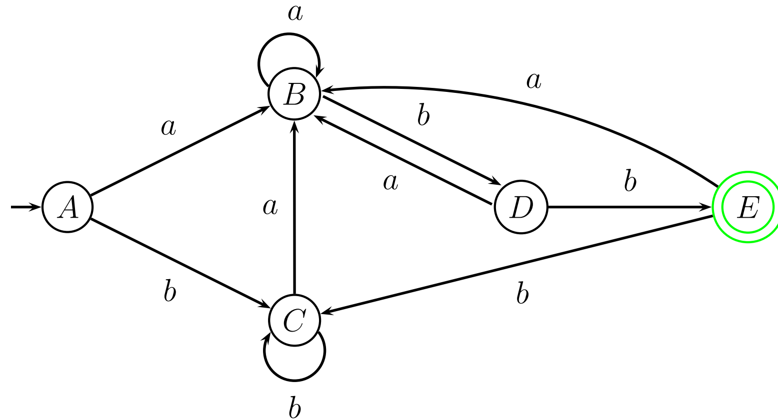


Figure 5.9: A non-minimal DFA for $\{a,b\}^*\{abb\}$

We now consider the construction of a regular expression from an NFA.

Proposition 5.6. *There is an algorithm, which, given any NFA N, constructs a regular expression $R \in \mathcal{R}(\Sigma)$, denoting $L(N)$, i.e., such that $L_R = L(N)$.*

As a corollary,

Reg. languages version 1 $\subseteq$ Reg. languages, version 2.

This is the *node elimination algorithm*.

The general idea is to allow more general labels on the edges of an NFA, namely, regular expressions. Then, such generalized NFA's are simplified by eliminating nodes one at a time, and readjusting labels.

Preprocessing, phase 1:

If necessary, we need to add a new start state with an ϵ-transition to the old start state, if there are incoming edges into the old start state.

If necessary, we need to add a new (unique) final state with ϵ-transitions from each of the old final states to the new final state, if there is more than one final state or some outgoing edge from any of the old final states.

At the end of this phase, the start state, say s, is a source (no incoming edges), and the final state, say t, is a sink (no outgoing edges).

Preprocessing, phase 2:

We need to "flatten" parallel edges. For any pair of states (p, q) ($p = q$ is possible), if there are k edges from p to q labeled $u_1, \ldots, u_k$, then create a single edge labeled with the regular expression

$$u_1 + \cdots + u_k.$$

For any pair of states (p, q) ($p = q$ is possible) such that there is **no** edge from p to q, we put an edge labeled $\emptyset$.

At the end of this phase, the resulting "*generalized NFA*" is such that for any pair of states (p, q) (where $p = q$ is possible), there is a unique edge labeled with some regular expression denoted as $R_{p,q}$. When $R_{p,q} = \emptyset$, this really means that there is no edge from p to q in the original NFA N.

By interpreting each $R_{p,q}$ as a function call (really, a macro) to the NFA $N_{p,q}$ accepting $\mathcal{L}[R_{p,q}]$ (constructed using the previous algorithm), we can verify that the original language $L(N)$ is accepted by this new generalized NFA.

Node elimination only applies if the generalized NFA has at least one node distinct from s and t.

Pick any node r distinct from s and t. For every pair (p, q) where $p \neq r$ and $q \neq r$, replace the label of the edge from p to q as indicated below:

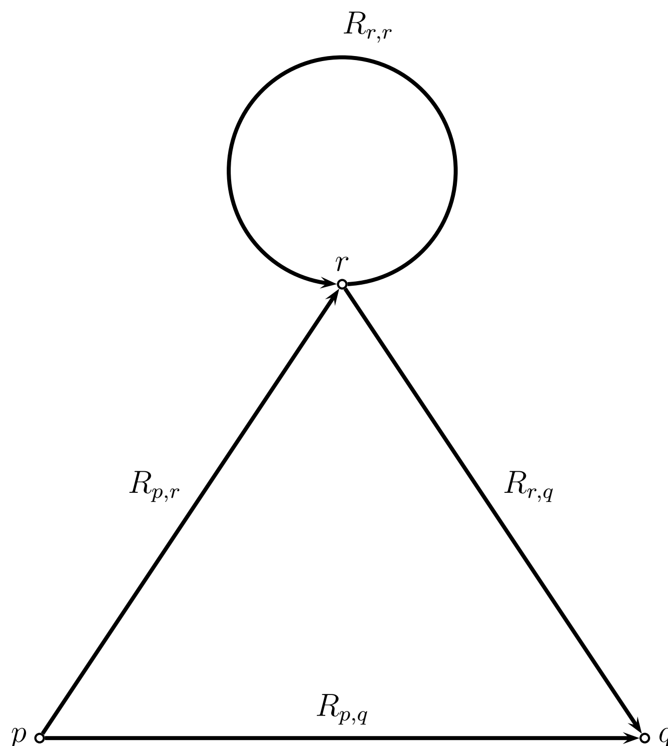


Figure 5.10: Before Eliminating node r

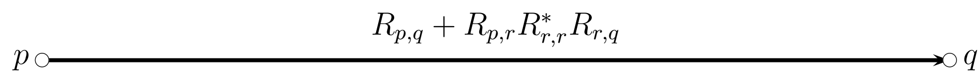


Figure 5.11: After Eliminating node r

At the end of this step, delete the node r and all edges adjacent to r.

Note that $p = q$ is possible, in which case the triangle is "flat". It is also possible that $p = s$ or $q = t$. Also, this step is performed for all **pairs** (p, q), which means that both (p, q) and (q, p) are considered (when $p \neq q$)).

Note that this step only has an effect if there are edges from p to r and from r to q in the original NFA N. Otherwise, r can simply be deleted, as well as the edges adjacent to r.

Other simplifications can be made. For example, when $R_{r,r} = \emptyset$, we can simplify $R_{p,r}R_{r,r}^* R_{r,q}$ to $R_{p,r}R_{r,q}$. When $R_{p,q} = \emptyset$, we have $R_{p,r}R_{r,r}^* R_{r,q}$.

The order in which the nodes are eliminated is irrelevant, although it affects the size of the final expression.

The algorithm stops when the only remaining nodes are s and t. Then, the label R of the edge from s to t is a regular expression denoting $L(N)$.

For example, let

$$L = \{w \in \Sigma^* \mid w \text{ contains an odd number of } a\text{'s} \\ \text{or an odd number of } b\text{'s}\}.$$

An NFA for L after the preprocessing phase is:

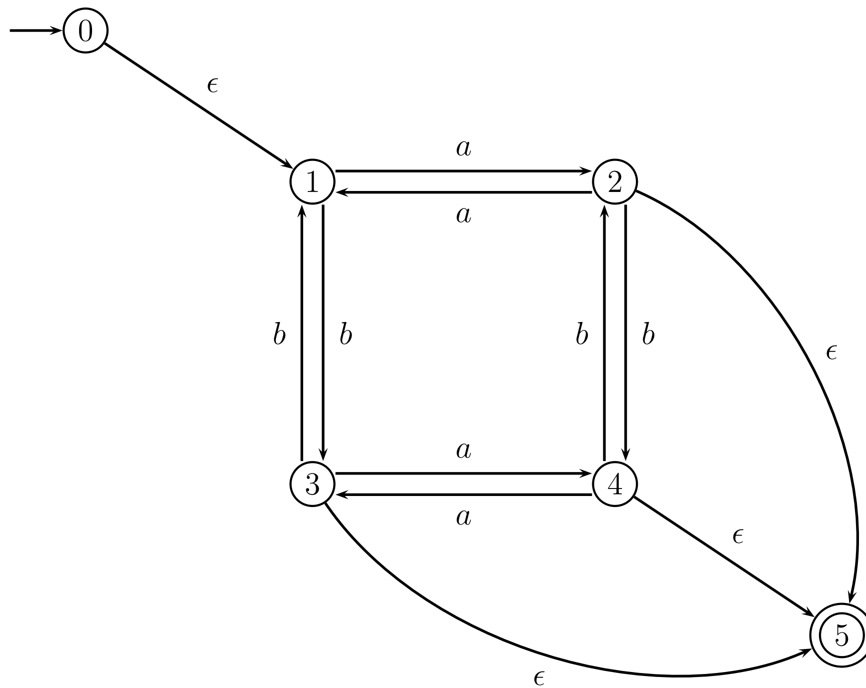


Figure 5.12: NFA for L (after preprocessing phase)

After eliminating node 2:

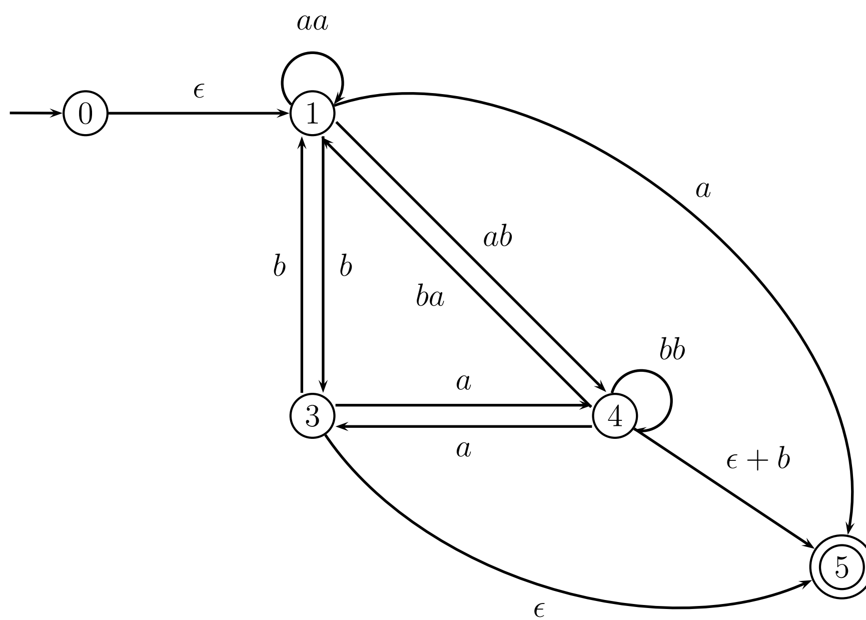


Figure 5.13: NFA for L (after eliminating node 2)

After eliminating node 3:

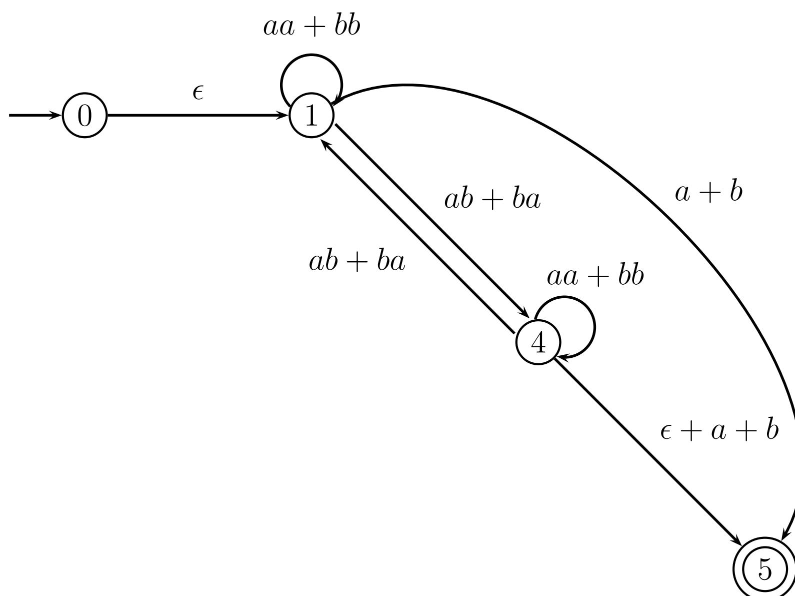


Figure 5.14: NFA for L (after eliminating node 3)

After eliminating node 4:

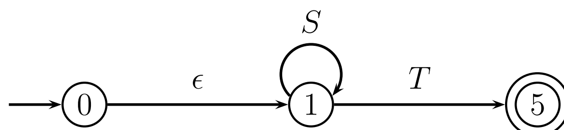


Figure 5.15: NFA for L (after eliminating node 4)

where

$$T = a + b + (ab + ba)(aa + bb)^*(\epsilon + a + b)$$

and

$$S = aa + bb + (ab + ba)(aa + bb)^*(ab + ba).$$

Finally, after eliminating node 1, we get:

$$R = (aa + bb + (ab + ba)(aa + bb)^*(ab + ba))^*(a + b + (ab + ba)(aa + bb)^*(\epsilon + a + b)).$$

5.7 Applications of Regular Expressions: Lexical analysis, Finding patterns in text

Regular expressions have several practical applications. The first important application is to *lexical analysis*.

A *lexical analyzer* is the first component of a *compiler*.

The purpose of a lexical analyzer is to scan the source program and break it into atomic components, known as *tokens*, i.e., substrings of consecutive characters that belong together logically.

Examples of tokens are: identifiers, keywords, numbers (in fixed point notation or floating point notation, etc.), arithmetic operators ($+, \cdot, -, \hat{}$), comparison operators ($<, >, =, <>$), assignment operator (:=), etc.

Tokens can be described by regular expressions. For this purpose, it is useful to enrich the syntax of regular expressions, as in UNIX.

For example, the 26 upper case letters of the (roman) alphabet, $A, \ldots, Z$, can be specified by the expression

[A-Z]

Similarly, the ten digits, $0, 1, \ldots, 9$, can be specified by the expression

[0-9]

The regular expression

$$R_1 + R_2 + \cdots + R_k$$

is denoted

$$[R_1 R_2 \cdots R_k]$$

So, the expression

[A-Za-z0-9]

denotes any letter (upper case or lower case) or digit. This is called an *alphanumeric*.

If we define an identifier as a string beginning with a letter (upper case or lower case) followed by any number of alphanumerics (including none), then we can use the following expression to specify identifiers:

[A-Za-z][A-Za-z0-9]∗

There are systems, such as `lex` or `flex` that accept as input a list of regular expressions describing the tokens of a programming language and construct a lexical analyzer for these tokens.

Such systems are called *lexical analyzer generators*. Basically, they build a DFA from the set of regular expressions using the algorithms that have been described earlier.

Usually, it is possible associate with every expression some action to be taken when the corresponding token is recognized

Another application of regular expressions is finding patterns in text.

Using a regular expression, we can specify a "vaguely defined" class of patterns.

Take the example of a street address. Most street addresses end with "Street", or "Avenue", or "Road" or "St.", or "Ave.", or "Rd.".

We can design a regular expression that captures the shape of most street addresses and then convert it to a DFA that can be used to search for street addresses in text.

For more on this, see Hopcroft-Motwani and Ullman.

5.8 Summary of Closure Properties of the Regular Languages

The family of regular languages is closed under many operations. In particular, it is closed under the following operations listed below. Some of the closure properties are left as a homework problem.

(1) Union, intersection, relative complement.

(2) Concatenation, Kleene $*$, Kleene $+$.

(3) Homomorphisms and inverse homomorphisms.

(4) gsm and inverse gsm mappings, a-transductions and inverse a-transductions.

Another useful operation is substitution.

Given any two alphabets Σ, Δ, a *substitution* is a function, $\tau\colon \Sigma \to 2^{\Delta^*}$, assigning some language, $\tau(a) \subseteq \Delta^*$, to every symbol $a \in \Sigma$.

A substitution $\tau\colon \Sigma \to 2^{\Delta^*}$ is extended to a map $\tau\colon 2^{\Sigma^*} \to 2^{\Delta^*}$ by first extending τ to strings using the following definition

$$\begin{aligned} \tau(\epsilon) &= \{\epsilon\}, \\ \tau(ua) &= \tau(u)\tau(a), \end{aligned}$$

where $u \in \Sigma^*$ and $a \in \Sigma$, and then to languages by letting

$$\tau(L) = \bigcup_{w \in L} \tau(w),$$

for every language $L \subseteq \Sigma^*$.

Observe that a homomorphism is a special kind of substitution.

A substitution is a *regular* substitution iff $\tau(a)$ is a regular language for every $a \in \Sigma$. The proof of the next proposition is left as a homework problem.

Proposition 5.7. *If L is a regular language and τ is a regular substitution, then $\tau(L)$ is also regular. Thus, the family of regular languages is closed under regular substitutions.*

5.9 Right-Invariant Equivalence Relations on Σ^*

The purpose of this section is to give one more characterization of the regular languages in terms of certain kinds of equivalence relations on strings. Pushing this characterization a bit further, we will be able to show how minimal DFA's can be found.

Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. The DFA D may be redundant, for example, if there are states that are not accessible from the start state. The set Q_r of *accessible or reachable states* is the subset of Q defined as

$$Q_r = \{p \in Q \mid \exists w \in \Sigma^*,\ \delta^*(q_0, w) = p\}.$$

If $Q \neq Q_r$, we can "clean up" D by deleting the states in $Q - Q_r$ and restricting the transition function δ to Q_r. This way, we get an equivalent DFA D_r such that $L(D) = L(D_r)$, where all the states of D_r are reachable. From now on, we assume that we are dealing with DFA's such that $D = D_r$, called *trim, or reachable*.

Recall that an *equivalence relation* $\simeq$ on a set A is a relation which is *reflexive*, *symmetric*, and *transitive*. Given any $a \in A$, the set

$$\{b \in A \mid a \simeq b\}$$

is called the *equivalence class of* a, and it is denoted as $[a]_\simeq$, or even as $[a]$. Recall that for any two elements $a, b \in A$, $[a] \cap [b] = \emptyset$ iff $a \not\simeq b$, and $[a] = [b]$ iff $a \simeq b$. The set of equivalence classes associated with the equivalence relation $\simeq$ is a *partition* Π of A (also denoted as $A/\simeq$). This means that it is a family of nonempty pairwise disjoint sets whose union is equal to A itself. The equivalence classes are also called the *blocks* of the partition Π. The number of blocks in the partition Π is called the *index* of $\simeq$ (and Π).

Given any two equivalence relations $\simeq_1$ and $\simeq_2$ with associated partitions Π_1 and Π_2,

$$\simeq_1 \subseteq \simeq_2$$

iff every block of the partition Π_1 is contained in some block of the partition Π_2. Then, every block of the partition Π_2 is the union of blocks of the partition Π_1, and we say that $\simeq_1$ is a *refinement* of $\simeq_2$ (and similarly, Π_1 is a refinement of Π_2). Note that Π_2 has at most as many blocks as Π_1 does.

We now define an equivalence relation on strings induced by a DFA. This equivalence is a kind of "observational" equivalence, in the sense that we decide that two strings u, v are equivalent iff, when feeding first u and then v to the DFA, u and v drive the DFA to the same state. From the point of view of the observer, u and v have the same effect (reaching the same state).

Definition 5.9. Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, we define the relation $\simeq_D$ on Σ^* as follows: for any two strings $u, v \in \Sigma^*$,

$$u \simeq_D v \quad \text{iff} \quad \delta^*(q_0, u) = \delta^*(q_0, v).$$

Example 5.1. We can figure out what the equivalence classes of $\simeq_D$ are for the following DFA:

	a	b
0	1	0
1	2	1
2	0	2

with 0 both start state and (unique) final state. For example

$$\begin{aligned} abbabbb &\simeq_D aa \\ ababab &\simeq_D \epsilon \\ bba &\simeq_D a. \end{aligned}$$

There are three equivalences classes:

$$[\epsilon]_{\simeq}, \quad [a]_{\simeq}, \quad [aa]_{\simeq}.$$

Observe that $L(D) = [\epsilon]_{\simeq}$. Also, the equivalence classes are in one–to–one correspondence with the states of D.

The relation $\simeq_D$ turns out to have some interesting properties. In particular, it is *right-invariant*, which means that for all $u, v, w \in \Sigma^*$, if $u \simeq v$, then $uw \simeq vw$.

Proposition 5.8. *Given any (trim) DFA $D = (Q, \Sigma, \delta, q_0, F)$, the relation $\simeq_D$ is an equivalence relation which is right-invariant and has finite index. Furthermore, if Q has n states, then the index of $\simeq_D$ is n, and every equivalence class of $\simeq_D$ is a regular language. Finally, $L(D)$ is the union of some of the equivalence classes of $\simeq_D$.*

Proof. The fact that $\simeq_D$ is an equivalence relation is a trivial verification. To prove that $\simeq_D$ is right-invariant, we first prove by induction on the length of v that for all $u, v \in \Sigma^*$, for all $p \in Q$,

$$\delta^*(p, uv) = \delta^*(\delta^*(p, u), v).$$

Then, if $u \simeq_D v$, which means that $\delta^*(q_0, u) = \delta^*(q_0, v)$, we have

$$\delta^*(q_0, uw) = \delta^*(\delta^*(q_0, u),\, w) = \delta^*(\delta^*(q_0, v),\, w) = \delta^*(q_0, vw),$$

which means that $uw \simeq_D vw$. Thus, $\simeq_D$ is right-invariant. We still have to prove that $\simeq_D$ has index n. Define the function $f\colon \Sigma^* \to Q$ such that

$$f(u) = \delta^*(q_0, u).$$

Note that if $u \simeq_D v$, which means that $\delta^*(q_0, u) = \delta^*(q_0, v)$, then $f(u) = f(v)$. Thus, the function $f\colon \Sigma^* \to Q$ has the *same value* on all the strings in some equivalence class $[u]$, so it induces a function $\widehat{f}\colon \Pi \to Q$ defined such that

$$\widehat{f}([u]) = f(u)$$

for every equivalence class $[u] \in \Pi$, where $\Pi = \Sigma^*/\simeq$ is the partition associated with $\simeq_D$. This function is well defined since $f(v)$ has the same value for all elements v in the equivalence class $[u]$.

However, the function $\widehat{f}\colon \Pi \to Q$ is injective (one-to-one), since $\widehat{f}([u]) = \widehat{f}([v])$ is equivalent to $f(u) = f(v)$ (since by definition of $\widehat{f}$ we have $\widehat{f}([u]) = f(u)$ and $\widehat{f}([v]) = f(v)$), which by definition of f means that $\delta^*(q_0, u) = \delta^*(q_0, v)$, which means precisely that $u \simeq_D v$, that is, $[u] = [v]$.

Since Q has n states, Π has at most n blocks. Moreover, since every state is accessible, for every $q \in Q$, there is some $w \in \Sigma^*$ so that $\delta^*(q_0, w) = q$, which shows that $\widehat{f}([w]) = f(w) = q$. Consequently, $\widehat{f}$ is also surjective. But then, being injective and surjective, $\widehat{f}$ is bijective and Π has exactly n blocks.

Every equivalence class of Π is a set of strings of the form

$$\{w \in \Sigma^* \mid \delta^*(q_0, w) = p\},$$

for some $p \in Q$, which is accepted by the DFA

$$D_p = (Q, \Sigma, \delta, q_0, \{p\})$$

obtained from D by changing F to $\{p\}$. Thus, every equivalence class is a regular language. Finally, since

$$\begin{aligned} L(D) &= \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\} \\ &= \bigcup_{f \in F} \{w \in \Sigma^* \mid \delta^*(q_0, w) = f\} \\ &= \bigcup_{f \in F} L(D_f), \end{aligned}$$

we see that $L(D)$ is the union of the equivalence classes corresponding to the final states in F. $\square$

One should not be too optimistic and hope that every equivalence relation on strings is right-invariant.

Example 5.2. For example, if $\Sigma = \{a\}$, the equivalence relation $\simeq$ given by the partition

$$\{\epsilon, a, a^4, a^9, a^{16}, \ldots, a^{n^2}, \ldots \mid n \geq 0\} \cup \{a^2, a^3, a^5, a^6, a^7, a^8, \ldots, a^m, \ldots \mid m \text{ is not a square}\}$$

we have $a \simeq a^4$, yet by concatenating on the right with a^5, since $aa^5 = a^6$ and $a^4a^5 = a^9$ we get

$$a^6 \not\simeq a^9,$$

that is, a^6 and a^9 are *not* equivalent. It turns out that the problem is that neither equivalence class is a regular language.

It is worth noting that a right-invariant equivalence relation is not necessarily *left-invariant*, which means that if $u \simeq v$ then $wu \simeq wv$.

Example 5.3. For example, if $\simeq$ is given by the four equivalence classes

$$C_1 = \{bb\}^*, \;\; C_2 = \{bb\}^*a, \;\; C_3 = b\{bb\}^*, \;\; C_4 = \{bb\}^*a\{a,b\}^+ \cup b\{bb\}^*a\{a,b\}^*,$$

then we can check that $\simeq$ is right-invariant by figuring out the inclusions $C_ia \subseteq C_j$ and $C_ib \subseteq C_j$, which are recorded in the following table:

	a	b
C_1	C_2	C_3
C_2	C_4	C_4
C_3	C_4	C_1
C_4	C_4	C_4

However, both $ab, ba \in C_4$, yet $bab \in C_4$ and $bba \in C_2$, so $\simeq$ is not left-invariant.

The remarkable fact due to Myhill and Nerode is that Proposition 5.8 has a converse. Indeed, given a right-invariant equivalence relation of finite index it is possible to reconstruct a DFA, and by a suitable choice of final state, every equivalence class is accepted by such a DFA. Let us show how this DFA is constructed using a simple example.

Example 5.4. Consider the equivalence relation $\simeq$ on $\{a,b\}^*$ given by the three equivalence classes

$$C_1 = \{\epsilon\}, \;\; C_2 = a\{a,b\}^*, \;\; C_3 = b\{a,b\}^*.$$

We leave it as an easy exercise to check that $\simeq$ is right-invariant. For example, if $u \simeq v$ and $u, v \in C_2$, then $u = ax$ and $v = ay$ for some $x, y \in \{a,b\}^*$, so for any $w \in \{a,b\}^*$ we have $uw = axw$ and $vw = ayw$, which means that we also have $uw, vw \in C_2$, thus $uw \simeq vw$.

For any subset $C \subseteq \{a, b\}^*$ and any string $w \in \{a, b\}^*$ define Cw as the set of strings

$$Cw = \{uw \mid u \in C\}.$$

There are two reasons why a DFA can be recovered from the right-invariant equivalence relation $\simeq$:

(1) For every equivalence class C_i and every string w, there is a unique equivalence class C_j such that

$$C_i w \subseteq C_j.$$

Actually, it is enough to check the above property for strings w of length 1 (*i.e.* symbols in the alphabet) because the property for arbitrary strings follows by induction.

(2) For every $w \in \Sigma^*$ and every class C_i,

$$C_1 w \subseteq C_i \quad \text{iff} \quad w \in C_i,$$

where C_1 is the equivalence class of the empty string.

We can make a table recording these inclusions.

Example 5.5. Continuing Example 5.4, we get:

	a	b
C_1	C_2	C_3
C_2	C_2	C_2
C_3	C_3	C_3

For example, from $C_1 = \{\epsilon\}$ we have $C_1 a = \{a\} \subseteq C_2$ and $C_1 b = \{b\} \subseteq C_3$, for $C_2 = a\{a, b\}^*$, we have $C_2 a = a\{a, b\}^* a \subseteq C_2$ and $C_2 a = a\{a, b\}^* b \subseteq C_2$, and for $C_3 = b\{a, b\}^*$, we have $C_3 a = b\{a, b\}^* a \subseteq C_3$ and $C_3 b = b\{a, b\}^* b \subseteq C_3$.

The key point is that the above table is the transition table of a DFA with start state $C_1 = [\epsilon]$. Furthermore, if C_i $(i = 1, 2, 3)$ is chosen as a single final state, the corresponding DFA D_i accepts C_i. This is the converse of Myhill-Nerode!

Observe that the inclusions $C_i w \subseteq C_j$ may be strict inclusions. For example, $C_1 a = \{a\}$ is a proper subset of $C_2 = a\{a, b\}^*$

Let us do another example.

Example 5.6. Consider the equivalence relation $\simeq$ given by the four equivalence classes

$$C_1 = \{\epsilon\}, \;\; C_2 = \{a\}, \;\; C_3 = \{b\}^+, \;\; C_4 = a\{a, b\}^+ \cup \{b\}^+ a\{a, b\}^*.$$

We leave it as an easy exercise to check that $\simeq$ is right-invariant.

We obtain the following table of inclusions $C_i a \subseteq C_j$ and $C_i b \subseteq C_j$:

	a	b
C_1	C_2	C_3
C_2	C_4	C_4
C_3	C_4	C_3
C_4	C_4	C_4

For example, from $C_3 = \{b\}^+$ we get $C_3 a = \{b\}^+ a \subseteq C_4$, and $C_3 b = \{b\}^+ b \subseteq C_3$.

The above table is the transition function of a DFA with four states and start state C_1. If C_i $(i = 1, 2, 3, 4)$ is chosen as a single final state, the corresponding DFA D_i accepts C_i.

Here is the general result.

Proposition 5.9. *Given any equivalence relation $\simeq$ on Σ^*, if $\simeq$ is right-invariant and has finite index n, then every equivalence class (block) in the partition Π associated with $\simeq$ is a regular language.*

Proof. Let $C_1, \ldots, C_n$ be the blocks of Π, and assume that $C_1 = [\epsilon]$ is the equivalence class of the empty string.

First, we claim that for every block C_i and every $w \in \Sigma^*$, there is a unique block C_j such that $C_i w \subseteq C_j$, where $C_i w = \{uw \mid u \in C_i\}$.

For every $u \in C_i$, the string uw belongs to one and only one of the blocks of Π, say C_j. For any other string $v \in C_i$, since (by definition) $u \simeq v$, by right invariance, we get $uw \simeq vw$, but since $uw \in C_j$ and C_j is an equivalence class, we also have $vw \in C_j$. This proves the first claim.

We also claim that for every $w \in \Sigma^*$, for every block C_i,

$$C_1 w \subseteq C_i \quad \text{iff} \quad w \in C_i.$$

If $C_1 w \subseteq C_i$, since $C_1 = [\epsilon]$, we have $\epsilon w = w \in C_i$. Conversely, if $w \in C_i$, for any $v \in C_1 = [\epsilon]$, since $\epsilon \simeq v$, by right invariance we have $w \simeq vw$, and thus $vw \in C_i$, which shows that $C_1 w \subseteq C_i$.

For every class C_k, let

$$D_k = (\{1, \ldots, n\}, \Sigma, \delta, 1, \{k\}),$$

where $\delta(i, a) = j$ iff $C_i a \subseteq C_j$. We will prove the following equivalence:

$$\delta^*(i, w) = j \quad \text{iff} \quad C_i w \subseteq C_j.$$

For this, we prove the following two implications by induction on $|w|$:

(a) If $\delta^*(i, w) = j$, then $C_i w \subseteq C_j$, and

(b) If $C_i w \subseteq C_j$, then $\delta^*(i, w) = j$.

The base case ($w = \epsilon$) is trivial for both (a) and (b). We leave the proof of the induction step for (a) as an exercise and give the proof of the induction step for (b) because it is more subtle. Let $w = ua$, with $a \in \Sigma$ and $u \in \Sigma^*$. If $C_i ua \subseteq C_j$, then by the first claim, we know that there is a unique block, C_k, such that $C_i u \subseteq C_k$. Furthermore, there is a unique block, C_h, such that $C_k a \subseteq C_h$, but $C_i u \subseteq C_k$ implies $C_i ua \subseteq C_k a$ so we get $C_i ua \subseteq C_h$. However, by the uniqueness of the block, C_j, such that $C_i ua \subseteq C_j$, we must have $C_h = C_j$. By the induction hypothesis, as $C_i u \subseteq C_k$, we have

$$\delta^*(i, u) = k$$

and, by definition of δ, as $C_k a \subseteq C_j$ $(= C_h)$, we have $\delta(k, a) = j$, so we deduce that

$$\delta^*(i, ua) = \delta(\delta^*(i, u), a) = \delta(k, a) = j,$$

as desired. Then, using the equivalence just proved and the second claim, we have

$$\begin{aligned} L(D_k) &= \{w \in \Sigma^* \mid \delta^*(1, w) \in \{k\}\} \\ &= \{w \in \Sigma^* \mid \delta^*(1, w) = k\} \\ &= \{w \in \Sigma^* \mid C_1 w \subseteq C_k\} \\ &= \{w \in \Sigma^* \mid w \in C_k\} = C_k, \end{aligned}$$

proving that every block, C_k, is a regular language. $\square$

In general it is false that $C_i a = C_j$ for some block C_j, and we can only claim that $C_i a \subseteq C_j$.

We can combine Proposition 5.8 and Proposition 5.9 to get the following characterization of a regular language due to Myhill and Nerode:

Theorem 5.10. *(Myhill-Nerode) A language L (over an alphabet Σ) is a regular language iff it is the union of some of the equivalence classes of an equivalence relation $\simeq$ on Σ^*, which is right-invariant and has finite index.*

Theorem 5.10 can also be used to prove that certain languages are not regular. A general scheme (not the only one) goes as follows: If L is not regular, then it must be infinite. Now, we argue by contradiction. If L was regular, then by Myhill-Nerode, there would be some equivalence relation $\simeq$, which is right-invariant and of finite index, and such that L is the union of some of the classes of $\simeq$. Because Σ^* is infinite and $\simeq$ has only finitely many equivalence classes, there are strings $x, y \in \Sigma^*$ with $x \neq y$ so that

$$x \simeq y.$$

If we can find a third string, $z \in \Sigma^*$, such that

$$xz \in L \quad \text{and} \quad yz \notin L,$$

then we reach a contradiction. Indeed, by right invariance, from $x \simeq y$, we get $xz \simeq yz$. But, L is the union of equivalence classes of $\simeq$, so if $xz \in L$, then we should also have $yz \in L$, contradicting $yz \notin L$. Therefore, L is not regular.

Then the scenario is this: to prove that L is not regular, first we check that L is infinite. If so, we try finding three strings x, y, z, where and x and $y \neq x$ are prefixes of strings in L such that

$$x \simeq y,$$

where $\simeq$ is a right-invariant relation of finite index such that L is the union of equivalence of L (which must exist by Myhill–Nerode since we are assuming by contradiction that L is regular), and where z is chosen so that

$$xz \in L \quad \text{and} \quad yz \notin L.$$

Example 5.7. For example, we prove that $L = \{a^n b^n \mid n \geq 1\}$ is not regular.

Assuming for the sake of contradiction that L is regular, there is some equivalence relation $\simeq$ which is right-invariant and of finite index and such that L is the union of some of the classes of $\simeq$. Since the sequence

$$a, aa, aaa, \ldots, a^i, \ldots$$

is infinite and $\simeq$ has a finite number of classes, two of these strings must belong to the same class, which means that $a^i \simeq a^j$ for some $i \neq j$. But since $\simeq$ is right invariant, by concatenating with b^i on the right, we see that $a^i b^i \simeq a^j b^i$ for some $i \neq j$. However $a^i b^i \in L$, and since L is the union of classes of $\simeq$, we also have $a^j b^i \in L$ for $i \neq j$, which is absurd, given the definition of L. Thus, in fact, L is not regular.

Here is another illustration of the use of the Myhill-Nerode Theorem to prove that a language is not regular.

Example 5.8. We claim that the language,

$$L' = \{a^{n!} \mid n \geq 1\},$$

is not regular, where $n!$ (n factorial) is given by $0! = 1$ and $(n+1)! = (n+1)n!$.

Assume L' is regular. Then, there is some equivalence relation $\simeq$ which is right-invariant and of finite index and such that L' is the union of some of the classes of $\simeq$. Since the sequence

$$a, a^2, \ldots, a^n, \ldots$$

is infinite, two of these strings must belong to the same class, which means that $a^p \simeq a^q$ for some p, q with $1 \leq p < q$. As $q! \geq q$ for all $q \geq 0$ and $q > p$, we can concatenate on the right with $a^{q!-p}$ and we get

$$a^p a^{q!-p} \simeq a^q a^{q!-p},$$

that is,

$$a^{q!} \simeq a^{q!+q-p}.$$

Since $p < q$ we have $q! < q! + q - p$. If we can show that

$$q! + q - p < (q+1)!$$

we will obtain a contradiction because then $a^{q!+q-p} \notin L'$, yet $a^{q!+q-p} \simeq a^{q!}$ and $a^{q!} \in L'$, contradicting Myhill-Nerode. Now, as $1 \leq p < q$, we have $q - p \leq q - 1$, so if we can prove that

$$q! + q - p \leq q! + q - 1 < (q+1)!$$

we will be done. However, $q! + q - 1 < (q+1)!$ is equivalent to

$$q - 1 < (q+1)! - q!,$$

and since $(q+1)! - q! = (q+1)q! - q! = qq!$, we simply need to prove that

$$q - 1 < q \leq qq!,$$

which holds for $q \geq 1$.

There is another version of the Myhill-Nerode Theorem involving congruences which is also quite useful. An equivalence relation, $\simeq$, on Σ^* is *left and right-invariant* iff for all $x, y, u, v \in \Sigma^*$,

$$\text{if} \quad x \simeq y \quad \text{then} \quad uxv \simeq uyv.$$

An equivalence relation, $\simeq$, on Σ^* is a *congruence* iff for all $u_1, u_2, v_1, v_2 \in \Sigma^*$,

$$\text{if} \quad u_1 \simeq v_1 \quad \text{and} \quad u_2 \simeq v_2 \quad \text{then} \quad u_1u_2 \simeq v_1v_2.$$

It is easy to prove that an equivalence relation is a congruence iff it is left and right-invariant.

For example, assume that $\simeq$ is a left and right-invariant equivalence relation, and assume that

$$u_1 \simeq v_1 \quad \text{and} \quad u_2 \simeq v_2.$$

By right-invariance applied to $u_1 \simeq v_1$, we get

$$u_1u_2 \simeq v_1u_2$$

and by left-invariance applied to $u_2 \simeq v_2$ we get

$$v_1u_2 \simeq v_1v_2.$$

By transitivity, we conclude that

$$u_1u_2 \simeq v_1v_2.$$

which shows that $\simeq$ is a congruence.

Proving that a congruence is left and right-invariant is even easier.

There is a version of Proposition 5.8 that applies to congruences and for this we define the relation $\sim_D$ as follows: For any (trim) DFA, $D = (Q, \Sigma, \delta, q_0, F)$, for all $x, y \in \Sigma^*$,

$$x \sim_D y \quad \text{iff} \quad (\forall q \in Q)(\delta^*(q, x) = \delta^*(q, y)).$$

Proposition 5.11. *Given any (trim) DFA, $D = (Q, \Sigma, \delta, q_0, F)$, the relation $\sim_D$ is an equivalence relation which is left and right-invariant and has finite index. Furthermore, if Q has n states, then the index of $\sim_D$ is at most n^n and every equivalence class of $\sim_D$ is a regular language. Finally, $L(D)$ is the union of some of the equivalence classes of $\sim_D$.*

Proof. We leave most of the proof of Proposition 5.11 as an exercise. The last two parts of the proposition are proved using the following facts:

(1) Since $\sim_D$ is left and right-invariant and has finite index, in particular, $\sim_D$ is right-invariant and has finite index, so by Proposition 5.9 every equivalence class of $\sim_D$ is regular.

(2) Observe that

$$\sim_D \subseteq \simeq_D,$$

since the condition $\delta^*(q, x) = \delta^*(q, y)$ holds for every $q \in Q$, so in particular for $q = q_0$. But then, every equivalence class of $\simeq_D$ is the union of equivalence classes of $\sim_D$ and since, by Proposition 5.8, L is the union of equivalence classes of $\simeq_D$, we conclude that L is also the union of equivalence classes of $\sim_D$.

This completes the proof. □

Using Proposition 5.11 and Proposition 5.9, we obtain another version of the Myhill-Nerode Theorem.

Theorem 5.12. *(Myhill-Nerode, Congruence Version) A language L (over an alphabet Σ) is a regular language iff it is the union of some of the equivalence classes of an equivalence relation $\simeq$ on Σ^*, which is a congruence and has finite index.*

We now consider an equivalence relation associated with a language L.

5.10 Finding minimal DFA's

Given any language L (not necessarily regular), we can define an equivalence relation ρ_L on Σ^* which is right-invariant, but not necessarily of finite index. The equivalence relation ρ_L is such that L is the union of equivalence classes of ρ_L. Furthermore, when L is regular, the relation ρ_L has finite index. In fact, this index is the size of a smallest DFA accepting L. As a consequence, if L is regular, a simple modification of the proof of Proposition 5.9 applied to $\simeq\, =\, \rho_L$ yields a minimal DFA D_{ρ_L} accepting L.

Then, given any trim DFA D accepting L, the equivalence relation ρ_L can be translated to an equivalence relation $\equiv$ on states, in such a way that for all $u, v \in \Sigma^*$,

$$u\rho_L v \quad \text{iff} \quad \varphi(u) \equiv \varphi(v),$$

where $\varphi\colon \Sigma^* \to Q$ is the function (run the DFA D on u from q_0) given by

$$\varphi(u) = \delta^*(q_0, u).$$

One can then construct a quotient DFA $D/\equiv$ whose states are obtained by merging all states in a given equivalence class of states into a single state, and the resulting DFA $D/\equiv$ is a mininal DFA. Even though $D/\equiv$ appears to depend on D, it is in fact unique, and isomorphic to the abstract DFA D_{ρ_L} induced by ρ_L.

The last step in obtaining the minimal DFA $D/\equiv$ is to give a constructive method to compute the state equivalence relation $\equiv$. This can be done by constructing a sequence of approximations $\equiv_i$, where each $\equiv_{i+1}$ refines $\equiv_i$. It turns out that if D has n states, then there is some index $i_0 \leq n-2$ such that

$$\equiv_j = \equiv_{i_0} \quad \text{for all } j \geq i_0 + 1,$$

and that

$$\equiv\, =\, \equiv_{i_0}.$$

Furthermore, $\equiv_{i+1}$ can be computed inductively from $\equiv_i$. In summary, we obtain a iterative algorithm for computing $\equiv$ that terminates in at most $n-2$ steps.

Definition 5.10. Given any language L (over Σ), we define the *right-invariant equivalence* ρ_L *associated with* L as the relation on Σ^* defined as follows: for any two strings $u, v \in \Sigma^*$,

$$u\rho_L v \quad \text{iff} \quad \forall w \in \Sigma^*(uw \in L \quad \text{iff} \quad vw \in L).$$

It is clear that the relation ρ_L is an equivalence relation, and it is right-invariant. To show right-invariance, argue as follows: if $u\rho_L v$, then for any $w \in \Sigma^*$, since $u\rho_L v$ means that

$$uz \in L \quad \text{iff} \quad vz \in L$$

for all $z \in \Sigma^*$, in particular the above equivalence holds for all z of the form $z = wy$ for any arbitary $y \in \Sigma^*$, so we have

$$uwy \in L \quad \text{iff} \quad vwy \in L$$

for all $y \in \Sigma^*$, which means that $uw\rho_L vw$.

It is also clear that L is the union of the equivalence classes of strings in L. This is because if $u \in L$ and $u\rho_L v$, by letting $w = \epsilon$ in the definition of ρ_L, we get

$$u \in L \quad \text{iff} \quad v \in L,$$

and since $u \in L$, we also have $v \in L$. This implies that if $u \in L$ then $[u]_{\rho_L} \subseteq L$ and so,

$$L = \bigcup_{u \in L} [u]_{\rho_L}.$$

Example 5.9. For example, consider the regular language

$$L = \{a\} \cup \{b^m \mid m \geq 1\}.$$

We leave it as an exercise to show that the equivalence relation ρ_L consists of the four equivalence classes

$$C_1 = \{\epsilon\},\ C_2 = \{a\},\ C_3 = \{b\}^+,\ C_4 = a\{a, b\}^+ \cup \{b\}^+ a\{a, b\}^*$$

encountered earlier in Example 5.6. Observe that

$$L = C_2 \cup C_3.$$

When L is regular, we have the following remarkable result:

Proposition 5.13. *Given any regular language L, for any (trim) DFA $D = (Q, \Sigma, \delta, q_0, F)$ such that $L = L(D)$, ρ_L is a right-invariant equivalence relation, and we have $\simeq_D \subseteq \rho_L$. Furthermore, if ρ_L has m classes and Q has n states, then $m \leq n$.*

Proof. By definition, $u \simeq_D v$ iff $\delta^*(q_0, u) = \delta^*(q_0, v)$. Since $w \in L(D)$ iff $\delta^*(q_0, w) \in F$, the fact that $u\rho_L v$ can be expressed as

$$\begin{aligned}
&\forall w \in \Sigma^*(uw \in L \quad \text{iff} \quad vw \in L) \\
&\text{iff} \\
&\forall w \in \Sigma^*(\delta^*(q_0, uw) \in F \quad \text{iff} \quad \delta^*(q_0, vw) \in F) \\
&\text{iff} \\
&\forall w \in \Sigma^*(\delta^*(\delta^*(q_0, u), w) \in F \quad \text{iff} \quad \delta^*(\delta^*(q_0, v), w) \in F),
\end{aligned}$$

and if $\delta^*(q_0, u) = \delta^*(q_0, v)$, this shows that $u\rho_L v$. Since the number of classes of $\simeq_D$ is n and $\simeq_D \subseteq \rho_L$, the equivalence relation ρ_L has fewer classes than $\simeq_D$, and $m \leq n$. ☐

Proposition 5.13 shows that when L is regular, the index m of ρ_L is finite, and it is a lower bound on the size of all DFA's accepting L. It remains to show that a DFA with m states accepting L exists.

However, going back to the proof of Proposition 5.9 starting with the right-invariant equivalence relation ρ_L of finite index m, if L is the union of the classes $C_{i_1}, \ldots, C_{i_k}$, the DFA

$$D_{\rho_L} = (\{1, \ldots, m\}, \Sigma, \delta, 1, \{i_1, \ldots, i_k\}),$$

where $\delta(i, a) = j$ iff $C_i a \subseteq C_j$, is such that $L = L(D_{\rho_L})$.

In summary, if L is regular, then the index of ρ_L is equal to the number of states of a minimal DFA for L, and D_{ρ_L} is a minimal DFA accepting L.

Example 5.10. For example, if

$$L = \{a\} \cup \{b^m \mid m \geq 1\}.$$

then we saw in Example 5.9 that ρ_L consists of the four equivalence classes

$$C_1 = \{\epsilon\},\ C_2 = \{a\},\ C_3 = \{b\}^+,\ C_4 = a\{a, b\}^+ \cup \{b\}^+ a\{a, b\}^*,$$

and we showed in Example 5.6 that the transition table of D_{ρ_L} is given by

	a	b
C_1	C_2	C_3
C_2	C_4	C_4
C_3	C_4	C_3
C_4	C_4	C_4

By picking the final states to be C_2 and C_3, we obtain the minimal DFA D_{ρ_L} accepting $L = \{a\} \cup \{b^m \mid m \geq 1\}$.

In the next section, we give an algorithm which allows us to find D_{ρ_L}, given any DFA D accepting L. This algorithms finds which states of D are equivalent.

5.11 State Equivalence and Minimal DFA's

The proof of Proposition 5.13 suggests the following definition of an equivalence between states:

Definition 5.11. Given any DFA $D = (Q, \Sigma, \delta, q_0, F)$, the relation $\equiv$ on Q, called *state equivalence*, is defined as follows: for all $p, q \in Q$,

$$p \equiv q \quad \text{iff} \quad \forall w \in \Sigma^* (\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F). \tag{$*$}$$

When $p \equiv q$, we say that *p and q are indistinguishable*.

It is trivial to verify that $\equiv$ is an equivalence relation, and that it satisfies the following property:

$$\text{if } p \equiv q \text{ then } \delta(p,a) \equiv \delta(q,a), \quad \text{for all } a \in \Sigma.$$

To prove the above, since the condition defining $\equiv$ must hold for all strings $w \in \Sigma^*$, in particular it must hold for all strings of the form $w = au$ with $a \in \Sigma$ and $u \in \Sigma^*$, so if $p \equiv q$ then we have

$$\begin{aligned}
& (\forall a \in \Sigma)(\forall u \in \Sigma^*)(\delta^*(p, au) \in F \quad \text{iff} \quad \delta^*(q, au) \in F) \\
\text{iff} \quad & (\forall a \in \Sigma)(\forall u \in \Sigma^*)(\delta^*(\delta^*(p, a), u) \in F \quad \text{iff} \quad \delta^*(\delta^*(q, a), u) \in F) \\
\text{iff} \quad & (\forall a \in \Sigma)(\forall u \in \Sigma^*)(\delta^*(\delta(p, a), u) \in F \quad \text{iff} \quad \delta^*(\delta(q, a), u) \in F) \\
\text{iff} \quad & (\forall a \in \Sigma)(\delta(p, a) \equiv \delta(q, a)).
\end{aligned}$$

$$\delta^*(p, \epsilon) \in F \quad \text{iff} \quad \delta^*(q, \epsilon) \in F,$$

which, since $\delta^*(p, \epsilon) = p$ and $\delta^*(q, \epsilon) = q$, is equivalent to

$$p \in F \quad \text{iff} \quad q \in F.$$

Therefore, if two states p, q are equivalent, then either both $p, q \in F$ or both $p, q \in \overline{F}$. This implies that a final state and a rejecting states are *never* equivalent.

Example 5.11. The reader should check that states A and C in the DFA below are equivalent and that no other distinct states are equivalent.

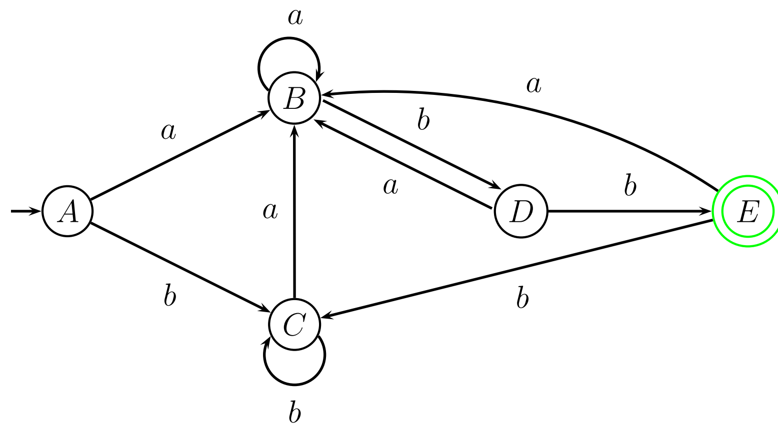


Figure 5.16: A non-minimal DFA for $\{a, b\}^*\{abb\}$

It is illuminating to express state equivalence as the equality of two languages. Given the DFA $D = (Q, \Sigma, \delta, q_0, F)$, let $D_p = (Q, \Sigma, \delta, p, F)$ be the DFA obtained from D by redefining the start state to be p. Then, it is clear that

$$p \equiv q \quad \text{iff} \quad L(D_p) = L(D_q).$$

This simple observation implies that there is an algorithm to test state equivalence. Indeed, we simply have to test whether the DFA's D_p and D_q accept the same language and this can be done using the cross-product construction. Indeed, $L(D_p) = L(D_q)$ iff $L(D_p) - L(D_q) = \emptyset$ and $L(D_q) - L(D_p) = \emptyset$. Now, if $(D_p \times D_q)_{1-2}$ denotes the cross-product DFA with starting state (p, q) and with final states $F \times (Q - F)$ and $(D_p \times D_q)_{2-1}$ denotes the cross-product DFA also with starting state (p, q) and with final states $(Q - F) \times F$, we know that

$$L((D_p \times D_q)_{1-2}) = L(D_p) - L(D_q) \quad \text{and} \quad L((D_p \times D_q)_{2-1}) = L(D_q) - L(D_p),$$

so all we need to do if to test whether $(D_p \times D_q)_{1-2}$ and $(D_p \times D_q)_{2-1}$ accept the empty language. However, we know that this is the case iff the set of states reachable from (p, q) in $(D_p \times D_q)_{1-2}$ contains no state in $F \times (Q - F)$ and the set of states reachable from (p, q) in $(D_p \times D_q)_{2-1}$ contains no state in $(Q - F) \times F$.

Actually, the graphs of $(D_p \times D_q)_{1-2}$ and $(D_p \times D_q)_{2-1}$ are identical, so we only need to check that no state in $(F \times (Q - F)) \cup ((Q - F) \times F)$ is reachable from (p, q) in that graph. This algorithm to test state equivalence is not the most efficient but it is quite reasonable (it runs in polynomial time).

If $L = L(D)$, Theorem 5.14 below shows the relationship between ρ_L and $\equiv$ and, more generally, between the DFA, D_{ρ_L}, and the DFA, $D/\equiv$, obtained as the quotient of the DFA D modulo the equivalence relation $\equiv$ on Q.

The minimal DFA $D/\equiv$ is obtained by merging the states in each block C_i of the partition Π associated with $\equiv$, forming states corresponding to the blocks C_i, and drawing a transition on input a from a block C_i to a block C_j of Π iff there is a transition $q = \delta(p, a)$ from any state $p \in C_i$ to any state $q \in C_j$ on input a.

The start state is the block containing q_0, and the final states are the blocks consisting of final states.

Example 5.12. For example, consider the DFA D_1 accepting $L = \{ab, ba\}^*$ shown in Figure 5.17.

This is not a minimal DFA. In fact,

$$0 \equiv 2 \quad \text{and} \quad 3 \equiv 5.$$

Here is the minimal DFA for L:

The minimal DFA D_2 is obtained by merging the states in the equivalence class $\{0, 2\}$ into a single state, similarly merging the states in the equivalence class $\{3, 5\}$ into a single state, and drawing the transitions between equivalence classes. We obtain the DFA shown in Figure 5.18.

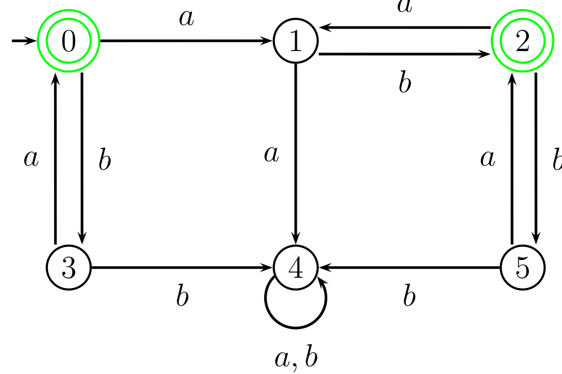


Figure 5.17: A nonminimal DFA D_1 for $L = \{ab, ba\}^*$

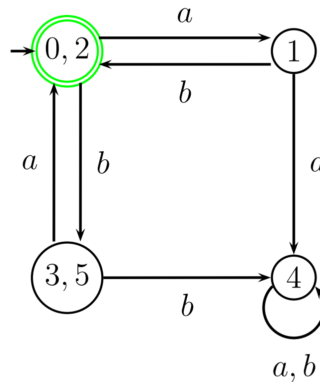


Figure 5.18: A minimal DFA D_2 for $L = \{ab, ba\}^*$

Formally, the quotient DFA $D/\equiv$ is defined such that

$$D/\equiv = (Q/\equiv, \Sigma, \delta/\equiv, [q_0]_\equiv, F/\equiv),$$

where

$$\delta/\equiv([p]_\equiv, a) = [\delta(p, a)]_\equiv.$$

Theorem 5.14. *For any (trim) DFA $D = (Q, \Sigma, \delta, q_0, F)$ accepting the regular language $L = L(D)$, the function $\varphi\colon \Sigma^* \to Q$ defined such that*

$$\varphi(u) = \delta^*(q_0, u)$$

satisfies the property

$$u\rho_L v \quad \text{iff} \quad \varphi(u) \equiv \varphi(v) \quad \text{for all } u, v \in \Sigma^*,$$

and induces a bijection $\widehat{\varphi}\colon \Sigma^*/\rho_L \to Q/\equiv$, defined such that

$$\widehat{\varphi}([u]_{\rho_L}) = [\delta^*(q_0, u)]_\equiv.$$

Furthermore, we have

$$[u]_{\rho_L} a \subseteq [v]_{\rho_L} \quad \textit{iff} \quad \delta(\varphi(u), a) \equiv \varphi(v).$$

Consequently, $\widehat{\varphi}$ induces an isomorphism of DFA's, $\widehat{\varphi}\colon D_{\rho_L} \to D/\equiv$.

Proof. Since $\varphi(u) = \delta^*(q_0, u)$ and $\varphi(v) = \delta^*(q_0, v)$, the fact that $\varphi(u) \equiv \varphi(v)$ can be expressed as

$$\begin{aligned}&\forall w \in \Sigma^*(\delta^*(\delta^*(q_0, u), w) \in F \quad \text{iff} \quad \delta^*(\delta^*(q_0, v), w) \in F)\\ &\text{iff}\\ &\forall w \in \Sigma^*(\delta^*(q_0, uw) \in F \quad \text{iff} \quad \delta^*(q_0, vw) \in F),\end{aligned}$$

which is exactly $u\,\rho_L\,v$. Therefore,

$$u\,\rho_L\,v \quad \text{iff} \quad \varphi(u) \equiv \varphi(v).$$

From the above, we see that the equivalence class $[\varphi(u)]_\equiv$ of $\varphi(u)$ does not depend on the choice of the representative in the equivalence class $[u]_{\rho_L}$ of $u \in \Sigma^*$, since for any $v \in \Sigma^*$, if $u\,\rho_L\,v$ then $\varphi(u) \equiv \varphi(v)$, so $[\varphi(u)]_\equiv = [\varphi(v)]_\equiv$. Therefore, the function $\varphi\colon \Sigma^* \to Q$ maps each equivalence class $[u]_{\rho_L}$ modulo ρ_L to the equivalence class $[\varphi(u)]_\equiv$ modulo $\equiv$, and so the function $\widehat{\varphi}\colon \Sigma^*/\rho_L \to Q/\equiv$ given by

$$\widehat{\varphi}([u]_{\rho_L}) = [\varphi(u)]_\equiv = [\delta^*(q_0, u)]_\equiv$$

is well-defined. Moreover, $\widehat{\varphi}$ is injective, since $\widehat{\varphi}([u]) = \widehat{\varphi}([v])$ iff $\varphi(u) \equiv \varphi(v)$ iff (from above) $u\rho_v v$ iff $[u] = [v]$. Since every state in Q is accessible, for every $q \in Q$, there is some $u \in \Sigma^*$ so that $\varphi(u) = \delta^*(q_0, u) = q$, so $\widehat{\varphi}([u]) = [q]_\equiv$ and $\widehat{\varphi}$ is surjective. Therefore, we have a bijection $\widehat{\varphi}\colon \Sigma^*/\rho_L \to Q/\equiv$.

Since $\varphi(u) = \delta^*(q_0, u)$, we have

$$\delta(\varphi(u), a) = \delta(\delta^*(q_0, u), a) = \delta^*(q_0, ua) = \varphi(ua),$$

and thus, $\delta(\varphi(u), a) \equiv \varphi(v)$ can be expressed as $\varphi(ua) \equiv \varphi(v)$. By the previous part, this is equivalent to $ua\rho_L v$, and we claim that this is equivalent to

$$[u]_{\rho_L} a \subseteq [v]_{\rho_L}.$$

First, if $[u]_{\rho_L} a \subseteq [v]_{\rho_L}$, then $ua \in [v]_{\rho_L}$, that is, $ua\rho_L v$. Conversely, if $ua\rho_L v$, then for every $u' \in [u]_{\rho_L}$, we have $u'\rho_L u$, so by right-invariance we get $u'a\rho_L ua$, and since $ua\rho_L v$, we get $u'a\rho_L v$, that is, $u'a \in [v]_{\rho_L}$. Since $u' \in [u]_{\rho_L}$ is arbitrary, we conclude that $[u]_{\rho_L} a \subseteq [v]_{\rho_L}$. Therefore, we proved that

$$\delta(\varphi(u), a) \equiv \varphi(v) \quad \text{iff} \quad [u]_{\rho_L} a \subseteq [v]_{\rho_L}.$$

The above shows that the transitions of D_{ρ_L} correspond to the transitions of $D/\equiv$. ∎

Theorem 5.14 shows that the DFA D_{ρ_L} is isomorphic to the DFA $D/\equiv$ obtained as the quotient of the DFA D modulo the equivalence relation $\equiv$ on Q. Since D_{ρ_L} is a minimal DFA accepting L, so is $D/\equiv$.

Example 5.13. Consider the following DFA D,

	a	b
1	2	3
2	4	4
3	4	3
4	5	5
5	5	5

with start state 1 and final states 2 and 3. It is easy to see that

$$L(D) = \{a\} \cup \{b^m \mid m \geq 1\}.$$

It is not hard to check that states 4 and 5 are equivalent, and no other pairs of distinct states are equivalent. The quotient DFA $D/\equiv$ is obtained my merging states 4 and 5, and we obtain the following minimal DFA:

	a	b
1	2	3
2	4	4
3	4	3
4	4	4

with start state 1 and final states 2 and 3. This DFA is isomorphic to the DFA D_{ρ_L} of Example 5.10.

There are other characterizations of the regular languages. Among those, the characterization in terms of right derivatives is of particular interest because it yields an alternative construction of minimal DFA's.

Definition 5.12. Given any language, $L \subseteq \Sigma^*$, for any string, $u \in \Sigma^*$, the *right derivative of L by u*, denoted L/u, is the language

$$L/u = \{w \in \Sigma^* \mid uw \in L\}.$$

Theorem 5.15. *If $L \subseteq \Sigma^*$ is any language, then L is regular iff it has finitely many right derivatives. Furthermore, if L is regular, then all its right derivatives are regular and their number is equal to the number of states of the minimal DFA's for L.*

Proof. It is easy to check that

$$L/u = L/v \qquad \text{iff} \qquad u\rho_L v.$$

The above shows that ρ_L has a finite number of classes, say m, iff there is a finite number of right derivatives, say n, and if so, $m = n$. If L is regular, then we know that the number of equivalence classes of ρ_L is the number of states of the minimal DFA's for L, so the number of right derivatives of L is equal to the size of the minimal DFA's for L.

Conversely, if the number of derivatives is finite, say m, then ρ_L has m classes and by Myhill-Nerode, L is regular. It remains to show that if L is regular then every right derivative is regular.

Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA accepting L. If $p = \delta^*(q_0, u)$, then let

$$D_p = (Q, \Sigma, \delta, p, F),$$

that is, D with with p as start state. It is clear that

$$L/u = L(D_p),$$

so L/u is regular for every $u \in \Sigma^*$. Also observe that if $|Q| = n$, then there are at most n DFA's D_p, so there is at most n right derivatives, which is another proof of the fact that a regular language has a finite number of right derivatives. ☐

If L is regular then the construction of a minimal DFA for L can be recast in terms of right derivatives. Let $L/u_1, L/u_2, \ldots, L/u_m$ be the set of all the right derivatives of L. Of course, we may assume that $u_1 = \epsilon$. We form a DFA whose states are the right derivatives, L/u_i. For every state, L/u_i, for every $a \in \Sigma$, there is a transition on input a from L/u_i to $L/u_j = L/(u_i a)$. The start state is $L = L/u_1$ and the final states are the right derivatives, L/u_i, for which $\epsilon \in L/u_i$.

We leave it as an exercise to check that the above DFA accepts L. One way to do this is to recall that $L/u = L/v$ iff $u\rho_L v$ and to observe that the above construction mimics the construction of D_{ρ_L} as in the Myhill-Nerode proposition (Proposition 5.9). This DFA is minimal since the number of right derivatives is equal to the size of the minimal DFA's for L.

We now return to state equivalence. Note that if $F = \emptyset$, then $\equiv$ has a single block (Q), and if $F = Q$, then $\equiv$ has a single block (F). In the first case, the minimal DFA is the one state DFA rejecting all strings. In the second case, the minimal DFA is the one state DFA accepting all strings. When $F \neq \emptyset$ and $F \neq Q$, there are at least two states in Q, and $\equiv$ also has at least two blocks, as we shall see shortly.

It remains to compute $\equiv$ explicitly. This is done using a sequence of approximations. In view of the previous discussion, we are assuming that $F \neq \emptyset$ and $F \neq Q$, which means that $n \geq 2$, where n is the number of states in Q.

Definition 5.13. Given any DFA $D = (Q, \Sigma, \delta, q_0, F)$, for every $i \geq 0$, the relation $\equiv_i$ on Q, called *i-state equivalence*, is defined as follows: for all $p, q \in Q$,

$$p \equiv_i q \quad \text{iff} \quad \forall w \in \Sigma^*,\ |w| \leq i\ (\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F).$$

When $p \equiv_i q$, we say that p *and* q *are i-indistinguishable*.

Since state equivalence $\equiv$ is defined such that

$$p \equiv q \quad \text{iff} \quad \forall w \in \Sigma^*(\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F),$$

we note that testing the condition

$$\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F$$

for all strings in Σ^* is equivalent to testing the above condition for all strings of length at most i for all $i \geq 0$, i.e.

$$p \equiv q \quad \text{iff} \quad \forall i \geq 0\, \forall w \in \Sigma^*,\ |w| \leq i\ (\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F).$$

Since $\equiv_i$ is defined such that

$$p \equiv_i q \quad \text{iff} \quad \forall w \in \Sigma^*,\ |w| \leq i\ (\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F),$$

we conclude that

$$p \equiv q \quad \text{iff} \quad \forall i \geq 0\, (p \equiv_i q).$$

This identity can also be expressed as

$$\equiv = \bigcap_{i \geq 0} \equiv_i\ .$$

If we assume that $F \neq \emptyset$ and $F \neq Q$, observe that $\equiv_0$ has exactly two equivalence classes F and $Q - F$, since ϵ is the only string of length 0, and since the condition

$$\delta^*(p, \epsilon) \in F \quad \text{iff} \quad \delta^*(q, \epsilon) \in F$$

is equivalent to the condition

$$p \in F \quad \text{iff} \quad q \in F.$$

It is also obvious from the definition of $\equiv_i$ that

$$\equiv \subseteq \cdots \subseteq \equiv_{i+1} \subseteq \equiv_i \subseteq \cdots \subseteq \equiv_1 \subseteq \equiv_0\ .$$

If this sequence was strictly decreasing for all $i \geq 0$, the partition associated with $\equiv_{i+1}$ would contain at least one more block than the partition associated with $\equiv_i$ and since we start with a partition with two blocks, the partition associated with $\equiv_i$ would have at least $i+2$ blocks.

But then, for $i = n-1$, the partition associated with $\equiv_{n-1}$ would have at least $n+1$ blocks, which is absurd since Q has only n states. Therefore, there is a smallest integer, $i_0 \leq n-2$, such that

$$\equiv_{i_0+1} = \equiv_{i_0} .$$

Thus, it remains to compute $\equiv_{i+1}$ from $\equiv_i$, which can be done using the following proposition: The proposition also shows that

$$\equiv = \equiv_{i_0} .$$

Proposition 5.16. *For any (trim) DFA $D = (Q, \Sigma, \delta, q_0, F)$, for all $p, q \in Q$, $p \equiv_{i+1} q$ iff $p \equiv_i q$ and $\delta(p, a) \equiv_i \delta(q, a)$, for every $a \in \Sigma$. Furthermore, if $F \neq \emptyset$ and $F \neq Q$, there is a smallest integer $i_0 \leq n-2$, such that*

$$\equiv_{i_0+1} = \equiv_{i_0} = \equiv .$$

Proof. By the definition of the relation $\equiv_i$,

$$p \equiv_{i+1} q \quad \text{iff} \quad \forall w \in \Sigma^*,\ |w| \leq i+1\, (\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F).$$

The trick is to observe that the condition

$$\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F$$

holds for all strings of length at most $i+1$ iff it holds for all strings of length at most i and for all strings of length between 1 and $i+1$. This is expressed as

$$\begin{aligned} p \equiv_{i+1} q \quad \text{iff} \quad & \forall w \in \Sigma^*,\ |w| \leq i\, (\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F) \\ & \text{and} \\ & \forall w \in \Sigma^*,\ 1 \leq |w| \leq i+1\, (\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F). \end{aligned}$$

Obviously, the first condition in the conjunction is $p \equiv_i q$, and since every string w such that $1 \leq |w| \leq i+1$ can be written as au where $a \in \Sigma$ and $0 \leq |u| \leq i$, the second condition in the conjunction can be written as

$$\forall a \in \Sigma\, \forall u \in \Sigma^*,\ |u| \leq i\, (\delta^*(p, au) \in F \quad \text{iff} \quad \delta^*(q, au) \in F).$$

However, $\delta^*(p, au) = \delta^*(\delta(p, a), u)$ and $\delta^*(q, au) = \delta^*(\delta(q, a), u)$, so that the above condition is really

$$\forall a \in \Sigma\, (\delta(p, a) \equiv_i \delta(q, a)).$$

Thus, we showed that

$$p \equiv_{i+1} q \quad \text{iff} \quad p \equiv_i q \quad \text{and} \quad \forall a \in \Sigma\, (\delta(p, a) \equiv_i \delta(q, a)).$$

We claim that if $\equiv_{i+1} = \equiv_i$ for some $i \geq 0$, then $\equiv_{i+j} = \equiv_i$ for all $j \geq 1$. This claim is proved by induction on j. For the base case j, the claim is that $\equiv_{i+1} = \equiv_i$, which is the hypothesis.

Assume inductively that $\equiv_{i+j} = \equiv_i$ for any $j \geq 1$. Since $p \equiv_{i+j+1} q$ iff $p \equiv_{i+j} q$ and $\delta(p, a) \equiv_{i+j} \delta(q, a)$, for every $a \in \Sigma$, and since by the induction hypothesis $\equiv_{i+j} = \equiv_i$, we obtain $p \equiv_{i+j+1} q$ iff $p \equiv_i q$ and $\delta(p, a) \equiv_i \delta(q, a)$, for every $a \in \Sigma$, which is equivalent to $p \equiv_{i+1} q$, and thus $\equiv_{i+j+1} = \equiv_{i+1}$. But $\equiv_{i+1} = \equiv_i$, so $\equiv_{i+j+1} = \equiv_i$, establishing the induction step.

Since

$$\equiv = \bigcap_{i \geq 0} \equiv_i, \quad \equiv_{i+1} \subseteq \equiv_i,$$

and since we know that there is a smallest index say i_0, such that $\equiv_j = \equiv_{i_0}$, for all $j \geq i_0 + 1$, we have $\equiv = \bigcap_{i=0}^{i_0} \equiv_i = \equiv_{i_0}$. $\square$

Using Proposition 5.16, we can compute $\equiv$ inductively, starting from $\equiv_0 = (F, Q - F)$, and computing $\equiv_{i+1}$ from $\equiv_i$, until the sequence of partitions associated with the $\equiv_i$ stabilizes.

Note that if $F = Q$ or $F = \emptyset$, then $\equiv = \equiv_0$, and the inductive characterization of Proposition 5.16 holds trivially.

There are a number of algorithms for computing $\equiv$, or to determine whether $p \equiv q$ for some given $p, q \in Q$.

A simple method to compute $\equiv$ is described in Hopcroft and Ullman. The basic idea is to propagate inequivalence, rather than equivalence.

The method consists in forming a triangular array corresponding to all unordered pairs (p, q), with $p \neq q$ (the rows and the columns of this triangular array are indexed by the states in Q, where the entries are below the descending diagonal). Initially, the entry (p, q) is marked iff p and q are **not 0-equivalent**, which means that p and q are not both in F or not both in $Q - F$.

Then, we process every unmarked entry on every row as follows: for any unmarked pair (p, q), we consider pairs $(\delta(p, a), \delta(q, a))$, for all $a \in \Sigma$. If any pair $(\delta(p, a), \delta(q, a))$ is already marked, this means that $\delta(p, a)$ and $\delta(q, a)$ are *inequivalent*, and thus p and q are *inequivalent*, and we mark the pair (p, q). We continue in this fashion, until at the end of a round during which all the rows are processed, nothing has changed. When the algorithm stops, all marked pairs are inequivalent, and all unmarked pairs correspond to equivalent states.

Let us illustrates the above method.

Example 5.14. Consider the following DFA accepting $\{a, b\}^*\{abb\}$:

	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

The start state is A, and the set of final states is $F = \{E\}$. (This is the DFA displayed in Figure 5.9.)

The initial (half) array is as follows, using $\times$ to indicate that the corresponding pair (say, (E, A)) consists of inequivalent states, and $\square$ to indicate that nothing is known yet.

$$\begin{array}{ccccc} B & \square & & & \\ C & \square & \square & & \\ D & \square & \square & \square & \\ E & \times & \times & \times & \times \\ & A & B & C & D \end{array}$$

After the first round, we have

$$\begin{array}{ccccc} B & \square & & & \\ C & \square & \square & & \\ D & \times & \times & \times & \\ E & \times & \times & \times & \times \\ & A & B & C & D \end{array}$$

After the second round, we have

$$\begin{array}{ccccc} B & \times & & & \\ C & \square & \times & & \\ D & \times & \times & \times & \\ E & \times & \times & \times & \times \\ & A & B & C & D \end{array}$$

Finally, nothing changes during the third round, and thus, only A and C are equivalent, and we get the four equivalence classes

$$(\{A, C\}, \{B\}, \{D\}, \{E\}).$$

We obtain the minimal DFA showed in Figure 5.19.

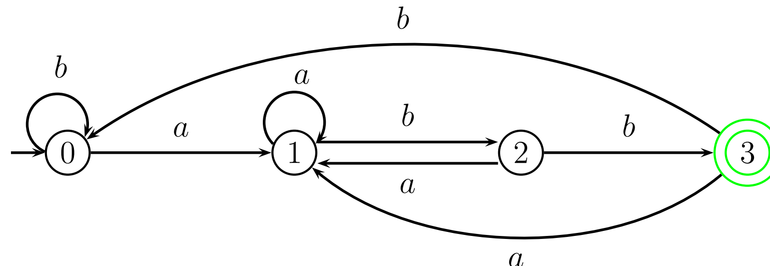


Figure 5.19: A minimal DFA accepting $\{a, b\}^*\{abb\}$

There are ways of improving the efficiency of this algorithm, see Hopcroft and Ullman for such improvements. Fast algorithms for testing whether $p \equiv q$ for some given $p, q \in Q$ also exist. One of these algorithms is based on "forward closures," following an idea of Knuth. Such an algorithm is related to a fast unification algorithm; see Section 5.13.

5.12 The Pumping Lemma

Another useful tool for proving that languages are not regular is the so-called *pumping lemma.*

Proposition 5.17. *(Pumping lemma) Given any DFA $D = (Q, \Sigma, \delta, q_0, F)$, there is some $m \geq 1$ such that for every $w \in \Sigma^*$, if $w \in L(D)$ and $|w| \geq m$, then there exists a decomposition of w as $w = uxv$, where*

(1) $x \neq \epsilon$,

(2) $ux^iv \in L(D)$, for all $i \geq 0$, and

(3) $|ux| \leq m$.

Moreover, m can be chosen to be the number of states of the DFA D.

Proof. Let m be the number of states in Q, and let $w = w_1 \ldots w_n$. Since Q contains the start state q_0, $m \geq 1$. Since $|w| \geq m$, we have $n \geq m$. Since $w \in L(D)$, let $(q_0, q_1, \ldots, q_n)$, be the sequence of states in the accepting computation of w (where $q_n \in F$). Consider the subsequence

$$(q_0, q_1, \ldots, q_m).$$

This sequence contains $m + 1$ states, but there are only m states in Q, and thus, we have $q_i = q_j$, for some i, j such that $0 \leq i < j \leq m$. Then, letting $u = w_1 \ldots w_i$, $x = w_{i+1} \ldots w_j$, and $v = w_{j+1} \ldots w_n$, it is clear that the conditions of the proposition hold. $\square$

An important consequence of the pumping lemma is that if a DFA D has m states and if there is some string $w \in L(D)$ such that $|w| \geq m$, then $L(D)$ is infinite.

Indeed, by the pumping lemma, $w \in L(D)$ can be written as $w = uxv$ with $x \neq \epsilon$, and

$$ux^i v \in L(D) \quad \text{for all } i \geq 0.$$

Since $x \neq \epsilon$, we have $|x| > 0$, so for all $i, j \geq 0$ with $i < j$ we have

$$|ux^i v| < |ux^i v| + (j - i)|x| = |ux^j v|,$$

which implies that $ux^i v \neq ux^j v$ for all $i < j$, and the set of strings

$$\{ux^i v \mid i \geq 0\} \subseteq L(D)$$

is an *infinite* subset of $L(D)$, which is itself infinite.

As a consequence, if $L(D)$ is finite, there are *no* strings w in $L(D)$ such that $|w| \geq m$. In this case, since the premise of the pumping lemma is false, the pumping lemma holds vacuously; that is, if $L(D)$ is finite, the pumping lemma yields no information.

Another corollary of the pumping lemma is that there is a test to decide whether a DFA D accepts an infinite language $L(D)$.

Proposition 5.18. *Let D be a DFA with m states, The language $L(D)$ accepted by D is infinite iff there is some string $w \in L(D)$ such that $m \leq |w| < 2m$.*

If $L(D)$ is infinite, there are strings of length $\geq m$ in $L(D)$, but a prirori there is no guarantee that there are "short" strings w in $L(D)$, that is, strings whose length is uniformly bounded by some function of m independent of D. The pumping lemma ensures that there are such strings, and the function is $m \mapsto 2m$.

Typically, the pumping lemma is used to prove that a language is not regular. The method is to proceed by contradiction, i.e., to assume (contrary to what we wish to prove) that a language L is indeed regular, and derive a contradiction of the pumping lemma. Thus, it would be helpful to see what the negation of the pumping lemma is, and for this, we first state the pumping lemma as a logical formula. We will use the following abbreviations:

$$\begin{aligned}
nat &= \{0, 1, 2, \ldots\}, \\
pos &= \{1, 2, \ldots\}, \\
A &\equiv w = uxv, \\
B &\equiv x \neq \epsilon, \\
C &\equiv |ux| \leq m, \\
P &\equiv \forall i \colon nat\, (ux^i v \in L(D)).
\end{aligned}$$

The pumping lemma can be stated as

$$\forall D\colon \mathrm{DFA}\, \exists m\colon pos\, \forall w\colon \Sigma^* \Big((w \in L(D) \wedge |w| \geq m) \supset (\exists u, x, v\colon \Sigma^*\, A \wedge B \wedge C \wedge P) \Big).$$

Recalling that

$$\neg(A \wedge B \wedge C \wedge P) \equiv \neg(A \wedge B \wedge C) \vee \neg P \equiv (A \wedge B \wedge C) \supset \neg P$$

and

$$\neg(R \supset S) \equiv R \wedge \neg S,$$

the negation of the pumping lemma can be stated as

$$\exists D\colon \mathrm{DFA}\, \forall m\colon pos\, \exists w\colon \Sigma^* \Big((w \in L(D) \wedge |w| \geq m) \wedge (\forall u, x, v\colon \Sigma^*\, (A \wedge B \wedge C) \supset \neg P) \Big).$$

Since

$$\neg P \equiv \exists i\colon nat\, (ux^i v \notin L(D)),$$

in order to show that the pumping lemma is contradicted, one needs to show that for some DFA D, for every $m \geq 1$, there is some string $w \in L(D)$ of length at least m, such that for every possible decomposition $w = uxv$ satisfying the constraints $x \neq \epsilon$ and $|ux| \leq m$, there is some $i \geq 0$ such that $ux^i v \notin L(D)$.

When proceeding by contradiction, we have a language L that we are (wrongly) assuming to be regular, and we can use any DFA D accepting L. The creative part of the argument is to pick the right $w \in L$ (not making any assumption on $m \leq |w|$).

As an illustration, let us use the pumping lemma to prove that $L_1 = \{a^n b^n \mid n \geq 1\}$ is not regular. The usefulness of the condition $|ux| \leq m$ lies in the fact that it reduces the number of legal decomposition uxv of w. We proceed by contradiction. Thus, let us assume that $L_1 = \{a^n b^n \mid n \geq 1\}$ is regular. If so, it is accepted by some DFA D. Now, we wish to contradict the pumping lemma. For every $m \geq 1$, let $w = a^m b^m$. Clearly, $w = a^m b^m \in L_1$ and $|w| \geq m$. Then, every legal decomposition u, x, v of w is such that

$$w = \underbrace{a \ldots a}_{u} \underbrace{a \ldots a}_{x} \underbrace{a \ldots a b \ldots b}_{v}$$

where $x \neq \epsilon$ and x ends within the a's, since $|ux| \leq m$. Since $x \neq \epsilon$, the string $uxxv$ is of the form $a^n b^m$ where $n > m$, and thus $uxxv \notin L_1$, contradicting the pumping lemma.

Let us consider two more examples. let $L_2 = \{a^m b^n \mid 1 \leq m < n\}$. We claim that L_2 is not regular. Our first proof uses the pumping lemma. For any $m \geq 1$, pick $w = a^m b^{m+1}$. We have $w \in L_2$ and $|w| \geq m$ so we need to contradict the pumping lemma. Every legal decomposition u, x, v of w is such that

$$w = \underbrace{a \ldots a}_{u} \underbrace{a \ldots a}_{x} \underbrace{a \ldots a b \ldots b}_{v}$$

where $x \neq \epsilon$ and x ends within the a's, since $|ux| \leq m$. Since $x \neq \epsilon$ and x consists of a's the string $ux^2v = uxxv$ contains at least $m+1$ a's and still $m+1$ b's, so $ux^2v \notin L_2$, contradicting the pumping lemma.

Our second proof uses Myhill-Nerode. Let $\simeq$ be a right-invariant equivalence relation of finite index such that L_2 is the union of classes of $\simeq$. If we consider the infinite sequence

$$a, a^2, \ldots, a^n, \ldots$$

since $\simeq$ has a finite number of classes there are two strings a^m and a^n with $m < n$ such that

$$a^m \simeq a^n.$$

By right-invariance by concatenating on the right with b^n we obtain

$$a^m b^n \simeq a^n b^n,$$

and since $m < n$ we have $a^m b^n \in L_2$ but $a^n b^n \notin L_2$, a contradiction.

Let us now consider the language $L_3 = \{a^m b^n \mid m \neq n\}$. This time let us begin by using Myhill-Nerode to prove that L_3 is not regular. The proof is the same as before, we obtain

$$a^m b^n \simeq a^n b^n,$$

and the contradiction is that $a^m b^n \in L_3$ and $a^n b^n \notin L_3$.

Let use now try to use the pumping lemma to prove that L_3 is not regular. For any $m \geq 1$ pick $w = a^m b^{m+1} \in L_3$. As in the previous case, every legal decomposition u, x, v of w is such that

$$w = \underbrace{a \ldots a}_{u} \underbrace{a \ldots a}_{x} \underbrace{a \ldots a b \ldots b}_{v}$$

where $x \neq \epsilon$ and x ends within the a's, since $|ux| \leq m$. However this time we have a problem, namely that we know that x is a nonempty string of a's but we don't know how many, so we can't guarantee that pumping up x will yield exactly the string $a^{m+1}b^{m+1}$. We made the wrong choice for w. There is a choice that will work but it is a bit tricky.

Fortunately, there is another simpler approach. Recall that the regular languages are closed under the boolean operations (union, intersection and complementation). Thus, L_3 is not regular iff its complement $\overline{L_3}$ is not regular. Observe that $\overline{L_3}$ contains $\{a^n b^n \mid n \geq 1\}$, which we showed to be nonregular. But there is another problem, which is that $\overline{L_3}$ contains other strings besides strings of the form $a^n b^n$, for example strings of the form $b^m a^n$ with $m, n > 0$.

Again, we can take care of this difficulty using the closure operations of the regular languages. If we can find a regular language R such that $\overline{L_3} \cap R$ is not regular, then $\overline{L_3}$ itself is not regular, since otherwise as $\overline{L_3}$ and R are regular then $\overline{L_3} \cap R$ is also regular. In our case, we can use $R = \{a\}^+\{b\}^+$ to obtain

$$\overline{L_3} \cap \{a\}^+\{b\}^+ = \{a^n b^n \mid n \geq 1\}.$$

Since $\{a^n b^n \mid n \geq 1\}$ is not regular, we reached our final contradiction. Observe how we use the language R to "clean up" $\overline{L}_3$ by intersecting it with R.

To complete a direct proof using the pumping lemma, the reader should try $w = a^{m!}b^{(m+1)!}$.

The use of the closure operations of the regular languages is often a quick way of showing that a language L is not regular by reducing the problem of proving that L is not regular to the problem of proving that some well-known language is not regular.

5.13 A Fast Algorithm for Checking State Equivalence Using a "Forward-Closure"

Given two states $p, q \in Q$, if $p \equiv q$, then we know that $\delta(p, a) \equiv \delta(q, a)$, for all $a \in \Sigma$. This suggests a method for testing whether two distinct states p, q are equivalent. Starting with the relation $R = \{(p, q)\}$, construct the smallest equivalence relation $R^\dagger$ containing R with the property that whenever $(r, s) \in R^\dagger$, then $(\delta(r, a), \delta(s, a)) \in R^\dagger$, for all $a \in \Sigma$. If we ever encounter a pair (r, s) such that $r \in F$ and $s \in \overline{F}$, or $r \in \overline{F}$ and $s \in F$, then r and s are inequivalent, and so are p and q. Otherwise, it can be shown that p and q are indeed equivalent. Thus, testing for the equivalence of two states reduces to finding an efficient method for computing the "forward closure" of a relation defined on the set of states of a DFA.

Such a method was worked out by John Hopcroft and Richard Karp and published in a 1971 Cornell technical report. This method is based on an idea of Donald Knuth for solving Exercise 11, in Section 2.3.5 of *The Art of Computer Programming*, Vol. 1, second edition, 1973. A sketch of the solution for this exercise is given on page 594. As far as I know, Hopcroft and Karp's method was never published in a journal, but a simple recursive algorithm does appear on page 144 of Aho, Hopcroft and Ullman's *The Design and Analysis of Computer Algorithms*, first edition, 1974. Essentially the same idea was used by Paterson and Wegman to design a fast unification algorithm (in 1978). We make a few definitions.

A relation $S \subseteq Q \times Q$ is a *forward closure* iff it is an equivalence relation and whenever $(r, s) \in S$, then $(\delta(r, a), \delta(s, a)) \in S$, for all $a \in \Sigma$. The *forward closure* of a relation $R \subseteq Q \times Q$ is the smallest equivalence relation $R^\dagger$ containing R which is forward closed.

We say that a forward closure S is *good* iff whenever $(r, s) \in S$, then $good(r, s)$, where $good(r, s)$ holds iff either both $r, s \in F$, or both $r, s \notin F$. Obviously, $bad(r, s)$ iff $\neg good(r, s)$.

Given any relation $R \subseteq Q \times Q$, recall that the smallest equivalence relation $R_\approx$ containing R is the relation $(R \cup R^{-1})^*$ (where $R^{-1} = \{(q, p) \mid (p, q) \in R\}$, and $(R \cup R^{-1})^*$ is the reflexive and transitive closure of $(R \cup R^{-1})$). The forward closure of R can be computed inductively by defining the sequence of relations $R_i \subseteq Q \times Q$ as follows:

$$R_0 = R_{\approx}$$
$$R_{i+1} = (R_i \cup \{(\delta(r,a), \delta(s,a)) \mid (r,s) \in R_i,\ a \in \Sigma\})_{\approx}.$$

It is not hard to prove that $R_{i_0+1} = R_{i_0}$ for some least i_0, and that $R^\dagger = R_{i_0}$ is the smallest forward closure containing R. The following two facts can also been established.

(a) if $R^\dagger$ is good, then
$$R^\dagger \subseteq \equiv . \tag{5.1}$$

(b) if $p \equiv q$, then
$$R^\dagger \subseteq \equiv,$$
that is, equation (5.1) holds. This implies that $R^\dagger$ is good.

As a consequence, we obtain the correctness of our procedure: $p \equiv q$ iff the forward closure $R^\dagger$ of the relation $R = \{(p,q)\}$ is good.

In practice, we maintain a partition Π representing the equivalence relation that we are closing under forward closure. We add each new pair $(\delta(r,a), \delta(s,a))$ one at a time, and immediately form the smallest equivalence relation containing the new relation. If $\delta(r,a)$ and $\delta(s,a)$ already belong to the same block of Π, we consider another pair, else we merge the blocks corresponding to $\delta(r,a)$ and $\delta(s,a)$, and then consider another pair.

The algorithm is recursive, but it can easily be implemented using a stack. To manipulate partitions efficiently, we represent them as lists of trees (forests). Each equivalence class C in the partition Π is represented by a tree structure consisting of nodes and parent pointers, with the pointers from the sons of a node to the node itself. The root has a null pointer. Each node also maintains a counter keeping track of the number of nodes in the subtree rooted at that node.

Note that pointers can be avoided. We can represent a forest of n nodes as a list of n pairs of the form $(\mathit{father}, \mathit{count})$. If $(\mathit{father}, \mathit{count})$ is the ith pair in the list, then $\mathit{father} = 0$ iff node i is a root node, otherwise, *father* is the index of the node in the list which is the parent of node i. The number *count* is the total number of nodes in the tree rooted at the ith node.

For example, the following list of nine nodes

$$((0,3), (0,2), (1,1), (0,2), (0,2), (1,1), (2,1), (4,1), (5,1))$$

represents a forest consisting of the following four trees:

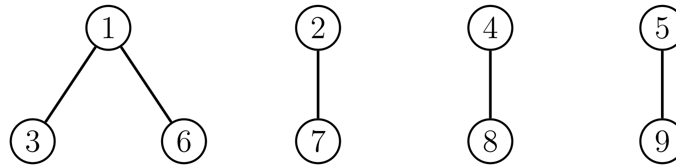


Figure 5.20: A forest of four trees

Two functions *union* and *find* are defined as follows. Given a state p, $\mathit{find}(p, \Pi)$ finds the root of the tree containing p as a node (not necessarily a leaf). Given two root nodes p, q, $\mathit{union}(p, q, \Pi)$ forms a new partition by merging the two trees with roots p and q as follows: if the counter of p is smaller than that of q, then let the root of p point to q, else let the root of q point to p.

For example, given the two trees shown on the left in Figure 5.21, $\mathit{find}(6, \Pi)$ returns 3 and $\mathit{find}(8, \Pi)$ returns 4. Then $\mathit{union}(3, 4, \Pi)$ yields the tree shown on the right in Figure 5.21.

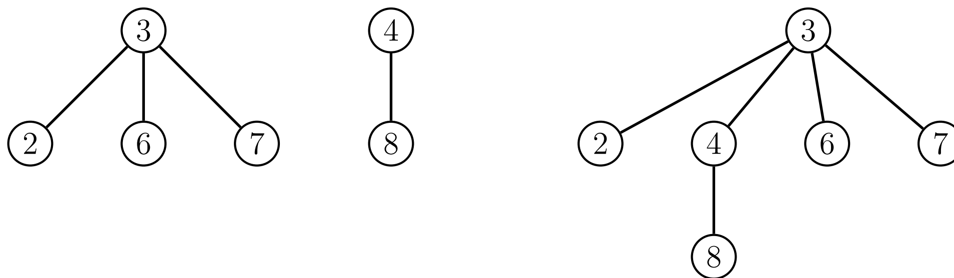


Figure 5.21: Applying the function *union* to the trees rooted at 3 and 4

In order to speed up the algorithm, using an idea due to Tarjan, we can modify *find* as follows: during a call $\mathit{find}(p, \Pi)$, as we follow the path from p to the root r of the tree containing p, we redirect the parent pointer of every node q on the path from p (including p itself) to r (we perform *path compression*). For example, applying $\mathit{find}(8, \Pi)$ to the tree shown on the right in Figure 5.21 yields the tree shown in Figure 5.22

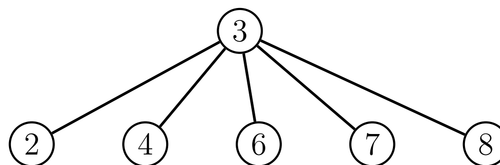


Figure 5.22: The result of applying *find* with path compression

Then, the algorithm is as follows:

```
function unif[p, q, Π, dd]: flag;
  begin
    trans := left(dd); ff := right(dd); pq := (p, q); st := (pq); flag := 1;
    k := Length(first(trans));
    while st ≠ () ∧ flag ≠ 0 do
      uv := top(st); uu := left(uv); vv := right(uv);
      pop(st);
      if bad(ff, uv) = 1 then flag := 0
      else
        u := find(uu, Π); v := find(vv, Π);
        if u ≠ v then
          union(u, v, Π);
          for i = 1 to k do
            u1 := delta(trans, uu, k − i + 1); v1 := delta(trans, vv, k − i + 1);
            uv := (u1, v1); push(st, uv)
          endfor
        endif
      endif
    endwhile
  end
```

The initial partition Π is the identity relation on Q, i.e., it consists of blocks $\{q\}$ for all states $q \in Q$. The algorithm uses a stack *st*. We are assuming that the DFA *dd* is specified by a list of two sublists, the first list, denoted *left*(*dd*) in the pseudo-code above, being a representation of the transition function, and the second one, denoted *right*(*dd*), the set of final states. The transition function itself is a list of lists, where the i-th list represents the i-th row of the transition table for *dd*. The function *delta* is such that $delta(trans, i, j)$ returns the j-th state in the i-th row of the transition table of *dd*. For example, we have the DFA

$$dd = (((2,3),(2,4),(2,3),(2,5),(2,3),(7,6),(7,8),(7,9),(7,6)),(5,9))$$

consisting of 9 states labeled $1, \ldots, 9$, and two final states 5 and 9 shown in Figure 5.23. Also, the alphabet has two letters, since every row in the transition table consists of two entries. For example, the two transitions from state 3 are given by the pair $(2, 3)$, which indicates that $\delta(3, a) = 2$ and $\delta(3, b) = 3$.

The sequence of steps performed by the algorithm starting with $p = 1$ and $q = 6$ is shown below. At every step, we show the current pair of states, the partition, and the stack.

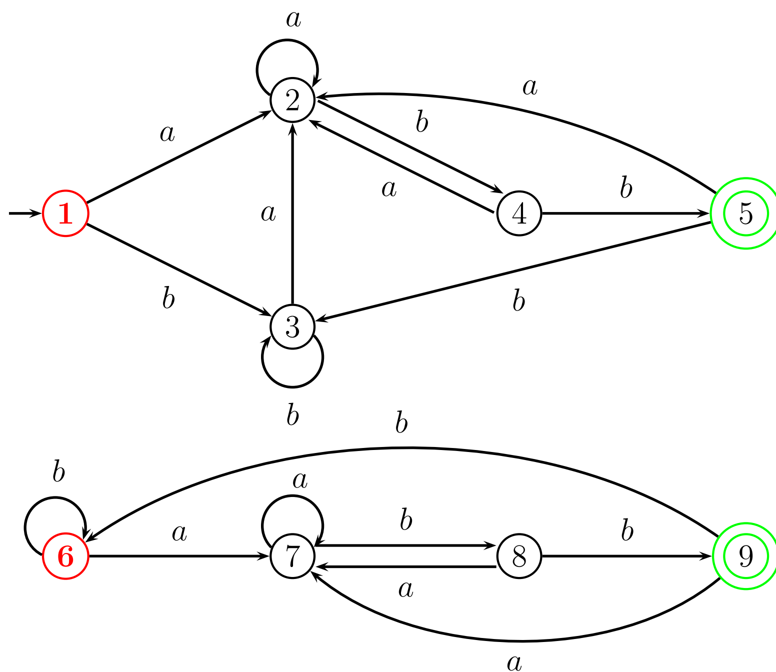


Figure 5.23: Testing state equivalence in a DFA

$p = 1, q = 6,\ \Pi = \{\{1, 6\}, \{2\}, \{3\}, \{4\}, \{5\}, \{7\}, \{8\}, \{9\}\},\ st = \{\{1, 6\}\}$

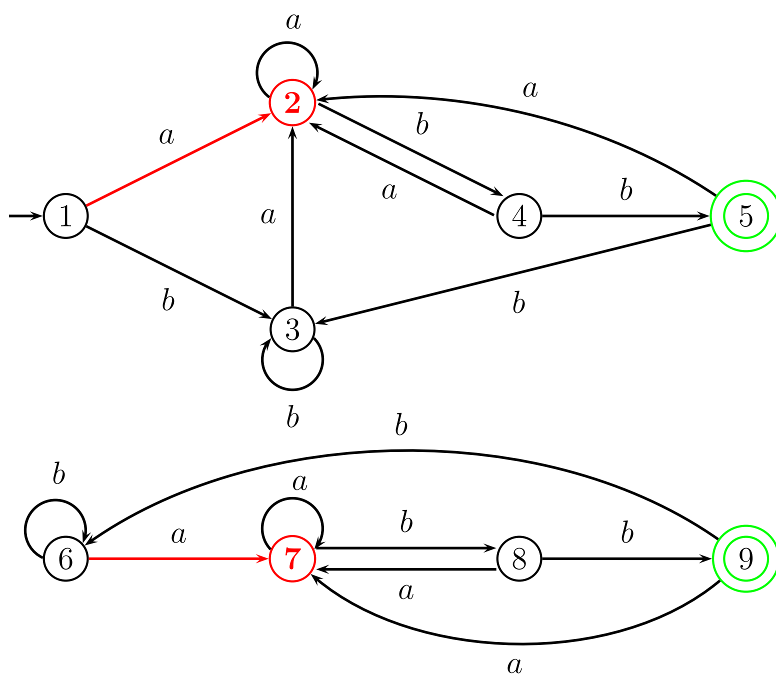


Figure 5.24: Testing state equivalence in a DFA

$p = 2, q = 7,\ \Pi = \{\{1, 6\}, \{2, 7\}, \{3\}, \{4\}, \{5\}, \{8\}, \{9\}\},\ st = \{\{3, 6\}, \{2, 7\}\}$

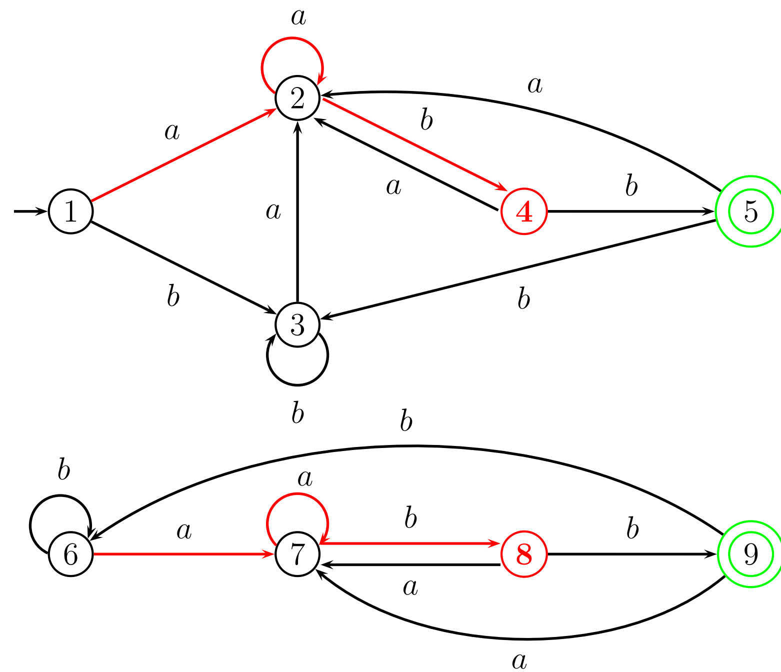


Figure 5.25: Testing state equivalence in a DFA

$p = 4, q = 8,\ \Pi = \{\{1, 6\}, \{2, 7\}, \{3\}, \{4, 8\}, \{5\}, \{9\}\},\ st = \{\{3, 6\}, \{4, 8\}\}$

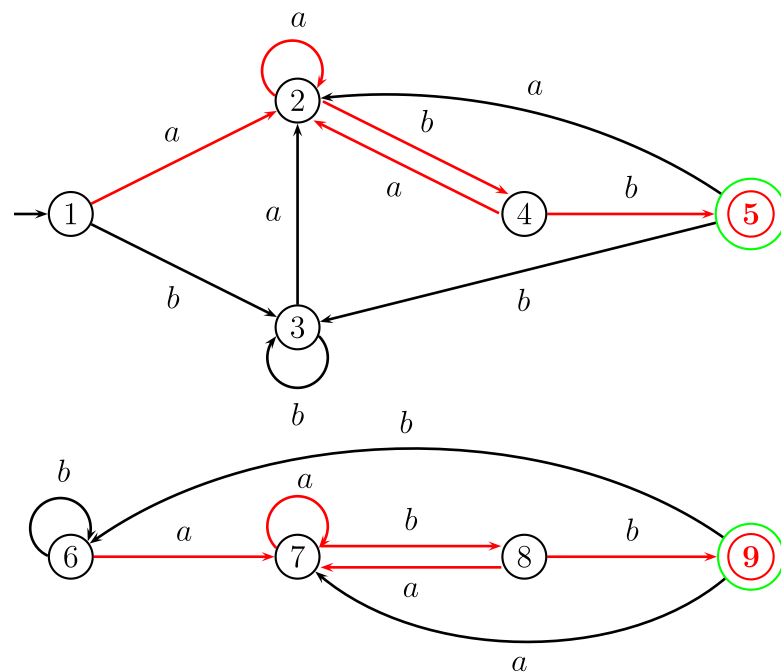


Figure 5.26: Testing state equivalence in a DFA

$p = 5, q = 9,\ \Pi = \{\{1, 6\}, \{2, 7\}, \{3\}, \{4, 8\}, \{5, 9\}\},\ st = \{\{3, 6\}, \{5, 9\}\}$

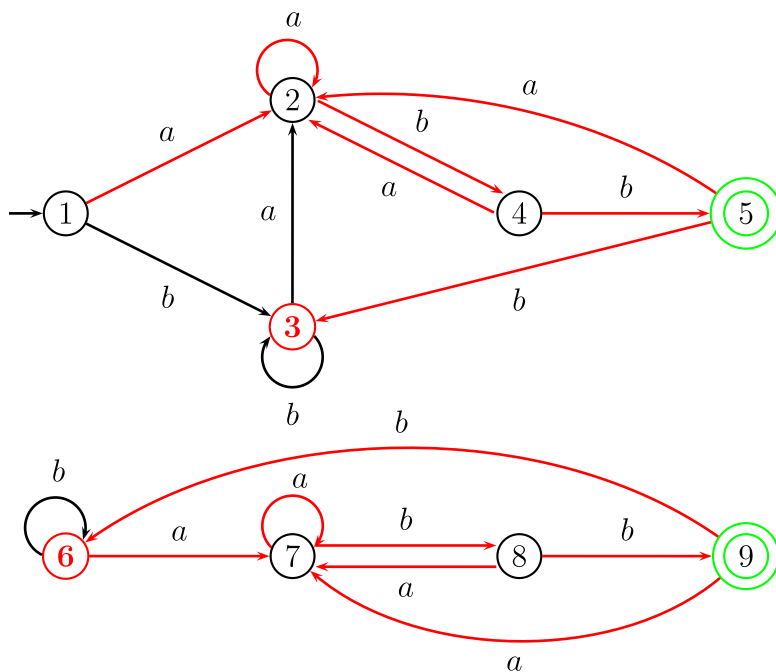


Figure 5.27: Testing state equivalence in a DFA

$p = 3, q = 6,\ \Pi = \{\{1,3,6\},\{2,7\},\{4,8\},\{5,9\}\},\ st = \{\{3,6\},\{3,6\}\}$

Since states 3 and 6 belong to the first block of the partition, the algorithm terminates. Since no block of the partition contains a bad pair, the states $p = 1$ and $q = 6$ are equivalent.

Let us now test whether the states $p = 3$ and $q = 7$ are equivalent.

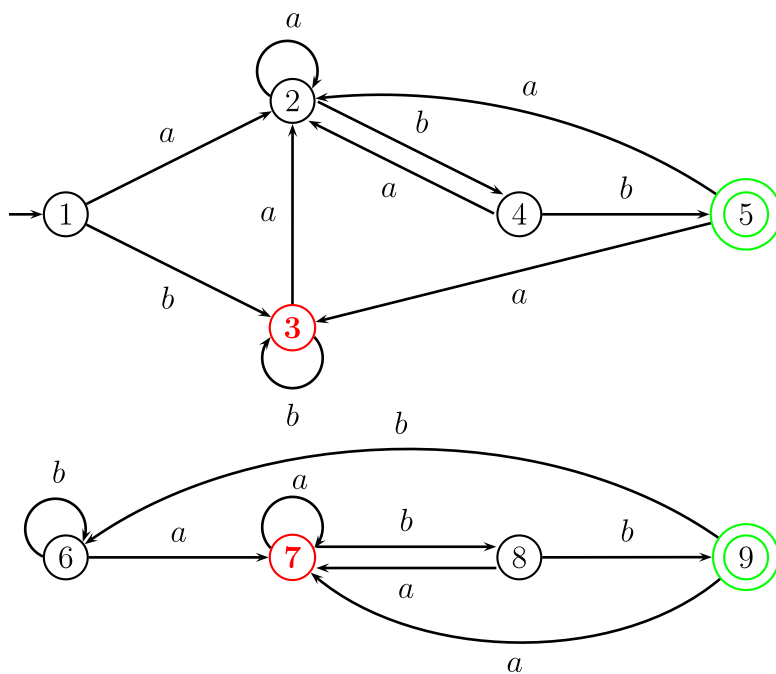


Figure 5.28: Testing state equivalence in a DFA

$p = 3, q = 7,\ \Pi = \{\{1\}, \{2\}, \{3, 7\}, \{4\}, \{5\}, \{6\}, \{8\}, \{9\}\},\ st = \{\{3, 7\}\}$

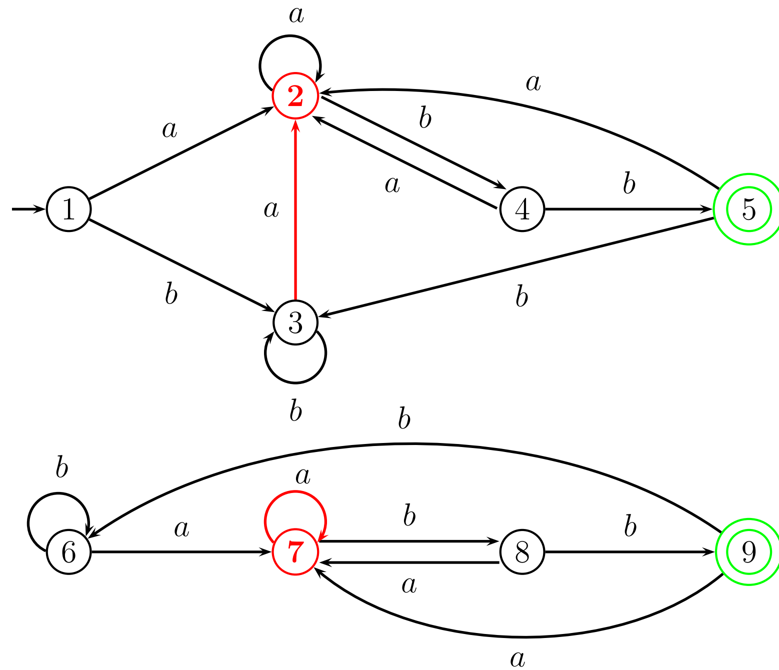


Figure 5.29: Testing state equivalence in a DFA

$p = 2, q = 7,\ \Pi = \{\{1\}, \{2, 3, 7\}, \{4\}, \{5\}, \{6\}, \{8\}, \{9\}\},\ st = \{\{3, 8\}, \{2, 7\}\}$

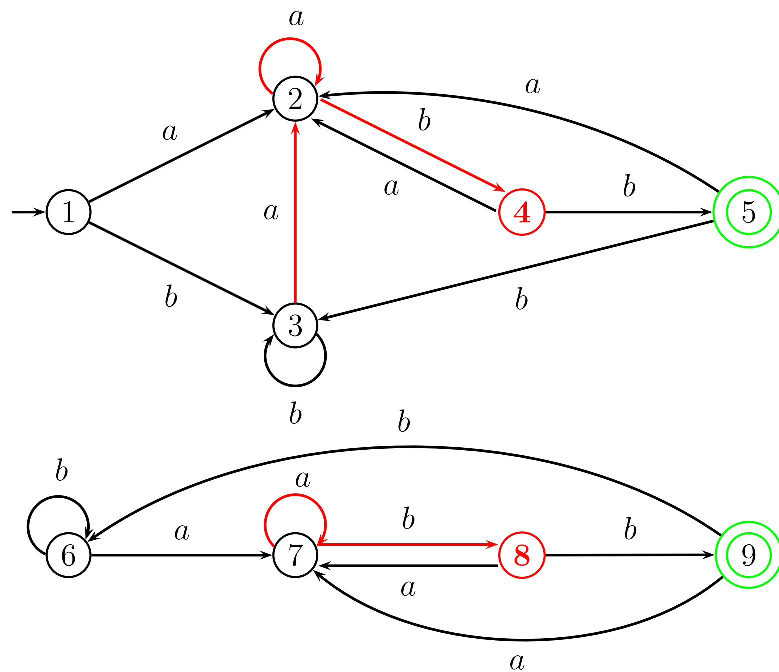


Figure 5.30: Testing state equivalence in a DFA

$p = 4, q = 8,\ \Pi = \{\{1\}, \{2, 3, 7\}, \{4, 8\}, \{5\}, \{6\}, \{9\}\},\ st = \{\{3, 8\}, \{4, 8\}\}$

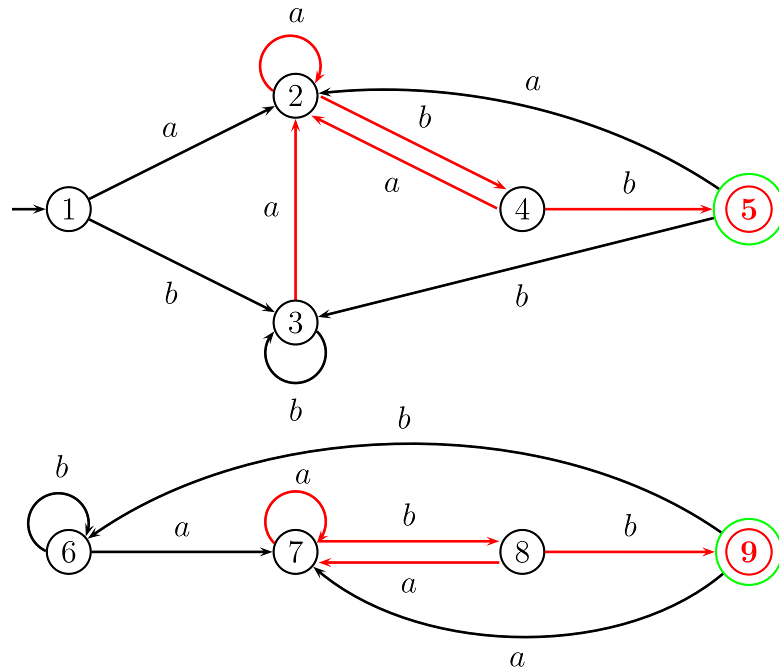


Figure 5.31: Testing state equivalence in a DFA

$p = 5, q = 9,\ \Pi = \{\{1\}, \{2, 3, 7\}, \{4, 8\}, \{5, 9\}, \{6\}\},\ st = \{\{3, 8\}, \{5, 9\}\}$

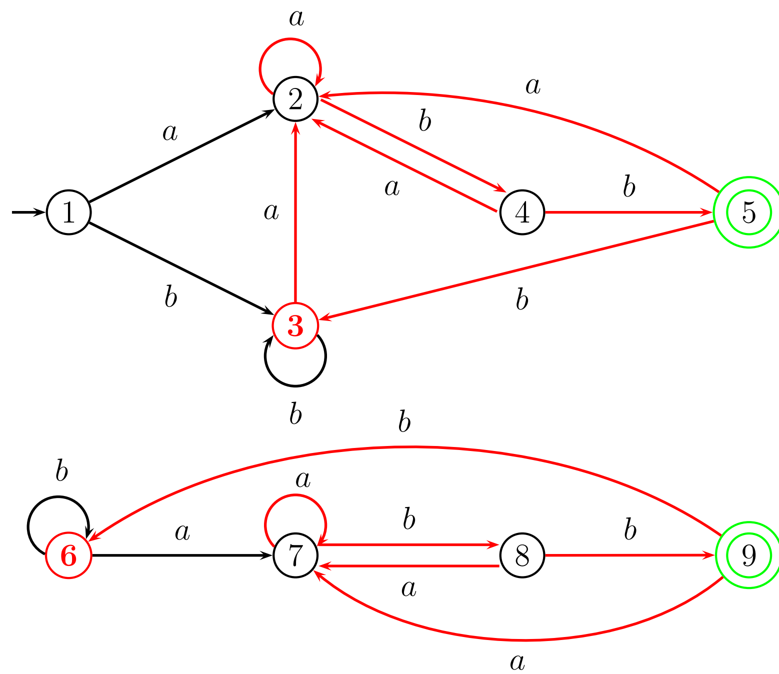


Figure 5.32: Testing state equivalence in a DFA

$p = 3, q = 6,\ \Pi = \{\{1\}, \{2, 3, 6, 7\}, \{4, 8\}, \{5, 9\}\},\ st = \{\{3, 8\}, \{3, 6\}\}$

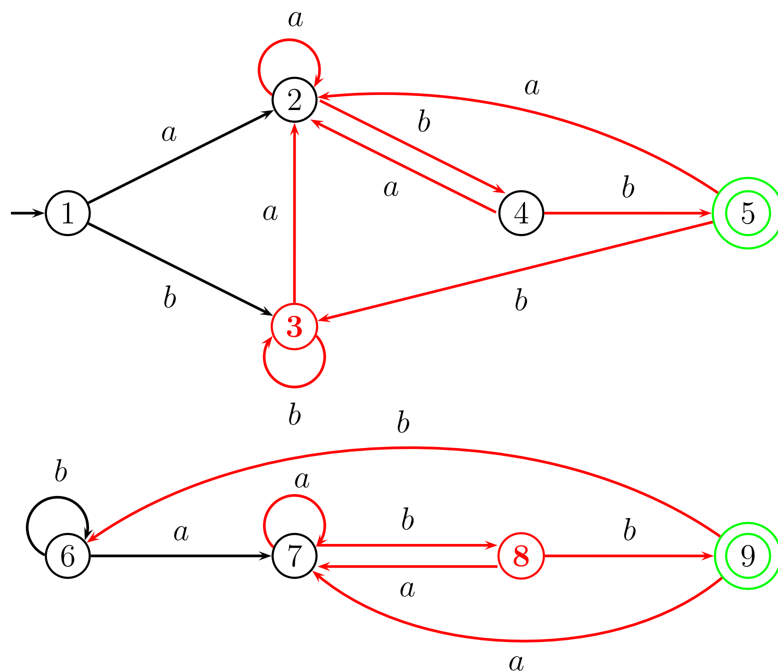


Figure 5.33: Testing state equivalence in a DFA

$p = 3, q = 8,\ \Pi = \{\{1\}, \{2, 3, 4, 6, 7, 8\}, \{5, 9\}\},\ st = \{\{3, 8\}\}$

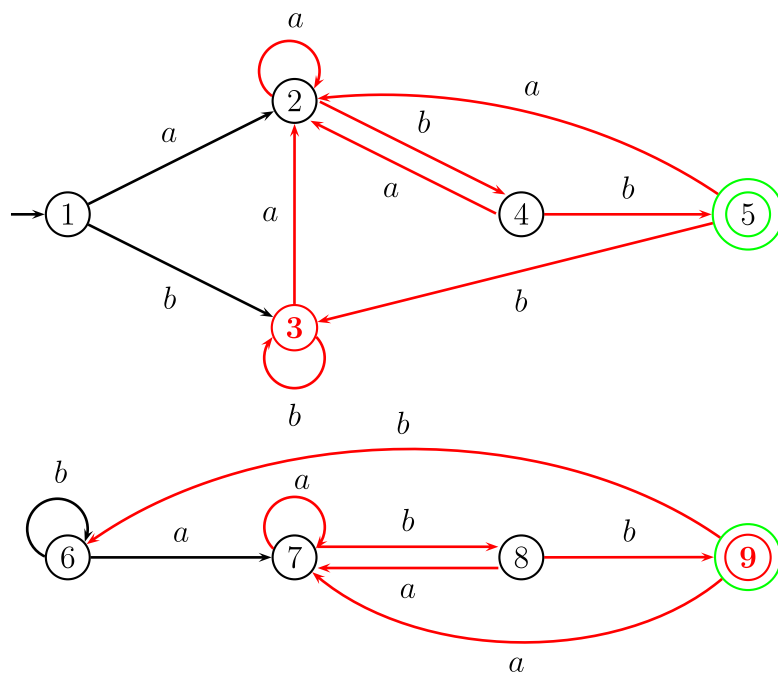


Figure 5.34: Testing state equivalence in a DFA

$p = 3, q = 9,\ \Pi = \{\{1\}, \{2, 3, 4, 6, 7, 8\}, \{5, 9\}\},\ st = \{\{3, 9\}\}$

Since the pair $(3, 9)$ is a bad pair, the algorithm stops, and the states $p = 3$ and $q = 7$ are inequivalent.

Chapter 6

Context-Free Grammars, Context-Free Languages, Parse Trees and Ogden's Lemma

6.1 Context-Free Grammars

A context-free grammar basically consists of a finite set of grammar rules. In order to define grammar rules, we assume that we have two kinds of symbols: the terminals, which are the symbols of the alphabet underlying the languages under consideration, and the nonterminals, which behave like variables ranging over strings of terminals. A rule is of the form $A \to \alpha$, where A is a single nonterminal, and the right-hand side α is a string of terminal and/or nonterminal symbols. As usual, first we need to define what the object is (a context-free grammar), and then we need to explain how it is used. Unlike automata, grammars are used to *generate* strings, rather than recognize strings.

Definition 6.1. A *context-free grammar (for short, CFG)* is a quadruple $G = (V, \Sigma, P, S)$, where

- V is a finite set of symbols called the *vocabulary (or set of grammar symbols)*;
- $\Sigma \subseteq V$ is the set of *terminal symbols (for short, terminals)*;
- $S \in (V - \Sigma)$ is a designated symbol called the *start symbol*;
- $P \subseteq (V - \Sigma) \times V^*$ is a finite set of *productions (or rewrite rules, or rules)*.

The set $N = V - \Sigma$ is called the set of *nonterminal symbols (for short, nonterminals)*. Thus, $P \subseteq N \times V^*$, and every production $\langle A, \alpha \rangle$ is also denoted as $A \to \alpha$. A production of the form $A \to \epsilon$ is called an *epsilon rule, or null rule*.

Remark: Context-free grammars are sometimes defined as $G = (V_N, V_T, P, S)$. The correspondence with our definition is that $\Sigma = V_T$ and $N = V_N$, so that $V = V_N \cup V_T$. Thus, in this other definition, it is necessary to assume that $V_T \cap V_N = \emptyset$.

Example 1. $G_1 = (\{E, a, b\}, \{a, b\}, P, E)$, where P is the set of rules

$$\begin{aligned} E &\longrightarrow aEb, \\ E &\longrightarrow ab. \end{aligned}$$

As we will see shortly, this grammar generates the language $L_1 = \{a^n b^n \mid n \geq 1\}$, which is not regular.

Example 2. $G_2 = (\{E, +, *, (,), a\}, \{+, *, (,), a\}, P, E)$, where P is the set of rules

$$\begin{aligned} E &\longrightarrow E + E, \\ E &\longrightarrow E * E, \\ E &\longrightarrow (E), \\ E &\longrightarrow a. \end{aligned}$$

This grammar generates a set of arithmetic expressions.

6.2 Derivations and Context-Free Languages

The productions of a grammar are used to derive strings. In this process, the productions are used as rewrite rules. Formally, we define the derivation relation associated with a context-free grammar. First, let us review the concepts of transitive closure and reflexive and transitive closure of a binary relation.

Given a set A, a *binary relation R on A* is any set of ordered pairs, i.e. $R \subseteq A \times A$. For short, instead of binary relation, we often simply say relation. Given any two relations R, S on A, their *composition* $R \circ S$ is defined as

$$R \circ S = \{(x, y) \in A \times A \mid \exists z \in A, (x, z) \in R \text{ and } (z, y) \in S\}.$$

The *identity relation I_A on A* is the relation I_A defined such that

$$I_A = \{(x, x) \mid x \in A\}.$$

For short, we often denote I_A as I. Note that

$$R \circ I = I \circ R = R$$

for every relation R on A. Given a relation R on A, for any $n \geq 0$ we define R^n as follows:

$$\begin{aligned} R^0 &= I, \\ R^{n+1} &= R^n \circ R. \end{aligned}$$

It is obvious that $R^1 = R$. It is also easily verified by induction that $R^n \circ R = R \circ R^n$. The *transitive closure R^+ of the relation R* is defined as

$$R^+ = \bigcup_{n \geq 1} R^n.$$

It is easily verified that R^+ is the smallest transitive relation containing R, and that $(x, y) \in R^+$ iff there is some $n \geq 1$ and some $x_0, x_1, \ldots, x_n \in A$ such that $x_0 = x$, $x_n = y$, and $(x_i, x_{i+1}) \in R$ for all i, $0 \leq i \leq n - 1$. The *transitive and reflexive closure R^* of the relation R* is defined as

$$R^* = \bigcup_{n \geq 0} R^n.$$

Clearly, $R^* = R^+ \cup I$. It is easily verified that R^* is the smallest transitive and reflexive relation containing R.

Definition 6.2. Given a context-free grammar $G = (V, \Sigma, P, S)$, the (one-step) *derivation relation $\Longrightarrow_G$ associated with G* is the binary relation $\Longrightarrow_G \subseteq V^* \times V^*$ defined as follows: for all $\alpha, \beta \in V^*$, we have

$$\alpha \Longrightarrow_G \beta$$

iff there exist $\lambda, \rho \in V^*$, and some production $(A \to \gamma) \in P$, such that

$$\alpha = \lambda A \rho \quad \text{and} \quad \beta = \lambda \gamma \rho.$$

The transitive closure of $\Longrightarrow_G$ is denoted as $\stackrel{+}{\Longrightarrow}_G$ and the reflexive and transitive closure of $\Longrightarrow_G$ is denoted as $\stackrel{*}{\Longrightarrow}_G$.

When the grammar G is clear from the context, we usually omit the subscript G in $\Longrightarrow_G$, $\stackrel{+}{\Longrightarrow}_G$, and $\stackrel{*}{\Longrightarrow}_G$.

A string $\alpha \in V^*$ such that $S \stackrel{*}{\Longrightarrow} \alpha$ is called a *sentential form*, and a string $w \in \Sigma^*$ such that $S \stackrel{*}{\Longrightarrow} w$ is called a *sentence*. A derivation $\alpha \stackrel{*}{\Longrightarrow} \beta$ involving n steps is denoted as $\alpha \stackrel{n}{\Longrightarrow} \beta$.

Note that a derivation step

$$\alpha \Longrightarrow_G \beta$$

is rather nondeterministic. Indeed, one can choose among various occurrences of nonterminals A in α, and also among various productions $A \to \gamma$ with left-hand side A.

For example, using the grammar $G_1 = (\{E, a, b\}, \{a, b\}, P, E)$, where P is the set of rules

$$E \longrightarrow aEb,$$
$$E \longrightarrow ab,$$

every derivation from E is of the form

$$E \stackrel{*}{\Longrightarrow} a^n E b^n \Longrightarrow a^n abb^n = a^{n+1} b^{n+1},$$

or

$$E \stackrel{*}{\Longrightarrow} a^n E b^n \Longrightarrow a^n aEbb^n = a^{n+1} E b^{n+1},$$

where $n \geq 0$.

Grammar G_1 is very simple: every string $a^n b^n$ has a unique derivation. This is usually not the case. For example, using the grammar $G_2 = (\{E, +, *, (,), a\}, \{+, *, (,), a\}, P, E)$, where P is the set of rules

$$\begin{aligned} E &\longrightarrow E + E, \\ E &\longrightarrow E * E, \\ E &\longrightarrow (E), \\ E &\longrightarrow a, \end{aligned}$$

the string $a + a * a$ has the following distinct derivations, where the boldface indicates which occurrence of E is rewritten:

$$\begin{aligned} &\mathbf{E} \Longrightarrow \mathbf{E} * E \Longrightarrow \mathbf{E} + E * E \\ &\Longrightarrow a + \mathbf{E} * E \Longrightarrow a + a * \mathbf{E} \Longrightarrow a + a * a, \end{aligned}$$

and

$$\begin{aligned} &\mathbf{E} \Longrightarrow \mathbf{E} + E \Longrightarrow a + \mathbf{E} \\ &\Longrightarrow a + \mathbf{E} * E \Longrightarrow a + a * \mathbf{E} \Longrightarrow a + a * a. \end{aligned}$$

In the above derivations, the leftmost occurrence of a nonterminal is chosen at each step. Such derivations are called *leftmost derivations*. We could systematically rewrite the rightmost occurrence of a nonterminal, getting *rightmost derivations*. The string $a + a * a$ also has the following two rightmost derivations, where the boldface indicates which occurrence of E is rewritten:

$$\begin{aligned} &\mathbf{E} \Longrightarrow E + \mathbf{E} \Longrightarrow E + E * \mathbf{E} \\ &\Longrightarrow E + \mathbf{E} * a \Longrightarrow \mathbf{E} + a * a \Longrightarrow a + a * a, \end{aligned}$$

and

$$\begin{aligned} &\mathbf{E} \Longrightarrow E * \mathbf{E} \Longrightarrow \mathbf{E} * a \\ &\Longrightarrow E + \mathbf{E} * a \Longrightarrow \mathbf{E} + a * a \Longrightarrow a + a * a. \end{aligned}$$

The language generated by a context-free grammar is defined as follows.

Definition 6.3. Given a context-free grammar $G = (V, \Sigma, P, S)$, the *language generated by* G is the set

$$L(G) = \{w \in \Sigma^* \mid S \stackrel{+}{\Longrightarrow} w\}.$$

A language $L \subseteq \Sigma^*$ is a *context-free language (for short, CFL)* iff $L = L(G)$ for some context-free grammar G.

It is technically very useful to consider derivations in which the leftmost nonterminal is always selected for rewriting, and dually, derivations in which the rightmost nonterminal is always selected for rewriting.

Definition 6.4. Given a context-free grammar $G = (V, \Sigma, P, S)$, the (one-step) *leftmost derivation relation* $\underset{lm}{\Longrightarrow}$ *associated with* G is the binary relation $\underset{lm}{\Longrightarrow} \subseteq V^* \times V^*$ defined as follows: for all $\alpha, \beta \in V^*$, we have

$$\alpha \underset{lm}{\Longrightarrow} \beta$$

iff there exist $u \in \Sigma^*$, $\rho \in V^*$, and some production $(A \to \gamma) \in P$, such that

$$\alpha = uA\rho \quad \text{and} \quad \beta = u\gamma\rho.$$

The transitive closure of $\underset{lm}{\Longrightarrow}$ is denoted as $\underset{lm}{\stackrel{+}{\Longrightarrow}}$ and the reflexive and transitive closure of $\underset{lm}{\Longrightarrow}$ is denoted as $\underset{lm}{\stackrel{*}{\Longrightarrow}}$. The (one-step) *rightmost derivation relation* $\underset{rm}{\Longrightarrow}$ *associated with* G is the binary relation $\underset{rm}{\Longrightarrow} \subseteq V^* \times V^*$ defined as follows: for all $\alpha, \beta \in V^*$, we have

$$\alpha \underset{rm}{\Longrightarrow} \beta$$

iff there exist $\lambda \in V^*$, $v \in \Sigma^*$, and some production $(A \to \gamma) \in P$, such that

$$\alpha = \lambda Av \quad \text{and} \quad \beta = \lambda\gamma v.$$

The transitive closure of $\underset{rm}{\Longrightarrow}$ is denoted as $\underset{rm}{\stackrel{+}{\Longrightarrow}}$ and the reflexive and transitive closure of $\underset{rm}{\Longrightarrow}$ is denoted as $\underset{rm}{\stackrel{*}{\Longrightarrow}}$.

Remarks: It is customary to use the symbols a, b, c, d, e for terminal symbols, and the symbols A, B, C, D, E for nonterminal symbols. The symbols u, v, w, x, y, z denote terminal strings, and the symbols $\alpha, \beta, \gamma, \lambda, \rho, \mu$ denote strings in V^*. The symbols X, Y, Z usually denote symbols in V.

Given a context-free grammar $G = (V, \Sigma, P, S)$, *parsing a string* w consists in finding out whether $w \in L(G)$, and if so, in producing a derivation for w. The following proposition is technically very important. It shows that leftmost and rightmost derivations are "universal". This has some important practical implications for the complexity of parsing algorithms.

Proposition 6.1. *Let $G = (V, \Sigma, P, S)$ be a context-free grammar. For every $w \in \Sigma^*$, for every derivation $S \stackrel{+}{\Longrightarrow} w$, there is a leftmost derivation $S \underset{lm}{\stackrel{+}{\Longrightarrow}} w$, and there is a rightmost derivation $S \underset{rm}{\stackrel{+}{\Longrightarrow}} w$.*

Proof. Of course, we have to somehow use induction on derivations, but this is a little tricky, and it is necessary to prove a stronger fact. We treat leftmost derivations, rightmost derivations being handled in a similar way.

Claim: For every $w \in \Sigma^*$, for every $\alpha \in V^+$, for every $n \geq 1$, if $\alpha \stackrel{n}{\Longrightarrow} w$, then there is a leftmost derivation $\alpha \underset{lm}{\stackrel{n}{\Longrightarrow}} w$.

The claim is proved by induction on n.

For $n = 1$, there exist some $\lambda, \rho \in V^*$ and some production $A \to \gamma$, such that $\alpha = \lambda A \rho$ and $w = \lambda\gamma\rho$. Since w is a terminal string, λ, ρ, and γ, are terminal strings. Thus, A is the only nonterminal in α, and the derivation step $\alpha \stackrel{1}{\Longrightarrow} w$ is a leftmost step (and a rightmost step!).

If $n > 1$, then the derivation $\alpha \stackrel{n}{\Longrightarrow} w$ is of the form

$$\alpha \Longrightarrow \alpha_1 \stackrel{n-1}{\Longrightarrow} w.$$

There are two subcases.

Case 1. If the derivation step $\alpha \Longrightarrow \alpha_1$ is a leftmost step $\alpha \underset{lm}{\Longrightarrow} \alpha_1$, by the induction hypothesis, there is a leftmost derivation $\alpha_1 \underset{lm}{\stackrel{n-1}{\Longrightarrow}} w$, and we get the leftmost derivation

$$\alpha \underset{lm}{\Longrightarrow} \alpha_1 \underset{lm}{\stackrel{n-1}{\Longrightarrow}} w.$$

Case 2. The derivation step $\alpha \Longrightarrow \alpha_1$ is a not a leftmost step. In this case, there must be some $u \in \Sigma^*$, $\mu, \rho \in V^*$, some nonterminals A and B, and some production $B \to \delta$, such that

$$\alpha = uA\mu B\rho \quad \text{and} \quad \alpha_1 = uA\mu\delta\rho,$$

where A is the leftmost nonterminal in α. Since we have a derivation $\alpha_1 \stackrel{n-1}{\Longrightarrow} w$ of length $n - 1$, by the induction hypothesis, there is a leftmost derivation

$$\alpha_1 \underset{lm}{\stackrel{n-1}{\Longrightarrow}} w.$$

Since $\alpha_1 = uA\mu\delta\rho$ where A is the leftmost terminal in α_1, the first step in the leftmost derivation $\alpha_1 \underset{lm}{\stackrel{n-1}{\Longrightarrow}} w$ is of the form

$$uA\mu\delta\rho \underset{lm}{\Longrightarrow} u\gamma\mu\delta\rho,$$

for some production $A \to \gamma$. Thus, we have a derivation of the form

$$\alpha = uA\mu B\rho \Longrightarrow uA\mu\delta\rho \underset{lm}{\Longrightarrow} u\gamma\mu\delta\rho \overset{n-2}{\underset{lm}{\Longrightarrow}} w.$$

We can commute the first two steps involving the productions $B \to \delta$ and $A \to \gamma$, and we get the derivation

$$\alpha = uA\mu B\rho \underset{lm}{\Longrightarrow} u\gamma\mu B\rho \Longrightarrow u\gamma\mu\delta\rho \overset{n-2}{\underset{lm}{\Longrightarrow}} w.$$

This may no longer be a leftmost derivation, but the first step is leftmost, and we are back in case 1. Thus, we conclude by applying the induction hypothesis to the derivation $u\gamma\mu B\rho \overset{n-1}{\Longrightarrow} w$, as in case 1. $\square$

Proposition 6.1 implies that

$$L(G) = \{w \in \Sigma^* \mid S \overset{+}{\underset{lm}{\Longrightarrow}} w\} = \{w \in \Sigma^* \mid S \overset{+}{\underset{rm}{\Longrightarrow}} w\}.$$

We observed that if we consider the grammar $G_2 = (\{E, +, *, (,), a\}, \{+, *, (,), a\}, P, E)$, where P is the set of rules

$$\begin{aligned}
E &\longrightarrow E + E,\\
E &\longrightarrow E * E,\\
E &\longrightarrow (E),\\
E &\longrightarrow a,
\end{aligned}$$

the string $a + a * a$ has the following two distinct leftmost derivations, where the boldface indicates which occurrence of E is rewritten:

$$\begin{aligned}
&\mathbf{E} \Longrightarrow \mathbf{E} * E \Longrightarrow \mathbf{E} + E * E\\
&\Longrightarrow a + \mathbf{E} * E \Longrightarrow a + a * \mathbf{E} \Longrightarrow a + a * a,
\end{aligned}$$

and

$$\begin{aligned}
&\mathbf{E} \Longrightarrow \mathbf{E} + E \Longrightarrow a + \mathbf{E}\\
&\Longrightarrow a + \mathbf{E} * E \Longrightarrow a + a * \mathbf{E} \Longrightarrow a + a * a.
\end{aligned}$$

When this happens, we say that we have an ambiguous grammars. In some cases, it is possible to modify a grammar to make it unambiguous. For example, the grammar G_2 can be modified as follows.

Let $G_3 = (\{E, T, F, +, *, (,), a\}, \{+, *, (,), a\}, P, E)$, where P is the set of rules

$$\begin{aligned}
E &\longrightarrow E + T,\\
E &\longrightarrow T,\\
T &\longrightarrow T * F,\\
T &\longrightarrow F,\\
F &\longrightarrow (E),\\
F &\longrightarrow a.
\end{aligned}$$

We leave as an exercise to show that $L(G_3) = L(G_2)$, and that every string in $L(G_3)$ has a unique leftmost derivation. Unfortunately, it is not always possible to modify a context-free grammar to make it unambiguous. There exist context-free languages that have no unambiguous context-free grammars. For example, the language

$$L_3 = \{a^m b^m c^n \mid m, n \geq 1\} \cup \{a^m b^n c^n \mid m, n \geq 1\}$$

is context-free, since it is generated by the following context-free grammar:

$$\begin{aligned}
S &\to S_1, \\
S &\to S_2, \\
S_1 &\to XC, \\
S_2 &\to AY, \\
X &\to aXb, \\
X &\to ab, \\
Y &\to bYc, \\
Y &\to bc, \\
A &\to aA, \\
A &\to a, \\
C &\to cC, \\
C &\to c.
\end{aligned}$$

However, it can be shown that L_3 has no unambiguous grammars. All this motivates the following definition.

Definition 6.5. A context-free grammar $G = (V, \Sigma, P, S)$ is *ambiguous* if there is some string $w \in L(G)$ that has two distinct leftmost derivations (or two distinct rightmost derivations). Thus, a grammar G is *unambiguous* if every string $w \in L(G)$ has a unique leftmost derivation (or a unique rightmost derivation). A context-free language L is *inherently ambiguous* if every CFG G for L is ambiguous.

Whether or not a grammar is ambiguous affects the complexity of parsing. Parsing algorithms for unambiguous grammars are more efficient than parsing algorithms for ambiguous grammars.

We now consider various normal forms for context-free grammars.

6.3 Normal Forms for Context-Free Grammars, Chomsky Normal Form

One of the main goals of this section is to show that every CFG G can be converted to an equivalent grammar in *Chomsky Normal Form (for short, CNF)*. A context-free grammar

$G = (V, \Sigma, P, S)$ is in Chomsky Normal Form iff its productions are of the form

$$\begin{aligned}&A \to BC,\\ &A \to a, \quad \text{or}\\ &S \to \epsilon,\end{aligned}$$

where $A, B, C \in N$, $a \in \Sigma$, $S \to \epsilon$ is in P iff $\epsilon \in L(G)$, and S does not occur on the right-hand side of any production.

Note that a grammar in Chomsky Normal Form does not have ϵ-rules, i.e., rules of the form $A \to \epsilon$, except when $\epsilon \in L(G)$, in which case $S \to \epsilon$ is the only ϵ-rule. It also does not have *chain rules*, i.e., rules of the form $A \to B$, where $A, B \in N$. Thus, in order to convert a grammar to Chomsky Normal Form, we need to show how to eliminate ϵ-rules and chain rules. This is not the end of the story, since we may still have rules of the form $A \to \alpha$ where either $|\alpha| \geq 3$ or $|\alpha| \geq 2$ and α contains terminals. However, dealing with such rules is a simple recoding matter, and we first focus on the elimination of ϵ-rules and chain rules. It turns out that ϵ-rules must be eliminated first.

The first step to eliminate ϵ-rules is to compute the set $E(G)$ of *erasable (or nullable) nonterminals*

$$E(G) = \{A \in N \mid A \stackrel{+}{\Longrightarrow} \epsilon\}.$$

The set $E(G)$ is computed using a sequence of approximations E_i defined as follows:

$$\begin{aligned}E_0 &= \{A \in N \mid (A \to \epsilon) \in P\},\\ E_{i+1} &= E_i \cup \{A \mid \exists (A \to B_1 \ldots B_j \ldots B_k) \in P,\ B_j \in E_i,\ 1 \leq j \leq k\}.\end{aligned}$$

Clearly, the E_i form an ascending chain

$$E_0 \subseteq E_1 \subseteq \cdots \subseteq E_i \subseteq E_{i+1} \subseteq \cdots \subseteq N,$$

and since N is finite, there is a least i, say i_0, such that $E_{i_0} = E_{i_0+1}$. We claim that $E(G) = E_{i_0}$. Actually, we prove the following proposition.

Proposition 6.2. *Given a context-free grammar $G = (V, \Sigma, P, S)$, one can construct a context-free grammar $G' = (V', \Sigma, P', S')$ such that:*

(1) $L(G') = L(G)$;

(2) P' contains no ϵ-rules other than $S' \to \epsilon$, and $S' \to \epsilon \in P'$ iff $\epsilon \in L(G)$;

(3) S' does not occur on the right-hand side of any production in P'.

Proof. We begin by proving that $E(G) = E_{i_0}$. For this, we prove that $E(G) \subseteq E_{i_0}$ and $E_{i_0} \subseteq E(G)$.

To prove that $E_{i_0} \subseteq E(G)$, we proceed by induction on i. Since $E_0 = \{A \in N \mid (A \to \epsilon) \in P\}$, we have $A \stackrel{1}{\Longrightarrow} \epsilon$, and thus $A \in E(G)$. By the induction hypothesis, $E_i \subseteq$

$E(G)$. If $A \in E_{i+1}$, either $A \in E_i$ and then $A \in E(G)$, or there is some production $(A \to B_1 \ldots B_j \ldots B_k) \in P$, such that $B_j \in E_i$ for all j, $1 \leq j \leq k$. By the induction hypothesis, $B_j \stackrel{+}{\Longrightarrow} \epsilon$ for each j, $1 \leq j \leq k$, and thus

$$A \Longrightarrow B_1 \ldots B_j \ldots B_k \stackrel{+}{\Longrightarrow} B_2 \ldots B_j \ldots B_k \stackrel{+}{\Longrightarrow} B_j \ldots B_k \stackrel{+}{\Longrightarrow} \epsilon,$$

which shows that $A \in E(G)$.

To prove that $E(G) \subseteq E_{i_0}$, we also proceed by induction, but on the length of a derivation $A \stackrel{+}{\Longrightarrow} \epsilon$. If $A \stackrel{1}{\Longrightarrow} \epsilon$, then $A \to \epsilon \in P$, and thus $A \in E_0$ since $E_0 = \{A \in N \mid (A \to \epsilon) \in P\}$. If $A \stackrel{n+1}{\Longrightarrow} \epsilon$, then

$$A \Longrightarrow \alpha \stackrel{n}{\Longrightarrow} \epsilon,$$

for some production $A \to \alpha \in P$. If α contains terminals of nonterminals not in $E(G)$, it is impossible to derive ϵ from α, and thus, we must have $\alpha = B_1 \ldots B_j \ldots B_k$, with $B_j \in E(G)$, for all j, $1 \leq j \leq k$. However, $B_j \stackrel{n_j}{\Longrightarrow} \epsilon$ where $n_j \leq n$, and by the induction hypothesis, $B_j \in E_{i_0}$. But then, we get $A \in E_{i_0+1} = E_{i_0}$, as desired. $\square$

Having shown that $E(G) = E_{i_0}$, we construct the grammar G'. Its set of production P' is defined as follows. First, we create the production $S' \to S$ where $S' \notin V$, to make sure that S' does not occur on the right-hand side of any rule in P'. Let

$$P_1 = \{A \to \alpha \in P \mid \alpha \in V^+\} \cup \{S' \to S\},$$

and let P_2 be the set of productions

$$\begin{aligned} P_2 = \{A \to \alpha_1\alpha_2 \ldots \alpha_k\alpha_{k+1} \mid \exists \alpha_1 \in V^*, \ldots, \exists \alpha_{k+1} \in V^*, \ \exists B_1 \in E(G), \ldots, \exists B_k \in E(G) \\ A \to \alpha_1 B_1 \alpha_2 \ldots \alpha_k B_k \alpha_{k+1} \in P, \ k \geq 1, \ \alpha_1 \ldots \alpha_{k+1} \neq \epsilon\}. \end{aligned}$$

Note that $\epsilon \in L(G)$ iff $S \in E(G)$. If $S \notin E(G)$, then let $P' = P_1 \cup P_2$, and if $S \in E(G)$, then let $P' = P_1 \cup P_2 \cup \{S' \to \epsilon\}$. We claim that $L(G') = L(G)$, which is proved by showing that every derivation using G can be simulated by a derivation using G', and vice-versa. All the conditions of the proposition are now met. $\square$

From a practical point of view, the construction or Proposition 6.2 is very costly. For example, given a grammar containing the productions

$$\begin{aligned} S &\to ABCDEF, \\ A &\to \epsilon, \\ B &\to \epsilon, \\ C &\to \epsilon, \\ D &\to \epsilon, \\ E &\to \epsilon, \\ F &\to \epsilon, \\ \ldots &\to \ldots, \end{aligned}$$

eliminating ϵ-rules will create $2^6 - 1 = 63$ new rules corresponding to the 63 nonempty subsets of the set $\{A, B, C, D, E, F\}$. We now turn to the elimination of chain rules.

It turns out that matters are greatly simplified if we first apply Proposition 6.2 to the input grammar G, and we explain the construction assuming that $G = (V, \Sigma, P, S)$ satisfies the conditions of Proposition 6.2. For every nonterminal $A \in N$, we define the set

$$I_A = \{B \in N \mid A \stackrel{+}{\Longrightarrow} B\}.$$

The sets I_A are computed using approximations $I_{A,i}$ defined as follows:

$$I_{A,0} = \{B \in N \mid (A \rightarrow B) \in P\},$$
$$I_{A,i+1} = I_{A,i} \cup \{C \in N \mid \exists (B \rightarrow C) \in P, \text{ and } B \in I_{A,i}\}.$$

Clearly, for every $A \in N$, the $I_{A,i}$ form an ascending chain

$$I_{A,0} \subseteq I_{A,1} \subseteq \cdots \subseteq I_{A,i} \subseteq I_{A,i+1} \subseteq \cdots \subseteq N,$$

and since N is finite, there is a least i, say i_0, such that $I_{A,i_0} = I_{A,i_0+1}$. We claim that $I_A = I_{A,i_0}$. Actually, we prove the following proposition.

Proposition 6.3. *Given a context-free grammar $G = (V, \Sigma, P, S)$, one can construct a context-free grammar $G' = (V', \Sigma, P', S')$ such that:*

(1) $L(G') = L(G)$;

(2) Every rule in P' is of the form $A \rightarrow \alpha$ where $|\alpha| \geq 2$, or $A \rightarrow a$ where $a \in \Sigma$, or $S' \rightarrow \epsilon$ iff $\epsilon \in L(G)$;

(3) S' does not occur on the right-hand side of any production in P'.

Proof. First, we apply Proposition 6.2 to the grammar G, obtaining a grammar $G_1 = (V_1, \Sigma, S_1, P_1)$. The proof that $I_A = I_{A,i_0}$ is similar to the proof that $E(G) = E_{i_0}$. First, we prove that $I_{A,i} \subseteq I_A$ by induction on i. This is staightforward. Next, we prove that $I_A \subseteq I_{A,i_0}$ by induction on derivations of the form $A \stackrel{+}{\Longrightarrow} B$. In this part of the proof, we use the fact that G_1 has no ϵ-rules except perhaps $S_1 \rightarrow \epsilon$, and that S_1 does not occur on the right-hand side of any rule. This implies that a derivation $A \stackrel{n+1}{\Longrightarrow} C$ is necessarily of the form $A \stackrel{n}{\Longrightarrow} B \Longrightarrow C$ for some $B \in N$. Then, in the induction step, we have $B \in I_{A,i_0}$, and thus $C \in I_{A,i_0+1} = I_{A,i_0}$.

We now define the following sets of rules. Let

$$P_2 = P_1 - \{A \rightarrow B \mid A \rightarrow B \in P_1\},$$

and let

$$P_3 = \{A \rightarrow \alpha \mid B \rightarrow \alpha \in P_1,\ \alpha \notin N_1,\ B \in I_A\}.$$

We claim that $G' = (V_1, \Sigma, P_2 \cup P_3, S_1)$ satisfies the conditions of the proposition. For example, S_1 does not appear on the right-hand side of any production, since the productions in P_3 have right-hand sides from P_1, and S_1 does not appear on the right-hand side in P_1. It is also easily shown that $L(G') = L(G_1) = L(G)$. □

Let us apply the method of Proposition 6.3 to the grammar

$$G_3 = (\{E, T, F, +, *, (,), a\}, \{+, *, (,), a\}, P, E),$$

where P is the set of rules

$$\begin{aligned}
E &\longrightarrow E + T, \\
E &\longrightarrow T, \\
T &\longrightarrow T * F, \\
T &\longrightarrow F, \\
F &\longrightarrow (E), \\
F &\longrightarrow a.
\end{aligned}$$

We get $I_E = \{T, F\}$, $I_T = \{F\}$, and $I_F = \emptyset$. The new grammar G'_3 has the set of rules

$$\begin{aligned}
E &\longrightarrow E + T, \\
E &\longrightarrow T * F, \\
E &\longrightarrow (E), \\
E &\longrightarrow a, \\
T &\longrightarrow T * F, \\
T &\longrightarrow (E), \\
T &\longrightarrow a, \\
F &\longrightarrow (E), \\
F &\longrightarrow a.
\end{aligned}$$

At this stage, the grammar obtained in Proposition 6.3 no longer has ϵ-rules (except perhaps $S' \to \epsilon$ iff $\epsilon \in L(G)$) or chain rules. However, it may contain rules $A \to \alpha$ with $|\alpha| \geq 3$, or with $|\alpha| \geq 2$ and where α contains terminals(s). To obtain the Chomsky Normal Form. we need to eliminate such rules. This is not difficult, but notationally a bit messy.

Proposition 6.4. *Given a context-free grammar $G = (V, \Sigma, P, S)$, one can construct a context-free grammar $G' = (V', \Sigma, P', S')$ such that $L(G') = L(G)$ and G' is in Chomsky Normal Form, that is, a grammar whose productions are of the form*

$$\begin{aligned}
A &\to BC, \\
A &\to a, \quad \textit{or} \\
S' &\to \epsilon,
\end{aligned}$$

where $A, B, C \in N'$, $a \in \Sigma$, $S' \to \epsilon$ is in P' iff $\epsilon \in L(G)$, and S' does not occur on the right-hand side of any production in P'.

Proof. First, we apply Proposition 6.3, obtaining G_1. Let Σ_r be the set of terminals occurring on the right-hand side of rules $A \to \alpha \in P_1$, with $|\alpha| \geq 2$. For every $a \in \Sigma_r$, let X_a be a new nonterminal not in V_1. Let

$$P_2 = \{X_a \to a \mid a \in \Sigma_r\}.$$

Let $P_{1,r}$ be the set of productions

$$A \to \alpha_1 a_1 \alpha_2 \cdots \alpha_k a_k \alpha_{k+1},$$

where $a_1, \ldots, a_k \in \Sigma_r$ and $\alpha_i \in N_1^*$. For every production

$$A \to \alpha_1 a_1 \alpha_2 \cdots \alpha_k a_k \alpha_{k+1}$$

in $P_{1,r}$, let

$$A \to \alpha_1 X_{a_1} \alpha_2 \cdots \alpha_k X_{a_k} \alpha_{k+1}$$

be a new production, and let P_3 be the set of all such productions. Let $P_4 = (P_1 - P_{1,r}) \cup P_2 \cup P_3$. Now, productions $A \to \alpha$ in P_4 with $|\alpha| \geq 2$ do not contain terminals. However, we may still have productions $A \to \alpha \in P_4$ with $|\alpha| \geq 3$. We can perform some recoding using some new nonterminals. For every production of the form

$$A \to B_1 \cdots B_k,$$

where $k \geq 3$, create the new nonterminals

$$[B_1 \cdots B_{k-1}], [B_1 \cdots B_{k-2}], \cdots, [B_1 B_2 B_3], [B_1 B_2],$$

and the new productions

$$\begin{aligned}
A &\to [B_1 \cdots B_{k-1}] B_k, \\
[B_1 \cdots B_{k-1}] &\to [B_1 \cdots B_{k-2}] B_{k-1}, \\
\cdots &\to \cdots, \\
[B_1 B_2 B_3] &\to [B_1 B_2] B_3, \\
[B_1 B_2] &\to B_1 B_2.
\end{aligned}$$

All the productions are now in Chomsky Normal Form, and it is clear that the same language is generated. ☐

Applying the first phase of the method of Proposition 6.4 to the grammar G_3', we get the

rules

$$
\begin{aligned}
E &\longrightarrow EX_+T,\\
E &\longrightarrow TX_*F,\\
E &\longrightarrow X_(EX_),\\
E &\longrightarrow a,\\
T &\longrightarrow TX_*F,\\
T &\longrightarrow X_(EX_),\\
T &\longrightarrow a,\\
F &\longrightarrow X_(EX_),\\
F &\longrightarrow a,\\
X_+ &\longrightarrow +,\\
X_* &\longrightarrow *,\\
X_(&\longrightarrow (,\\
X_) &\longrightarrow).
\end{aligned}
$$

After applying the second phase of the method, we get the following grammar in Chomsky Normal Form:

$$
\begin{aligned}
E &\longrightarrow [EX_+]T,\\
[EX_+] &\longrightarrow EX_+,\\
E &\longrightarrow [TX_*]F,\\
[TX_*] &\longrightarrow TX_*,\\
E &\longrightarrow [X_(E]X_),\\
[X_(E] &\longrightarrow X_(E,\\
E &\longrightarrow a,\\
T &\longrightarrow [TX_*]F,\\
T &\longrightarrow [X_(E]X_),\\
T &\longrightarrow a,\\
F &\longrightarrow [X_(E]X_),\\
F &\longrightarrow a,\\
X_+ &\longrightarrow +,\\
X_* &\longrightarrow *,\\
X_(&\longrightarrow (,\\
X_) &\longrightarrow).
\end{aligned}
$$

For large grammars, it is often convenient to use the abbreviation which consists in grouping productions having a common left-hand side, and listing the right-hand sides separated

by the symbol $|$. Thus, a group of productions

$$
\begin{aligned}
A &\to \alpha_1, \\
A &\to \alpha_2, \\
\cdots &\to \cdots, \\
A &\to \alpha_k,
\end{aligned}
$$

may be abbreviated as

$$A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k.$$

An interesting corollary of the CNF is the following decidability result. There is an algorithm which, given a context-free grammar G, given any string $w \in \Sigma^*$, decides whether $w \in L(G)$. Indeed, we first convert G to a grammar G' in Chomsky Normal Form. If $w = \epsilon$, we can test whether $\epsilon \in L(G)$, since this is the case iff $S' \to \epsilon \in P'$. If $w \neq \epsilon$, letting $n = |w|$, note that since the rules are of the form $A \to BC$ or $A \to a$, where $a \in \Sigma$, any derivation for w has $n - 1 + n = 2n - 1$ steps. Thus, we enumerate all (leftmost) derivations of length $2n - 1$.

There are much better parsing algorithms than this naive algorithm. We now show that every regular language is context-free.

6.4 Regular Languages are Context-Free

The regular languages can be characterized in terms of very special kinds of context-free grammars, right-linear (and left-linear) context-free grammars.

Definition 6.6. A context-free grammar $G = (V, \Sigma, P, S)$ is *left-linear* iff its productions are of the form

$$
\begin{aligned}
A &\to Ba, \\
A &\to a, \\
A &\to \epsilon.
\end{aligned}
$$

where $A, B \in N$, and $a \in \Sigma$. A context-free grammar $G = (V, \Sigma, P, S)$ is *right-linear* iff its productions are of the form

$$
\begin{aligned}
A &\to aB, \\
A &\to a, \\
A &\to \epsilon.
\end{aligned}
$$

where $A, B \in N$, and $a \in \Sigma$.

The following proposition shows the equivalence between NFA's and right-linear grammars.

Proposition 6.5. *A language L is regular if and only if it is generated by some right-linear grammar.*

Proof. Let $L = L(D)$ for some DFA $D = (Q, \Sigma, \delta, q_0, F)$. We construct a right-linear grammar G as follows. Let $V = Q \cup \Sigma$, $S = q_0$, and let P be defined as follows:

$$P = \{p \to aq \mid q = \delta(p, a),\ p, q \in Q,\ a \in \Sigma\} \cup \{p \to \epsilon \mid p \in F\}.$$

It is easily shown by induction on the length of w that

$$p \stackrel{*}{\Longrightarrow} wq \quad \text{iff} \quad q = \delta^*(p, w),$$

and thus, $L(D) = L(G)$.

Conversely, let $G = (V, \Sigma, P, S)$ be a right-linear grammar. First, let $G = (V', \Sigma, P', S)$ be the right-linear grammar obtained from G by adding the new nonterminal E to N, replacing every rule in P of the form $A \to a$ where $a \in \Sigma$ by the rule $A \to aE$, and adding the rule $E \to \epsilon$. It is immediately verified that $L(G') = L(G)$. Next, we construct the NFA $M = (Q, \Sigma, \delta, q_0, F)$ as follows: $Q = N' = N \cup \{E\}$, $q_0 = S$, $F = \{A \in N' \mid A \to \epsilon\}$, and

$$\delta(A, a) = \{B \in N' \mid A \to aB \in P'\},$$

for all $A \in N$ and all $a \in \Sigma$. It is easily shown by induction on the length of w that

$$A \stackrel{*}{\Longrightarrow} wB \quad \text{iff} \quad B \in \delta^*(A, w),$$

and thus, $L(M) = L(G') = L(G)$. ☐

A similar proposition holds for left-linear grammars. It is also easily shown that the regular languages are exactly the languages generated by context-free grammars whose rules are of the form

$$A \to Bu,$$
$$A \to u,$$

where $A, B \in N$, and $u \in \Sigma^*$.

6.5 Useless Productions in Context-Free Grammars

Given a context-free grammar $G = (V, \Sigma, P, S)$, it may contain rules that are useless for a number of reasons. For example, consider the grammar $G_3 = (\{E, A, a, b\}, \{a, b\}, P, E)$, where P is the set of rules

$$E \longrightarrow aEb,$$
$$E \longrightarrow ab,$$
$$E \longrightarrow A,$$
$$A \longrightarrow bAa.$$

The problem is that the nonterminal A does not derive any terminal strings, and thus, it is useless, as well as the last two productions. Let us now consider the grammar $G_4 = (\{E, A, a, b, c, d\}, \{a, b, c, d\}, P, E)$, where P is the set of rules

$$
\begin{aligned}
E &\longrightarrow aEb, \\
E &\longrightarrow ab, \\
A &\longrightarrow cAd, \\
A &\longrightarrow cd.
\end{aligned}
$$

This time, the nonterminal A generates strings of the form $c^n d^n$, but there is no derivation $E \stackrel{+}{\Longrightarrow} \alpha$ from E where A occurs in α. The nonterminal A is not connected to E, and the last two rules are useless. Fortunately, it is possible to find such useless rules, and to eliminate them.

Let $T(G)$ be the set of nonterminals that actually derive some terminal string, i.e.

$$T(G) = \{A \in (V - \Sigma) \mid \exists w \in \Sigma^*,\ A \Longrightarrow^+ w\}.$$

The set $T(G)$ can be defined by stages. We define the sets T_n $(n \geq 1)$ as follows:

$$T_1 = \{A \in (V - \Sigma) \mid \exists (A \longrightarrow w) \in P,\ \text{with}\ w \in \Sigma^*\},$$

and

$$T_{n+1} = T_n \cup \{A \in (V - \Sigma) \mid \exists (A \longrightarrow \beta) \in P,\ \text{with}\ \beta \in (T_n \cup \Sigma)^*\}.$$

It is easy to prove that there is some least n such that $T_{n+1} = T_n$, and that for this n, $T(G) = T_n$.

If $S \notin T(G)$, then $L(G) = \emptyset$, and G is equivalent to the trivial grammar

$$G' = (\{S\}, \Sigma, \emptyset, S).$$

If $S \in T(G)$, then let $U(G)$ be the set of nonterminals that are actually useful, i.e.,

$$U(G) = \{A \in T(G) \mid \exists \alpha, \beta \in (T(G) \cup \Sigma)^*,\ S \Longrightarrow^* \alpha A \beta\}.$$

The set $U(G)$ can also be computed by stages. We define the sets U_n $(n \geq 1)$ as follows:

$$U_1 = \{A \in T(G) \mid \exists (S \longrightarrow \alpha A \beta) \in P,\ \text{with}\ \alpha, \beta \in (T(G) \cup \Sigma)^*\},$$

and

$$U_{n+1} = U_n \cup \{B \in T(G) \mid \exists (A \longrightarrow \alpha B \beta) \in P,\ \text{with}\ A \in U_n,\ \alpha, \beta \in (T(G) \cup \Sigma)^*\}.$$

It is easy to prove that there is some least n such that $U_{n+1} = U_n$, and that for this n, $U(G) = U_n \cup \{S\}$. Then, we can use $U(G)$ to transform G into an equivalent CFG in

which every nonterminal is useful (i.e., for which $V - \Sigma = U(G)$). Indeed, simply delete all rules containing symbols not in $U(G)$. The details are left as an exercise. We say that a context-free grammar G is *reduced* if all its nonterminals are useful, i.e., $N = U(G)$.

It should be noted than although dull, the above considerations are important in practice. Certain algorithms for constructing parsers, for example, LR-parsers, may loop if useless rules are not eliminated!

We now consider another normal form for context-free grammars, the Greibach Normal Form.

6.6 The Greibach Normal Form

Every CFG G can also be converted to an equivalent grammar in *Greibach Normal Form (for short, GNF)*. A context-free grammar $G = (V, \Sigma, P, S)$ is in Greibach Normal Form iff its productions are of the form

$$\begin{aligned} &A \to aBC, \\ &A \to aB, \\ &A \to a, \quad \text{or} \\ &S \to \epsilon, \end{aligned}$$

where $A, B, C \in N$, $a \in \Sigma$, $S \to \epsilon$ is in P iff $\epsilon \in L(G)$, and S does not occur on the right-hand side of any production.

Note that a grammar in Greibach Normal Form does not have ϵ-rules other than possibly $S \to \epsilon$. More importantly, except for the special rule $S \to \epsilon$, every rule produces some terminal symbol.

An important consequence of the Greibach Normal Form is that every nonterminal is not left recursive. A nonterminal A is *left recursive* iff $A \stackrel{+}{\Longrightarrow} A\alpha$ for some $\alpha \in V^*$. Left recursive nonterminals cause top-down determinitic parsers to loop. The Greibach Normal Form provides a way of avoiding this problem.

There are no easy proofs that every CFG can be converted to a Greibach Normal Form. A particularly elegant method due to Rosenkrantz using least fixed-points and matrices will be given in section 6.9.

Proposition 6.6. *Given a context-free grammar $G = (V, \Sigma, P, S)$, one can construct a context-free grammar $G' = (V', \Sigma, P', S')$ such that $L(G') = L(G)$ and G' is in Greibach Normal Form, that is, a grammar whose productions are of the form*

$$\begin{aligned} &A \to aBC, \\ &A \to aB, \\ &A \to a, \quad \textit{or} \\ &S' \to \epsilon, \end{aligned}$$

where $A, B, C \in N'$, $a \in \Sigma$, $S' \to \epsilon$ *is in* P' *iff* $\epsilon \in L(G)$, *and* S' *does not occur on the right-hand side of any production in* P'.

6.7 Least Fixed-Points

Context-free languages can also be characterized as least fixed-points of certain functions induced by grammars. This characterization yields a rather quick proof that every context-free grammar can be converted to Greibach Normal Form. This characterization also reveals very clearly the recursive nature of the context-free languages.

We begin by reviewing what we need from the theory of partially ordered sets.

Definition 6.7. Given a partially ordered set $\langle A, \leq\rangle$, an *ω-chain* $(a_n)_{n\geq 0}$ is a sequence such that $a_n \leq a_{n+1}$ for all $n \geq 0$. The *least-upper bound* of an ω-chain (a_n) is an element $a \in A$ such that:

(1) $a_n \leq a$, for all $n \geq 0$;

(2) For any $b \in A$, if $a_n \leq b$, for all $n \geq 0$, then $a \leq b$.

A partially ordered set $\langle A, \leq\rangle$ is an *ω-chain complete poset* iff it has a least element $\perp$, and iff every ω-chain has a least upper bound denoted as $\bigsqcup a_n$.

Remark: The ω in ω-chain means that we are considering countable chains (ω is the ordinal associated with the order-type of the set of natural numbers). This notation may seem arcane, but is standard in denotational semantics.

For example, given any set X, the power set 2^X ordered by inclusion is an ω-chain complete poset with least element $\emptyset$. The Cartesian product $\underbrace{2^X \times \cdots \times 2^X}_{n}$ ordered such that

$$(A_1, \ldots, A_n) \leq (B_1, \ldots, B_n)$$

iff $A_i \subseteq B_i$ (where $A_i, B_i \in 2^X$) is an ω-chain complete poset with least element $(\emptyset, \ldots, \emptyset)$.

We are interested in functions between partially ordered sets.

Definition 6.8. Given any two partially ordered sets $\langle A_1, \leq_1\rangle$ and $\langle A_2, \leq_2\rangle$, a function $f\colon A_1 \to A_2$ is *monotonic* iff for all $x, y \in A_1$,

$$x \leq_1 y \quad \text{implies that} \quad f(x) \leq_2 f(y).$$

If $\langle A_1, \leq_1\rangle$ and $\langle A_2, \leq_2\rangle$ are ω-chain complete posets, a function $f\colon A_1 \to A_2$ is *ω-continuous* iff it is monotonic, and for every ω-chain (a_n),

$$f(\bigsqcup a_n) = \bigsqcup f(a_n).$$

Remark: Note that we are not requiring that an ω-continuous function $f\colon A_1 \to A_2$ preserve least elements, i.e., it is possible that $f(\perp_1) \neq \perp_2$.

We now define the crucial concept of a least fixed-point.

Definition 6.9. Let $\langle A, \leq \rangle$ be a partially ordered set, and let $f\colon A \to A$ be a function. A *fixed-point of* f is an element $a \in A$ such that $f(a) = a$. The *least fixed-point of* f is an element $a \in A$ such that $f(a) = a$, and for every $b \in A$ such that $f(b) = b$, then $a \leq b$.

The following proposition gives sufficient conditions for the existence of least fixed-points. It is one of the key propositions in denotational semantics.

Proposition 6.7. *Let $\langle A, \leq \rangle$ be an ω-chain complete poset with least element $\perp$. Every ω-continuous function $f\colon A \to A$ has a unique least fixed-point x_0 given by*

$$x_0 = \bigsqcup f^n(\perp).$$

Furthermore, for any $b \in A$ such that $f(b) \leq b$, then $x_0 \leq b$.

Proof. First, we prove that the sequence

$$\perp, f(\perp), f^2(\perp), \ldots, f^n(\perp), \ldots$$

is an ω-chain. This is shown by induction on n. Since $\perp$ is the least element of A, we have $\perp \leq f(\perp)$. Assuming by induction that $f^n(\perp) \leq f^{n+1}(\perp)$, since f is ω-continuous, it is monotonic, and thus we get $f^{n+1}(\perp) \leq f^{n+2}(\perp)$, as desired.

Since A is an ω-chain complete poset, the ω-chain $(f^n(\perp))$ has a least upper bound

$$x_0 = \bigsqcup f^n(\perp).$$

Since f is ω-continuous, we have

$$f(x_0) = f(\bigsqcup f^n(\perp)) = \bigsqcup f(f^n(\perp)) = \bigsqcup f^{n+1}(\perp) = x_0,$$

and x_0 is indeed a fixed-point of f.

Clearly, if $f(b) \leq b$ implies that $x_0 \leq b$, then $f(b) = b$ implies that $x_0 \leq b$. Thus, assume that $f(b) \leq b$ for some $b \in A$. We prove by induction of n that $f^n(\perp) \leq b$. Indeed, $\perp \leq b$, since $\perp$ is the least element of A. Assuming by induction that $f^n(\perp) \leq b$, by monotonicity of f, we get

$$f(f^n(\perp)) \leq f(b),$$

and since $f(b) \leq b$, this yields

$$f^{n+1}(\perp) \leq b.$$

Since $f^n(\perp) \leq b$ for all $n \geq 0$, we have

$$x_0 = \bigsqcup f^n(\perp) \leq b.$$

□

The second part of Proposition 6.7 is very useful to prove that functions have the same least fixed-point. For example, under the conditions of Proposition 6.7, if $g\colon A \to A$ is another ω-chain continuous function, letting x_0 be the least fixed-point of f and y_0 be the least fixed-point of g, if $f(y_0) \leq y_0$ and $g(x_0) \leq x_0$, we can deduce that $x_0 = y_0$. Indeed, since $f(y_0) \leq y_0$ and x_0 is the least fixed-point of f, we get $x_0 \leq y_0$, and since $g(x_0) \leq x_0$ and y_0 is the least fixed-point of g, we get $y_0 \leq x_0$, and therefore $x_0 = y_0$.

Proposition 6.7 also shows that the least fixed-point x_0 of f can be approximated as much as desired, using the sequence $(f^n(\perp))$. We will now apply this fact to context-free grammars. For this, we need to show how a context-free grammar $G = (V, \Sigma, P, S)$ with m nonterminals induces an ω-continuous map

$$\Phi_G\colon \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m} \to \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m}.$$

6.8 Context-Free Languages as Least Fixed-Points

Given a context-free grammar $G = (V, \Sigma, P, S)$ with m nonterminals $A_1, \ldots A_m$, grouping all the productions having the same left-hand side, the grammar G can be concisely written as

$$\begin{aligned}
A_1 &\to \alpha_{1,1} + \cdots + \alpha_{1,n_1},\\
\cdots &\to \cdots\\
A_i &\to \alpha_{i,1} + \cdots + \alpha_{i,n_i},\\
\cdots &\to \cdots\\
A_m &\to \alpha_{m,1} + \cdots + \alpha_{m,n_n}.
\end{aligned}$$

Given any set A, let $\mathcal{P}_{fin}(A)$ be the set of finite subsets of A.

Definition 6.10. Let $G = (V, \Sigma, P, S)$ be a context-free grammar with m nonterminals A_1, ..., A_m. For any m-tuple $\Lambda = (L_1, \ldots, L_m)$ of languages $L_i \subseteq \Sigma^*$, we define the function

$$\Phi[\Lambda]\colon \mathcal{P}_{fin}(V^*) \to 2^{\Sigma^*}$$

inductively as follows:

$$\begin{aligned}
\Phi[\Lambda](\emptyset) &= \emptyset,\\
\Phi[\Lambda](\{\epsilon\}) &= \{\epsilon\},\\
\Phi[\Lambda](\{a\}) &= \{a\}, \quad \text{if } a \in \Sigma,\\
\Phi[\Lambda](\{A_i\}) &= L_i, \quad \text{if } A_i \in N,\\
\Phi[\Lambda](\{\alpha X\}) &= \Phi[\Lambda](\{\alpha\})\Phi[\Lambda](\{X\}), \quad \text{if } \alpha \in V^+,\ X \in V,\\
\Phi[\Lambda](Q \cup \{\alpha\}) &= \Phi[\Lambda](Q) \cup \Phi[\Lambda](\{\alpha\}), \quad \text{if } Q \in \mathcal{P}_{fin}(V^*),\ Q \neq \emptyset,\ \alpha \in V^*,\ \alpha \notin Q.
\end{aligned}$$

Then, writing the grammar G as

$$\begin{aligned} A_1 &\to \alpha_{1,1} + \cdots + \alpha_{1,n_1}, \\ \cdots &\to \cdots \\ A_i &\to \alpha_{i,1} + \cdots + \alpha_{i,n_i}, \\ \cdots &\to \cdots \\ A_m &\to \alpha_{m,1} + \cdots + \alpha_{m,n_n}, \end{aligned}$$

we define the map

$$\Phi_G \colon \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m} \to \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m}$$

such that

$$\Phi_G(L_1, \ldots L_m) = (\Phi[\Lambda](\{\alpha_{1,1}, \ldots, \alpha_{1,n_1}\}), \ldots, \Phi[\Lambda](\{\alpha_{m,1}, \ldots, \alpha_{m,n_m}\}))$$

for all $\Lambda = (L_1, \ldots, L_m) \in \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m}$.

One should verify that the map $\Phi[\Lambda]$ is well defined, but this is easy. The following proposition is easily shown:

Proposition 6.8. *Given a context-free grammar $G = (V, \Sigma, P, S)$ with m nonterminals A_1, …, A_m, the map*

$$\Phi_G \colon \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m} \to \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m}$$

is ω-continuous.

Now, $\underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m}$ is an ω-chain complete poset, and the map Φ_G is ω-continous. Thus, by Proposition 6.7, the map Φ_G has a least-fixed point. It turns out that the components of this least fixed-point are precisely the languages generated by the grammars (V, Σ, P, A_i). Before proving this fact, let us give an example illustrating it.

Example. Consider the grammar $G = (\{A, B, a, b\}, \{a, b\}, P, A)$ defined by the rules

$$\begin{aligned} A &\to BB + ab, \\ B &\to aBb + ab. \end{aligned}$$

The least fixed-point of Φ_G is the least upper bound of the chain

$$(\Phi_G^n(\emptyset, \emptyset)) = ((\Phi_{G,A}^n(\emptyset, \emptyset), \Phi_{G,B}^n(\emptyset, \emptyset)),$$

where

$$\Phi_{G,A}^0(\emptyset, \emptyset) = \Phi_{G,B}^0(\emptyset, \emptyset) = \emptyset,$$

and

$$\Phi^{n+1}_{G,A}(\emptyset,\emptyset) = \Phi^n_{G,B}(\emptyset,\emptyset)\Phi^n_{G,B}(\emptyset,\emptyset) \cup \{ab\},$$
$$\Phi^{n+1}_{G,B}(\emptyset,\emptyset) = a\Phi^n_{G,B}(\emptyset,\emptyset)b \cup \{ab\}.$$

It is easy to verify that

$$\Phi^1_{G,A}(\emptyset,\emptyset) = \{ab\},$$
$$\Phi^1_{G,B}(\emptyset,\emptyset) = \{ab\},$$
$$\Phi^2_{G,A}(\emptyset,\emptyset) = \{ab, abab\},$$
$$\Phi^2_{G,B}(\emptyset,\emptyset) = \{ab, aabb\},$$
$$\Phi^3_{G,A}(\emptyset,\emptyset) = \{ab, abab, abaabb, aabbab, aabbaabb\},$$
$$\Phi^3_{G,B}(\emptyset,\emptyset) = \{ab, aabb, aaabbb\}.$$

By induction, we can easily prove that the two components of the least fixed-point are the languages

$$L_A = \{a^m b^m a^n b^n \mid m, n \geq 1\} \cup \{ab\} \quad \text{and} \quad L_B = \{a^n b^n \mid n \geq 1\}.$$

Letting $G_A = (\{A, B, a, b\}, \{a, b\}, P, A)$ and $G_B = (\{A, B, a, b\}, \{a, b\}, P, B)$, it is indeed true that $L_A = L(G_A)$ and $L_B = L(G_B)$.

We have the following theorem due to Ginsburg and Rice:

Theorem 6.9. *Given a context-free grammar $G = (V, \Sigma, P, S)$ with m nonterminals A_1, ..., A_m, the least fixed-point of the map Φ_G is the m-tuple of languages*

$$(L(G_{A_1}), \ldots, L(G_{A_m})),$$

where $G_{A_i} = (V, \Sigma, P, A_i)$.

Proof. Writing G as

$$A_1 \to \alpha_{1,1} + \cdots + \alpha_{1,n_1},$$
$$\cdots \to \cdots$$
$$A_i \to \alpha_{i,1} + \cdots + \alpha_{i,n_i},$$
$$\cdots \to \cdots$$
$$A_m \to \alpha_{m,1} + \cdots + \alpha_{m,n_n},$$

let $M = \max\{|\alpha_{i,j}|\}$ be the maximum length of right-hand sides of rules in P. Let

$$\Phi^n_G(\emptyset, \ldots, \emptyset) = (\Phi^n_{G,1}(\emptyset, \ldots, \emptyset), \ldots, \Phi^n_{G,m}(\emptyset, \ldots, \emptyset)).$$

Then, for any $w \in \Sigma^*$, observe that

$$w \in \Phi^1_{G,i}(\emptyset, \ldots, \emptyset)$$

iff there is some rule $A_i \to \alpha_{i,j}$ with $w = \alpha_{i,j}$, and that

$$w \in \Phi^n_{G,i}(\emptyset, \ldots, \emptyset)$$

for some $n \geq 2$ iff there is some rule $A_i \to \alpha_{i,j}$ with $\alpha_{i,j}$ of the form

$$\alpha_{i,j} = u_1 A_{j_1} u_2 \cdots u_k A_{j_k} u_{k+1},$$

where $u_1, \ldots, u_{k+1} \in \Sigma^*$, $k \geq 1$, and some $w_1, \ldots, w_k \in \Sigma^*$ such that

$$w_h \in \Phi^{n-1}_{G,j_h}(\emptyset, \ldots, \emptyset),$$

and

$$w = u_1 w_1 u_2 \cdots u_k w_k u_{k+1}.$$

We prove the following two claims.

Claim 1: For every $w \in \Sigma^*$, if $A_i \stackrel{n}{\Longrightarrow} w$, then $w \in \Phi^p_{G,i}(\emptyset, \ldots, \emptyset)$, for some $p \geq 1$.

Claim 2: For every $w \in \Sigma^*$, if $w \in \Phi^n_{G,i}(\emptyset, \ldots, \emptyset)$, with $n \geq 1$, then $A_i \stackrel{p}{\Longrightarrow} w$ for some $p \leq (M+1)^{n-1}$.

Proof of Claim 1. We proceed by induction on n. If $A_i \stackrel{1}{\Longrightarrow} w$, then $w = \alpha_{i,j}$ for some rule $A \to \alpha_{i,j}$, and by the remark just before the claim, $w \in \Phi^1_{G,i}(\emptyset, \ldots, \emptyset)$.

If $A_i \stackrel{n+1}{\Longrightarrow} w$ with $n \geq 1$, then

$$A_i \stackrel{n}{\Longrightarrow} \alpha_{i,j} \Longrightarrow w$$

for some rule $A_i \to \alpha_{i,j}$. If

$$\alpha_{i,j} = u_1 A_{j_1} u_2 \cdots u_k A_{j_k} u_{k+1},$$

where $u_1, \ldots, u_{k+1} \in \Sigma^*$, $k \geq 1$, then $A_{j_h} \stackrel{n_h}{\Longrightarrow} w_h$, where $n_h \leq n$, and

$$w = u_1 w_1 u_2 \cdots u_k w_k u_{k+1}$$

for some $w_1, \ldots, w_k \in \Sigma^*$. By the induction hypothesis,

$$w_h \in \Phi^{p_h}_{G,j_h}(\emptyset, \ldots, \emptyset),$$

for some $p_h \geq 1$, for every h, $1 \leq h \leq k$. Letting $p = \max\{p_1, \ldots, p_k\}$, since each sequence $(\Phi^q_{G,i}(\emptyset, \ldots, \emptyset))$ is an ω-chain, we have $w_h \in \Phi^p_{G,j_h}(\emptyset, \ldots, \emptyset)$ for every h, $1 \leq h \leq k$, and by the remark just before the claim, $w \in \Phi^{p+1}_{G,i}(\emptyset, \ldots, \emptyset)$. $\square$

Proof of Claim 2. We proceed by induction on n. If $w \in \Phi^1_{G,i}(\emptyset, \ldots, \emptyset)$, by the remark just before the claim, then $w = \alpha_{i,j}$ for some rule $A \to \alpha_{i,j}$, and $A_i \stackrel{1}{\Longrightarrow} w$.

If $w \in \Phi^n_{G,i}(\emptyset, \ldots, \emptyset)$ for some $n \geq 2$, then there is some rule $A_i \to \alpha_{i,j}$ with $\alpha_{i,j}$ of the form

$$\alpha_{i,j} = u_1 A_{j_1} u_2 \cdots u_k A_{j_k} u_{k+1},$$

where $u_1, \ldots, u_{k+1} \in \Sigma^*$, $k \geq 1$, and some $w_1, \ldots, w_k \in \Sigma^*$ such that

$$w_h \in \Phi^{n-1}_{G,j_h}(\emptyset, \ldots, \emptyset),$$

and

$$w = u_1 w_1 u_2 \cdots u_k w_k u_{k+1}.$$

By the induction hypothesis, $A_{j_h} \stackrel{p_h}{\Longrightarrow} w_h$ with $p_h \leq (M+1)^{n-2}$, and thus

$$A_i \Longrightarrow u_1 A_{j_1} u_2 \cdots u_k A_{j_k} u_{k+1} \stackrel{p_1}{\Longrightarrow} \cdots \stackrel{p_k}{\Longrightarrow} w,$$

so that $A_i \stackrel{p}{\Longrightarrow} w$ with

$$p \leq p_1 + \cdots + p_k + 1 \leq M(M+1)^{n-2} + 1 \leq (M+1)^{n-1},$$

since $k \leq M$. □

Combining Claim 1 and Claim 2, we have

$$L(G_{A_i}) = \bigcup_n \Phi^n_{G,i}(\emptyset, \ldots, \emptyset),$$

which proves that the least fixed-point of the map Φ_G is the m-tuple of languages

$$(L(G_{A_1}), \ldots, L(G_{A_m})).$$

□

We now show how theorem 6.9 can be used to give a short proof that every context-free grammar can be converted to Greibach Normal Form.

6.9 Least Fixed-Points and the Greibach Normal Form

The hard part in converting a grammar $G = (V, \Sigma, P, S)$ to Greibach Normal Form is to convert it to a grammar in so-called *weak Greibach Normal Form*, where the productions are of the form

$$\begin{aligned} &A \to a\alpha, \quad \text{or} \\ &S \to \epsilon, \end{aligned}$$

where $a \in \Sigma$, $\alpha \in V^*$, and if $S \to \epsilon$ is a rule, then S does not occur on the right-hand side of any rule. Indeed, if we first convert G to Chomsky Normal Form, it turns out that we will get rules of the form $A \to aBC$, $A \to aB$ or $A \to a$.

Using the algorithm for eliminating ϵ-rules and chain rules, we can first convert the original grammar to a grammar with no chain rules and no ϵ-rules except possibly $S \to \epsilon$, in which case, S does not appear on the right-hand side of rules. Thus, for the purpose of converting to weak Greibach Normal Form, we can assume that we are dealing with grammars without chain rules and without ϵ-rules. Let us also assume that we computed the set $T(G)$ of nonterminals that actually derive some terminal string, and that useless productions involving symbols not in $T(G)$ have been deleted.

Let us explain the idea of the conversion using the following grammar:

$$\begin{aligned} A &\to AaB + BB + b. \\ B &\to Bd + BAa + aA + c. \end{aligned}$$

The first step is to group the right-hand sides α into two categories: those whose leftmost symbol is a terminal ($\alpha \in \Sigma V^*$) and those whose leftmost symbol is a nonterminal ($\alpha \in NV^*$). It is also convenient to adopt a matrix notation, and we can write the above grammar as

$$(A, B) = (A, B) \begin{pmatrix} aB & \emptyset \\ B & \{d, Aa\} \end{pmatrix} + (b, \{aA, c\})$$

Thus, we are dealing with matrices (and row vectors) whose entries are finite subsets of V^*. For notational simplicity, braces around singleton sets are omitted. The finite subsets of V^* form a semiring, where addition is union, and multiplication is concatenation. Addition and multiplication of matrices are as usual, except that the semiring operations are used. We will also consider matrices whose entries are languages over Σ. Again, the languages over Σ form a semiring, where addition is union, and multiplication is concatenation. The identity element for addition is $\emptyset$, and the identity element for multiplication is $\{\epsilon\}$. As above, addition and multiplication of matrices are as usual, except that the semiring operations are used. For example, given any languages $A_{i,j}$ and $B_{i,j}$ over Σ, where $i, j \in \{1, 2\}$, we have

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} = \begin{pmatrix} A_{1,1}B_{1,1} \cup A_{1,2}B_{2,1} & A_{1,1}B_{1,2} \cup A_{1,2}B_{2,2} \\ A_{2,1}B_{1,1} \cup A_{2,2}B_{2,1} & A_{2,1}B_{1,2} \cup A_{2,2}B_{2,2} \end{pmatrix}$$

Letting $X = (A, B)$, $K = (b, \{aA, c\})$, and

$$H = \begin{pmatrix} aB & \emptyset \\ B & \{d, Aa\} \end{pmatrix}$$

the above grammar can be concisely written as

$$X = XH + K.$$

More generally, given any context-free grammar $G = (V, \Sigma, P, S)$ with m nonterminals $A_1, \ldots, A_m$, assuming that there are no chain rules, no ϵ-rules, and that every nonterminal belongs to $T(G)$, letting

$$X = (A_1, \ldots, A_m),$$

we can write G as

$$X = XH + K,$$

for some appropriate $m \times m$ matrix H in which every entry contains a set (possibly empty) of strings in V^+, and some row vector K in which every entry contains a set (possibly empty) of strings α each beginning with a terminal ($\alpha \in \Sigma V^*$).

Given an $m \times m$ square matrix $A = (A_{i,j})$ of languages over Σ, we can define the matrix A^* whose entry $A^*_{i,j}$ is given by

$$A^*_{i,j} = \bigcup_{n \geq 0} A^n_{i,j},$$

where $A^0 = Id_m$, the identity matrix, and A^n is the n-th power of A. Similarly, we define A^+ where

$$A^+_{i,j} = \bigcup_{n \geq 1} A^n_{i,j}.$$

Given a matrix A where the entries are finite subset of V^*, where $N = \{A_1, \ldots, A_m\}$, for any m-tuple $\Lambda = (L_1, \ldots, L_m)$ of languages over Σ, we let

$$\Phi[\Lambda](A) = (\Phi[\Lambda](A_{i,j})).$$

Given a system $X = XH + K$ where H is an $m \times m$ matrix and X, K are row matrices, if H and K do not contain any nonterminals, we claim that the least fixed-point of the grammar G associated with $X = XH + K$ is KH^*. This is easily seen by computing the approximations $X^n = \Phi^n_G(\emptyset, \ldots, \emptyset)$. Indeed, $X^0 = K$, and

$$X^n = KH^n + KH^{n-1} + \cdots + KH + K = K(H^n + H^{n-1} + \cdots + H + I_m).$$

Similarly, if Y is an $m \times m$ matrix of nonterminals, the least fixed-point of the grammar associated with $Y = HY + H$ is H^+ (provided that H does not contain any nonterminals).

Given any context-free grammar $G = (V, \Sigma, P, S)$ with m nonterminals $A_1, \ldots, A_m$, writing G as $X = XH + K$ as explained earlier, we can form another grammar GH by creating m^2 new nonterminals $Y_{i,j}$, where the rules of this new grammar are defined by the system of two matrix equations

$$X = KY + K,$$
$$Y = HY + H,$$

where $Y = (Y_{i,j})$.

The following proposition is the key to the Greibach Normal Form.

Proposition 6.10. *Given any context-free grammar $G = (V, \Sigma, P, S)$ with m nonterminals $A_1, \ldots, A_m$, writing G as*

$$X = XH + K$$

as explained earlier, if GH is the grammar defined by the system of two matrix equations

$$X = KY + K,$$
$$Y = HY + H,$$

as explained above, then the components in X of the least-fixed points of the maps Φ_G and Φ_{GH} are equal.

Proof. Let U be the least-fixed point of Φ_G, and let (V, W) be the least fixed-point of Φ_{GH}. We shall prove that $U = V$. For notational simplicity, let us denote $\Phi[U](H)$ as $H[U]$ and $\Phi[U](K)$ as $K[U]$.

Since U is the least fixed-point of $X = XH + K$, we have

$$U = UH[U] + K[U].$$

Since $H[U]$ and $K[U]$ do not contain any nonterminals, by a previous remark, $K[U]H^*[U]$ is the least-fixed point of $X = XH[U] + K[U]$, and thus,

$$K[U]H^*[U] \leq U.$$

On the other hand, by monotonicity,

$$K[U]H^*[U]H\Big[K[U]H^*[U]\Big] + K\Big[K[U]H^*[U]\Big] \leq K[U]H^*[U]H[U] + K[U] = K[U]H^*[U],$$

and since U is the least fixed-point of $X = XH + K$,

$$U \leq K[U]H^*[U].$$

Therefore, $U = K[U]H^*[U]$. We can prove in a similar manner that $W = H[V]^+$.

Let $Z = H[U]^+$. We have

$$K[U]Z + K[U] = K[U]H[U]^+ + K[U] = K[U]H[U]^* = U,$$

and

$$H[U]Z + H[U] = H[U]H[U]^+ + H[U] = H[U]^+ = Z,$$

and since (V, W) is the least fixed-point of $X = KY + K$ and $Y = HY + H$, we get $V \leq U$ and $W \leq H[U]^+$.

We also have

$$V = K[V]W + K[V] = K[V]H[V]^+ + K[V] = K[V]H[V]^*,$$

and

$$VH[V] + K[V] = K[V]H[V]^*H[V] + K[V] = K[V]H[V]^* = V,$$

and since U is the least fixed-point of $X = XH + K$, we get $U \leq V$. Therefore, $U = V$, as claimed. $\square$

Note that the above proposition actually applies to any grammar. Applying Proposition 6.10 to our example grammar, we get the following new grammar:

$$(A, B) = (b, \{aA, c\}) \begin{pmatrix} Y_1 & Y_2 \\ Y_3 & Y_4 \end{pmatrix} + (b, \{aA, c\}),$$

$$\begin{pmatrix} Y_1 & Y_2 \\ Y_3 & Y_4 \end{pmatrix} = \begin{pmatrix} aB & \emptyset \\ B & \{d, Aa\} \end{pmatrix} \begin{pmatrix} Y_1 & Y_2 \\ Y_3 & Y_4 \end{pmatrix} + \begin{pmatrix} aB & \emptyset \\ B & \{d, Aa\} \end{pmatrix}$$

There are still some nonterminals appearing as leftmost symbols, but using the equations defining A and B, we can replace A with

$$\{bY_1, aAY_3, cY_3, b\}$$

and B with

$$\{bY_2, aAY_4, cY_4, aA, c\},$$

obtaining a system in weak Greibach Normal Form. This amounts to converting the matrix

$$H = \begin{pmatrix} aB & \emptyset \\ B & \{d, Aa\} \end{pmatrix}$$

to the matrix

$$L = \begin{pmatrix} aB & \emptyset \\ \{bY_2, aAY_4, cY_4, aA, c\} & \{d, bY_1a, aAY_3a, cY_3a, ba\} \end{pmatrix}$$

The weak Greibach Normal Form corresponds to the new system

$$X = KY + K,$$
$$Y = LY + L.$$

This method works in general for any input grammar with no ϵ-rules, no chain rules, and such that every nonterminal belongs to $T(G)$. Under these conditions, the row vector K contains some nonempty entry, all strings in K are in ΣV^*, and all strings in H are in V^+. After obtaining the grammar GH defined by the system

$$X = KY + K,$$
$$Y = HY + H,$$

we use the system $X = KY + K$ to express every nonterminal A_i in terms of expressions containing strings $\alpha_{i,j}$ involving a terminal as the leftmost symbol ($\alpha_{i,j} \in \Sigma V^*$), and we replace all leftmost occurrences of nonterminals in H (occurrences A_i in strings of the form $A_i\beta$, where $\beta \in V^*$) using the above expressions. In this fashion, we obtain a matrix L, and it is immediately shown that the system

$$X = KY + K,$$
$$Y = LY + L,$$

generates the same tuple of languages. Furthermore, this last system corresponds to a weak Greibach Normal Form.

It we start with a grammar in Chomsky Normal Form (with no production $S \to \epsilon$) such that every nonterminal belongs to $T(G)$, we actually get a Greibach Normal Form (the entries in K are terminals, and the entries in H are nonterminals). Thus, we have justified Proposition 6.6. The method is also quite economical, since it introduces only m^2 new nonterminals. However, the resulting grammar may contain some useless nonterminals.

6.10 Tree Domains and Gorn Trees

Derivation trees play a very important role in parsing theory and in the proof of a strong version of the pumping lemma for the context-free languages known as Ogden's lemma. Thus, it is important to define derivation trees rigorously. We do so using Gorn trees.

Let $\mathbf{N}_+ = \{1, 2, 3, \ldots\}$.

Definition 6.11. A *tree domain* D is a nonempty subset of strings in $\mathbf{N}_+^*$ satisfying the conditions:

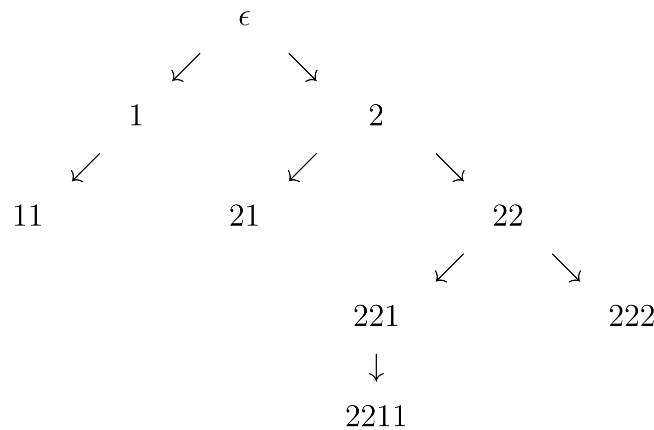
(1) For all $u, v \in \mathbf{N}_+^*$, if $uv \in D$, then $u \in D$.

(2) For all $u \in \mathbf{N}_+^*$, for every $i \in \mathbf{N}_+$, if $ui \in D$ then $uj \in D$ for every j, $1 \leq j \leq i$.

The tree domain

$$D = \{\epsilon, 1, 2, 11, 21, 22, 221, 222, 2211\}$$

is represented as follows:



A tree labeled with symbols from a set Δ is defined as follows.

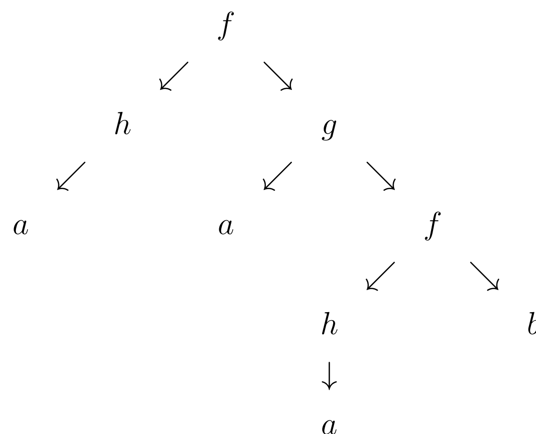
Definition 6.12. Given a set Δ of labels, a Δ-tree (for short, a *tree*) is a total function $t : D \to \Delta$, where D is a tree domain.

The domain of a tree t is denoted as $dom(t)$. Every string $u \in dom(t)$ is called a *tree address* or a *node*.

Let $\Delta = \{f, g, h, a, b\}$. The tree $t : D \to \Delta$, where D is the tree domain of the previous example and t is the function whose graph is

$$\{(\epsilon, f), (1, h), (2, g), (11, a), (21, a), (22, f), (221, h), (222, b), (2211, a)\}$$

is represented as follows:



The *outdegree* (sometimes called *ramification*) $r(u)$ of a node u is the cardinality of the set

$$\{i \mid ui \in dom(t)\}.$$

Note that the outdegree of a node can be infinite. Most of the trees that we shall consider will be *finite-branching*, that is, for every node u, $r(u)$ will be an integer, and hence finite. If the outdegree of all nodes in a tree is bounded by n, then we can view the domain of the tree as being defined over $\{1, 2, \ldots, n\}^*$.

A node of outdegree 0 is called a *leaf*. The node whose address is ϵ is called the *root* of the tree. A tree is *finite* if its domain $dom(t)$ is finite. Given a node u in $dom(t)$, every node of the form ui in $dom(t)$ with $i \in \mathbf{N}_+$ is called a *son* (or *immediate successor*) of u.

Tree addresses are totally ordered *lexicographically*: $u \leq v$ if either u is a prefix of v or, there exist strings $x, y, z \in \mathbf{N}_+^*$ and $i, j \in \mathbf{N}_+$, with $i < j$, such that $u = xiy$ and $v = xjz$.

In the first case, we say that u is an *ancestor* (or *predecessor*) of v (or u *dominates* v) and in the second case, that u is *to the left* of v.

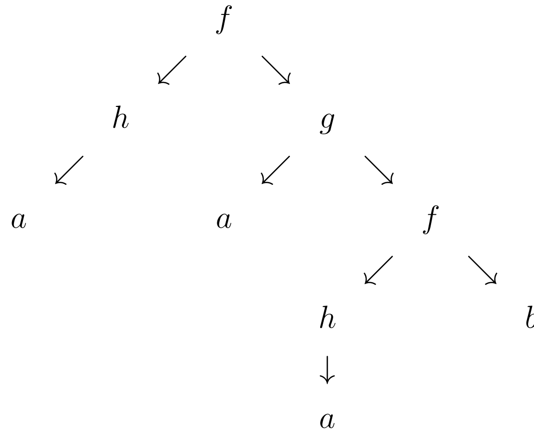
If $y = \epsilon$ and $z = \epsilon$, we say that xi is a *left brother* (or *left sibling*) of xj, $(i < j)$. Two tree addresses u and v are *independent* if u is not a prefix of v and v is not a prefix of u.

Given a finite tree t, the *yield* of t is the string

$$t(u_1)t(u_2)\cdots t(u_k),$$

where $u_1, u_2, \ldots, u_k$ is the sequence of leaves of t in lexicographic order.

For example, the yield of the tree below is $aaab$:



Given a finite tree t, the *depth* of t is the integer

$$d(t) = \max\{|u| \mid u \in dom(t)\}.$$

Given a tree t and a node u in $dom(t)$, the *subtree rooted at* u is the tree t/u, whose domain is the set

$$\{v \mid uv \in dom(t)\}$$

and such that $t/u(v) = t(uv)$ for all v in $dom(t/u)$.

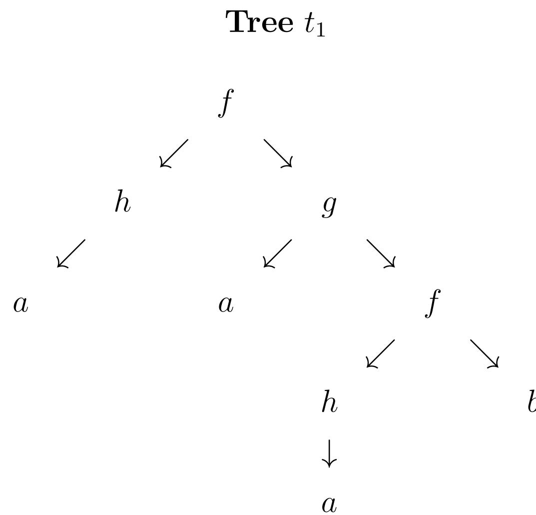
Another important operation is the operation of tree replacement (or tree substitution).

Definition 6.13. Given two trees t_1 and t_2 and a tree address u in t_1, the *result of substituting* t_2 at u in t_1, denoted by $t_1[u \leftarrow t_2]$, is the function whose graph is the set of pairs

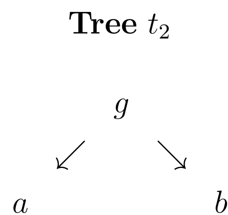
$$\{(v, t_1(v)) \mid v \in dom(t_1),\ u \text{ is not a prefix of } v\} \cup \{(uv, t_2(v)) \mid v \in dom(t_2)\}.$$

Let t_1 and t_2 be the trees defined by the following diagrams:

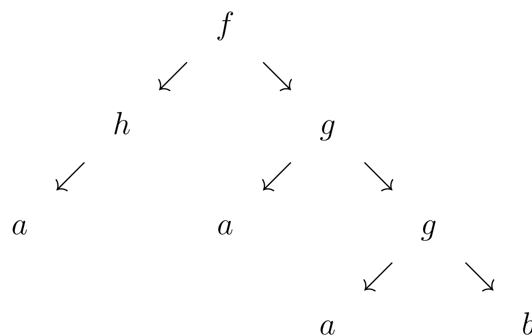
Tree t_1



Tree t_2



The tree $t_1[22 \leftarrow t_2]$ is defined by the following diagram:



We can now define derivation trees and relate derivations to derivation trees.

6.11 Derivations Trees

Definition 6.14. Given a context-free grammar $G = (V, \Sigma, P, S)$, for any $A \in N$, an *A-derivation tree for G* is a $(V \cup \{\epsilon\})$-tree t (a tree with set of labels $(V \cup \{\epsilon\})$) such that:

(1) $t(\epsilon) = A$;

(2) For every nonleaf node $u \in dom(t)$, if $u1, \ldots, uk$ are the successors of u, then either there is a production $B \to X_1 \cdots X_k$ in P such that $t(u) = B$ and $t(ui) = X_i$ for all i, $1 \leq i \leq k$, or $B \to \epsilon \in P$, $t(u) = B$ and $t(u1) = \epsilon$. A *complete derivation (or parse tree)* is an S-tree whose yield belongs to Σ^*.

A derivation tree for the grammar

$$G_3 = (\{E, T, F, +, *, (,), a\}, \{+, *, (,), a\}, P, E),$$

where P is the set of rules

$$\begin{aligned} E &\longrightarrow E + T, \\ E &\longrightarrow T, \\ T &\longrightarrow T * F, \\ T &\longrightarrow F, \\ F &\longrightarrow (E), \\ F &\longrightarrow a, \end{aligned}$$

is shown in Figure 6.1. The yield of the derivation tree is $a + a * a$.

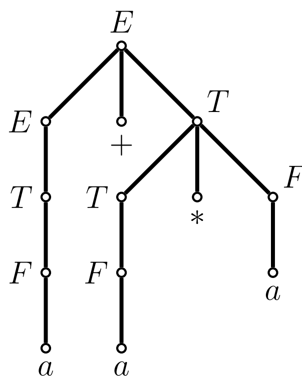


Figure 6.1: A complete derivation tree

Derivations trees are associated to derivations inductively as follows.

Definition 6.15. Given a context-free grammar $G = (V, \Sigma, P, S)$, for any $A \in N$, if $\pi : A \stackrel{n}{\Longrightarrow} \alpha$ is a derivation in G, we construct an *A-derivation tree* t_π *with yield* α as follows.

(1) If $n = 0$, then t_π is the one-node tree such that $dom(t_\pi) = \{\epsilon\}$ and $t_\pi(\epsilon) = A$.

(2) If $A \stackrel{n-1}{\Longrightarrow} \lambda B\rho \Longrightarrow \lambda\gamma\rho = \alpha$, then if t_1 is the A-derivation tree with yield $\lambda B\rho$ associated with the derivation $A \stackrel{n-1}{\Longrightarrow} \lambda B\rho$, and if t_2 is the tree associated with the production $B \rightarrow \gamma$ (that is, if

$$\gamma = X_1 \cdots X_k,$$

then $dom(t_2) = \{\epsilon, 1, \ldots, k\}$, $t_2(\epsilon) = B$, and $t_2(i) = X_i$ for all i, $1 \le i \le k$, or if $\gamma = \epsilon$, then $dom(t_2) = \{\epsilon, 1\}$, $t_2(\epsilon) = B$, and $t_2(1) = \epsilon$), then

$$t_\pi = t_1[u \leftarrow t_2],$$

where u is the address of the leaf labeled B in t_1.

The tree t_π is the *A-derivation tree associated with the derivation $A \stackrel{n}{\Longrightarrow} \alpha$*.

Given the grammar

$$G_2 = (\{E, +, *, (,), a\}, \{+, *, (,), a\}, P, E),$$

where P is the set of rules

$$\begin{aligned} E &\longrightarrow E + E, \\ E &\longrightarrow E * E, \\ E &\longrightarrow (E), \\ E &\longrightarrow a, \end{aligned}$$

the parse trees associated with two derivations of the string $a + a * a$ are shown in Figure 6.2:

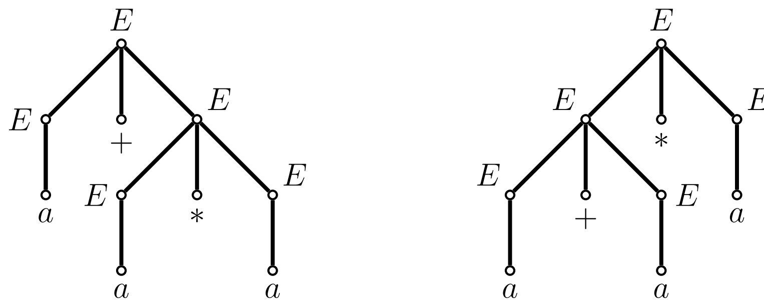


Figure 6.2: Two derivation trees for $a + a * a$

The following proposition is easily shown.

Proposition 6.11. *Let $G = (V, \Sigma, P, S)$ be a context-free grammar. For any derivation $A \stackrel{n}{\Longrightarrow} \alpha$, there is a unique A-derivation tree associated with this derivation, with yield α. Conversely, for any A-derivation tree t with yield α, there is a unique leftmost derivation $A \mathrel{\mathop{\Longrightarrow}\limits^{*}_{lm}} \alpha$ in G having t as its associated derivation tree.*

We will now prove a strong version of the pumping lemma for context-free languages due to Bill Ogden (1968).

6.12 Ogden's Lemma

Ogden's lemma states some combinatorial properties of parse trees that are deep enough. The yield w of such a parse tree can be split into 5 substrings u, v, x, y, z such that

$$w = uvxyz,$$

where u, v, x, y, z satisfy certain conditions. It turns out that we get a more powerful version of the lemma if we allow ourselves to *mark* certain occurrences of symbols in w before invoking the lemma. We can imagine that *marked occurrences* in a nonempty string w are occurrences of symbols in w in boldface, or red, or any given color (but one color only). For example, given $w = aaababbbaa$, we can mark the symbols of even index as follows:

$$a\mathbf{a}a\mathbf{b}a\mathbf{b}b\mathbf{b}a\mathbf{a}.$$

More rigorously, we can define a *marking* of a nonnull string $w\colon \{1, \ldots, n\} \to \Sigma$ as any function $m\colon \{1, \ldots, n\} \to \{0, 1\}$. Then, a letter w_i in w is a *marked occurrence* iff $m(i) = 1$, and an *unmarked occurrence* if $m(i) = 0$. The number of marked occurrences in w is equal to

$$\sum_{i=1}^{n} m(i).$$

Ogden's lemma only yields useful information for grammars G generating an infinite language. We could make this hypothesis, but it seems more elegant to use the precondition that the lemma only applies to strings $w \in L(D)$ such that w contains at least K marked occurrences, for a constant K large enough. If K is large enough, $L(G)$ will indeed be infinite.

Proposition 6.12. *For every context-free grammar G, there is some integer $K > 1$ such that, for every string $w \in \Sigma^+$, for every marking of w, if $w \in L(G)$ and w contains at least K marked occurrences, then there exists some decomposition of w as $w = uvxyz$, and some $A \in N$, such that the following properties hold:*

(1) There are derivations $S \stackrel{+}{\Longrightarrow} uAz$, $A \stackrel{+}{\Longrightarrow} vAy$, and $A \stackrel{+}{\Longrightarrow} x$, so that

$$uv^n xy^n z \in L(G)$$

for all $n \geq 0$ (the pumping property);

(2) x contains some marked occurrence;

(3) Either (both u and v contain some marked occurrence), or (both y and z contain some marked occurrence);

(4) vxy contains less than K marked occurrences.

Proof. Let t be any parse tree for w. We call a leaf of t a *marked leaf* if its label is a marked occurrence in the marked string w. The general idea is to make sure that K is large enough so that parse trees with yield w contain enough repeated nonterminals along some path from the root to some marked leaf. Let $r = |N|$, and let

$$p = \max\{2,\ \max\{|\alpha| \mid (A \to \alpha) \in P\}\}.$$

We claim that $K = p^{2r+3}$ does the job.

The key concept in the proof is the notion of a B-node. Given a parse tree t, a B*-node* is a node with at least two immediate successors u_1, u_2, such that for $i = 1, 2$, either u_i is a marked leaf, or u_i has some marked leaf as a descendant. We construct a path from the root to some marked leaf, so that for every B-node, we pick the leftmost successor with the maximum number of marked leaves as descendants. Formally, define a path $(s_0, \ldots, s_n)$ from the root to some marked leaf, so that:

(i) Every node s_i has some marked leaf as a descendant, and s_0 is the root of t;

(ii) If s_j is in the path, s_j is not a leaf, and s_j has a single immediate descendant which is either a marked leaf or has marked leaves as its descendants, let s_{j+1} be that unique immediate descendant of s_i.

(iii) If s_j is a B-node in the path, then let s_{j+1} be the leftmost immediate successors of s_j with the maximum number of marked leaves as descendants (assuming that if s_{j+1} is a marked leaf, then it is its own descendant).

(iv) If s_j is a leaf, then it is a marked leaf and $n = j$.

We will show that the path $(s_0, \ldots, s_n)$ contains at least $2r + 3$ B-nodes.

Claim: For every i, $0 \leq i \leq n$, if the path $(s_i, \ldots, s_n)$ contains b B-nodes, then s_i has at most p^b marked leaves as descendants.

Proof. We proceed by "backward induction", i.e., by induction on $n - i$. For $i = n$, there are no B-nodes, so that $b = 0$, and there is indeed $p^0 = 1$ marked leaf s_n. Assume that the claim holds for the path $(s_{i+1}, \ldots, s_n)$.

If s_i is not a B-node, then the number b of B-nodes in the path $(s_{i+1}, \ldots, s_n)$ is the same as the number of B-nodes in the path $(s_i, \ldots, s_n)$, and s_{i+1} is the only immediate successor of s_i having a marked leaf as descendant. By the induction hypothesis, s_{i+1} has at most p^b marked leaves as descendants, and this is also an upper bound on the number of marked leaves which are descendants of s_i.

If s_i is a B-node, then if there are b B-nodes in the path $(s_{i+1}, \ldots, s_n)$, there are $b + 1$ B-nodes in the path $(s_i, \ldots, s_n)$. By the induction hypothesis, s_{i+1} has at most p^b marked leaves as descendants. Since s_i is a B-node, s_{i+1} was chosen to be the leftmost immediate successor of s_i having the maximum number of marked leaves as descendants. Thus, since

the outdegree of s_i is at most p, and each of its immediate successors has at most p^b marked leaves as descendants, the node s_i has at most $pp^d = p^{d+1}$ marked leaves as descendants, as desired. $\square$

Applying the claim to s_0, since w has at least $K = p^{2r+3}$ marked occurrences, we have $p^b \geq p^{2r+3}$, and since $p \geq 2$, we have $b \geq 2r + 3$, and the path $(s_0, \ldots, s_n)$ contains at least $2r + 3$ B-nodes (Note that this would not follow if we had $p = 1$).

Let us now select the lowest $2r + 3$ B-nodes in the path, $(s_0, \ldots, s_n)$, and denote them $(b_1, \ldots, b_{2r+3})$. Every B-node b_i has at least two immediate successors $u_i < v_i$ such that u_i or v_i is on the path $(s_0, \ldots, s_n)$. If the path goes through u_i, we say that b_i is a *right B-node* and if the path goes through v_i, we say that b_i is a *left B-node*. Since $2r+3 = r+2+r+1$, either there are $r+2$ left B-nodes or there are $r+2$ right B-nodes in the path $(b_1, \ldots, b_{2r+3})$. Let us assume that there are $r + 2$ left B-nodes, the other case being similar.

Let $(d_1, \ldots, d_{r+2})$ be the lowest $r + 2$ left B-nodes in the path. Since there are $r + 1$ B-nodes in the sequence $(d_2, \ldots, d_{r+2})$, and there are only r distinct nonterminals, there are two nodes d_i and d_j, with $2 \leq i < j \leq r + 2$, such that $t(d_i) = t(d_j) = A$, for some $A \in N$. We can assume that d_i is an ancestor of d_j, and thus, $d_j = d_i\alpha$, for some $\alpha \neq \epsilon$.

If we prune out the subtree t/d_i rooted at d_i from t, we get an S-derivation tree having a yield of the form uAz, and we have a derivation of the form $S \stackrel{+}{\Longrightarrow} uAz$, since there are at least $r + 2$ left B-nodes on the path, and we are looking at the lowest $r + 1$ left B-nodes. Considering the subtree t/d_i, pruning out the subtree t/d_j rooted at α in t/d_i, we get an A-derivation tree having a yield of the form vAy, and we have a derivation of the form $A \stackrel{+}{\Longrightarrow} vAy$. Finally, the subtree t/d_j is an A-derivation tree with yield x, and we have a derivation $A \stackrel{+}{\Longrightarrow} x$. This proves (1) of the lemma.

Since s_n is a marked leaf and a descendant of d_j, x contains some marked occurrence, proving (2).

Since d_1 is a left B-node, some left sibbling of the immediate successor of d_1 on the path has some distinguished leaf in u as a descendant. Similarly, since d_i is a left B-node, some left sibbling of the immediate successor of d_i on the path has some distinguished leaf in v as a descendant. This proves (3).

$(d_j, \ldots, b_{2r+3})$ has at most $2r+1$ B-nodes, and by the claim shown earlier, d_j has at most p^{2r+1} marked leaves as descendants. Since $p^{2r+1} < p^{2r+3} = K$, this proves (4). $\square$

Observe that condition (2) implies that $x \neq \epsilon$, and condition (3) implies that either $u \neq \epsilon$ and $v \neq \epsilon$, or $y \neq \epsilon$ and $z \neq \epsilon$. Thus, the pumping condition (1) implies that the set $\{uv^nxy^nz \mid n \geq 0\}$ is an infinite subset of $L(G)$, and $L(G)$ is indeed infinite, as we mentioned earlier. Note that $K \geq 3$, and in fact, $K \geq 32$. The "standard pumping lemma" due to Bar-Hillel, Perles, and Shamir, is obtained by letting all occurrences be marked in $w \in L(G)$.

Proposition 6.13. *For every context-free grammar G (without ϵ-rules), there is some integer $K > 1$ such that, for every string $w \in \Sigma^+$, if $w \in L(G)$ and $|w| \geq K$, then there exists some decomposition of w as $w = uvxyz$, and some $A \in N$, such that the following properties hold:*

(1) There are derivations $S \stackrel{+}{\Longrightarrow} uAz$, $A \stackrel{+}{\Longrightarrow} vAy$, and $A \stackrel{+}{\Longrightarrow} x$, so that

$$uv^n xy^n z \in L(G)$$

for all $n \geq 0$ (the pumping property);

(2) $x \neq \epsilon$;

(3) Either $v \neq \epsilon$ or $y \neq \epsilon$;

(4) $|vxy| \leq K$.

A stronger version could be stated, and we are just following tradition in stating this standard version of the pumping lemma.

Ogden's lemma or the pumping lemma can be used to show that certain languages are not context-free. The method is to proceed by contradiction, i.e., to assume (contrary to what we wish to prove) that a language L is indeed context-free, and derive a contradiction of Ogden's lemma (or of the pumping lemma). Thus, as in the case of the regular languages, it would be helpful to see what the negation of Ogden's lemma is, and for this, we first state Ogden's lemma as a logical formula.

For any nonnull string $w\colon \{1, \ldots, n\} \to \Sigma$, for any marking $m\colon \{1, \ldots, n\} \to \{0, 1\}$ of w, for any substring y of w, where $w = xyz$, with $|x| = h$ and $k = |y|$, the number of marked occurrences in y, denoted as $|m(y)|$, is defined as

$$|m(y)| = \sum_{i=h+1}^{i=h+k} m(i).$$

We will also use the following abbreviations:

$$\begin{aligned}
nat &= \{0, 1, 2, \ldots\},\\
nat32 &= \{32, 33, \ldots\},\\
A &\equiv w = uvxyz,\\
B &\equiv |m(x)| \geq 1,\\
C &\equiv (|m(u)| \geq 1 \wedge |m(v)| \geq 1) \vee (|m(y)| \geq 1 \wedge |m(z)| \geq 1),\\
D &\equiv |m(vxy)| < K,\\
P &\equiv \forall n\colon nat\ (uv^n xy^n z \in L(D)).
\end{aligned}$$

Ogden's lemma can then be stated as

$$\forall G\colon \text{CFG}\ \exists K\colon nat32\ \forall w\colon \Sigma^*\ \forall m\colon \text{marking}$$
$$\Big((w \in L(D) \wedge |m(w)| \geq K) \supset (\exists u, v, x, y, z\colon \Sigma^*\ A \wedge B \wedge C \wedge D \wedge P)\Big).$$

Recalling that

$$\neg(A \wedge B \wedge C \wedge D \wedge P) \equiv \neg(A \wedge B \wedge C \wedge D) \vee \neg P \equiv (A \wedge B \wedge C \wedge D) \supset \neg P$$

and

$$\neg(P \supset Q) \equiv P \wedge \neg Q,$$

the negation of Ogden's lemma can be stated as

$$\exists G\colon \text{CFG}\ \forall K\colon nat32\ \exists w\colon \Sigma^*\ \exists m\colon \text{marking}$$
$$\Big((w \in L(D) \wedge |m(w)| \geq K) \wedge (\forall u, v, x, y, z\colon \Sigma^*\ (A \wedge B \wedge C \wedge D) \supset \neg P)\Big).$$

Since

$$\neg P \equiv \exists n\colon nat\ (uv^n xy^n z \notin L(D)),$$

in order to show that Ogden's lemma is contradicted, one needs to show that for some context-free grammar G, for every $K \geq 2$, there is some string $w \in L(D)$ and some marking m of w with at least K marked occurrences in w, such that for every possible decomposition $w = uvxyz$ satisfying the constraints $A \wedge B \wedge C \wedge D$, there is some $n \geq 0$ such that $uv^n xy^n z \notin L(D)$. When proceeding by contradiction, we have a language L that we are (wrongly) assuming to be context-free and we can use any CFG grammar G generating L. The creative part of the argument is to pick the right $w \in L$ and the right marking of w (not making any assumption on K).

As an illustration, we show that the language

$$L = \{a^n b^n c^n \mid n \geq 1\}$$

is not context-free. Since L is infinite, we will be able to use the pumping lemma.

The proof proceeds by contradiction. If L was context-free, there would be some context-free grammar G such that $L = L(G)$, and some constant $K > 1$ as in Ogden's lemma. Let $w = a^K b^K c^K$, and choose the $b's$ as marked occurrences. Then by Ogden's lemma, x contains some marked occurrence, and either both u, v or both y, z contain some marked occurrence. Assume that both u and v contain some b. We have the following situation:

$$\underbrace{a \cdots ab \cdots b}_{u}\underbrace{b \cdots b}_{v}\underbrace{b \cdots bc \cdots c}_{xyz}.$$

If we consider the string $uvvxyyz$, the number of a's is still K, but the number of b's is strictly greater than K since v contains at least one b, and thus $uvvxyyz \notin L$, a contradiction.

If both y and z contain some b we will also reach a contradiction because in the string $uvvxyyz$, the number of c's is still K, but the number of b's is strictly greater than K. Having reached a contradiction in all cases, we conclude that L is not context-free.

Let us now show that the language

$$L = \{a^m b^n c^m d^n \mid m, n \geq 1\}$$

is not context-free.

Again, we proceed by contradiction. This time, let

$$w = a^K b^K c^K d^K,$$

where the b's and c's are marked occurrences.

By Ogden's lemma, either both u, v contain some marked occurrence, or both y, z contain some marked occurrence, and x contains some marked occurrence. Let us first consider the case where both u, v contain some marked occurrence.

If v contains some b, since $uvvxyyz \in L$, v must contain only b's, since otherwise we would have a bad string in L, and we have the following situation:

$$\underbrace{a \cdots ab \cdots b}_{u}\underbrace{b \cdots b}_{v}\underbrace{b \cdots bc \cdots cd \cdots d}_{xyz}.$$

Since $uvvxyyz \in L$, the only way to preserve an equal number of b's and d's is to have $y \in d^+$. But then, vxy contains c^K, which contradicts (4) of Ogden's lemma.

If v contains some c, since x also contains some marked occurrence, it must be some c, and v contains only c's and we have the following situation:

$$\underbrace{a \cdots ab \cdots bc \cdots c}_{u}\underbrace{c \cdots c}_{v}\underbrace{c \cdots cd \cdots d}_{xyz}.$$

Since $uvvxyyz \in L$ and the number of a's is still K whereas the number of c's is strictly more than K, this case is impossible.

Let us now consider the case where both y, z contain some marked occurrence. Reasoning as before, the only possibility is that $v \in a^+$ and $y \in c^+$:

$$\underbrace{a \cdots a}_{u}\underbrace{a \cdots a}_{v}\underbrace{a \cdots ab \cdots bc \cdots c}_{x}\underbrace{c \cdots c}_{y}\underbrace{c \cdots cd \cdots d}_{z}.$$

But then, vxy contains b^K, which contradicts (4) of Ogden's Lemma. Since a contradiction was obtained in all cases, L is not context-free.

Ogden's lemma can also be used to show that the context-free language

$$\{a^m b^n c^n \mid m, n \geq 1\} \cup \{a^m b^m c^n \mid m, n \geq 1\}$$

is inherently ambiguous. The proof is quite involved.

Another corollary of the pumping lemma is that it is decidable whether a context-free grammar generates an infinite language.

Proposition 6.14. *Given any context-free grammar, G, if K is the constant of Ogden's lemma, then the following equivalence holds:*

$L(G)$ is infinite iff there is some $w \in L(G)$ such that $K \leq |w| < 2K$.

Proof. Let $K = p^{2r+3}$ be the constant from the proof of Proposition 6.12. If there is some $w \in L(G)$ such that $|w| \geq K$, we already observed that the pumping lemma implies that $L(G)$ contains an infinite subset of the form $\{uv^n xy^n z \mid n \geq 0\}$. Conversely, assume that $L(G)$ is infinite. If $|w| < K$ for all $w \in L(G)$, then $L(G)$ is finite. Thus, there is some $w \in L(G)$ such that $|w| \geq K$. Let $w \in L(G)$ be a minimal string such that $|w| \geq K$. By the pumping lemma, we can write w as $w = uvxyxz$, where $x \neq \epsilon$, $vy \neq \epsilon$, and $|vxy| \leq K$. By the pumping property, $uxz \in L(G)$. If $|w| \geq 2K$, then

$$|uxz| = |uvxyz| - |vy| > |uvxyz| - |vxy| \geq 2K - K = K,$$

and $|uxz| < |uvxyz|$, contradicting the minimality of w. Thus, we must have $|w| < 2K$. $\square$

In particular, if G is in Chomsky Normal Form, it can be shown that we just have to consider derivations of length at most $4K - 3$.

6.13 Pushdown Automata

We have seen that the regular languages are exactly the languages accepted by DFA's or NFA's. The context-free languages are exactly the languages accepted by pushdown automata, for short, PDA's. However, although there are two versions of PDA's, deterministic and nondeterministic, contrary to the fact that every NFA can be converted to a DFA, nondeterministic PDA's are strictly more poweful than deterministic PDA's (DPDA's). Indeed, there are context-free languages that cannot be accepted by DPDA's.

Thus, the natural machine model for the context-free languages is nondeterministic, and for this reason, we just use the abbreviation PDA, as opposed to NPDA. We adopt a definition of a PDA in which the pushdown store, or stack, must not be empty for a move to take place. Other authors allow PDA's to make move when the stack is empty. Novices seem to be confused by such moves, and this is why we do not allow moves with an empty stack.

Intuitively, a PDA consists of an input tape, a nondeterministic finite-state control, and a stack.

Given any set X possibly infinite, let $\mathcal{P}_{fin}(X)$ be the set of all finite subsets of X.

Definition 6.16. A *pushdown automaton* is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where

- Q is a finite set of *states*;
- Σ is a finite *input alphabet*;
- Γ is a finite *pushdown store (or stack) alphabet*;
- $q_0 \in Q$ is the *start state* (or *initial state*);
- $Z_0 \in \Gamma$ is the *initial stack symbol* (or *bottom marker*);
- $F \subseteq Q$ is the set of *final (or accepting) states*;
- $\delta\colon Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to \mathcal{P}_{fin}(Q \times \Gamma^*)$ is the *transition function*.

A transition is of the form $(q, \gamma) \in \delta(p, a, Z)$, where $p, q \in Q$, $Z \in \Gamma$, $\gamma \in \Gamma^*$ and $a \in \Sigma \cup \{\epsilon\}$. A transition of the form $(q, \gamma) \in \delta(p, \epsilon, Z)$ is called an *ϵ-transition (or ϵ-move)*.

The way a PDA operates is explained in terms of *Instantaneous Descriptions*, for short *ID's*. Intuitively, an Instantaneous Description is a snapshot of the PDA. An ID is a triple of the form

$$(p, u, \alpha) \in Q \times \Sigma^* \times \Gamma^*.$$

The idea is that p is the current state, u is the remaining input, and α represents the stack.

It is important to note that we use the convention that the **leftmost** symbol in α represents the topmost stack symbol.

Given a PDA M, we define a relation $\vdash_M$ between pairs of ID's. This is very similar to the derivation relation $\Longrightarrow_G$ associated with a context-free grammar.

Intuitively, a PDA scans the input tape symbol by symbol from left to right, making moves that cause a change of state, an update to the stack (but only at the top), and either advancing the reading head to the next symbol, or not moving the reading head during an ϵ-move.

Definition 6.17. Given a PDA

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F),$$

the relation $\vdash_M$ is defined as follows:

(1) For any move $(q, \gamma) \in \delta(p, a, Z)$, where $p, q \in Q$, $Z \in \Gamma$, $a \in \Sigma$, for every ID of the form $(p, av, Z\alpha)$, we have

$$(p, av, Z\alpha) \vdash_M (q, v, \gamma\alpha).$$

(2) For any move $(q, \gamma) \in \delta(p, \epsilon, Z)$, where $p, q \in Q$, $Z \in \Gamma$, for every ID of the form $(p, u, Z\alpha)$, we have

$$(p, u, Z\alpha) \vdash_M (q, u, \gamma\alpha).$$

As usual, $\vdash_M^+$ is the transitive closure of $\vdash_M$, and $\vdash_M^*$ is the reflexive and transitive closure of $\vdash_M$.

A move of the form

$$(p, au, Z\alpha) \vdash_M (q, u, \alpha)$$

where $a \in \Sigma \cup \{\epsilon\}$, is called a *pop move*.

A move on a real input symbol $a \in \Sigma$ causes this input symbol to be consumed, and the reading head advances to the next input symbol. On the other hand, during an ϵ-move, the reading head stays put.

When

$$(p, u, \alpha) \vdash_M^* (q, v, \beta)$$

we say that we have a *computation*.

There are several equivalent ways of defining acceptance by a PDA.

Definition 6.18. Given a PDA

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F),$$

the following languages are defined:

(1) $T(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash_M^* (f, \epsilon, \alpha)$, where $f \in F$, and $\alpha \in \Gamma^*\}$.

We say that $T(M)$ is the *language accepted by M by final state*.

(2) $N(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash_M^* (q, \epsilon, \epsilon)$, where $q \in Q\}$.

We say that $N(M)$ is the *language accepted by M by empty stack*.

(3) $L(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash_M^* (f, \epsilon, \epsilon)$, where $f \in F\}$.

We say that $L(M)$ is the *language accepted by M by final state and empty stack*.

In all cases, note that the input w must be consumed entirely.

The following proposition shows that the acceptance mode does not matter for PDA's. As we will see shortly, it does matter for DPDAs.

Proposition 6.15. *For any language L, the following facts hold.*

(1) If $L = T(M)$ for some PDA M, then $L = L(M')$ for some PDA M'.

(2) If $L = N(M)$ for some PDA M, then $L = L(M')$ for some PDA M'.

(3) If $L = L(M)$ for some PDA M, then $L = T(M')$ for some PDA M'.

(4) If $L = L(M)$ for some PDA M, then $L = N(M')$ for some PDA M'.

In view of Proposition 6.15, the three acceptance modes T, N, L are equivalent.

The following PDA accepts the language

$$L = \{a^n b^n \mid n \geq 1\}$$

by empty stack.

$Q = \{1, 2\}$, $\Gamma = \{Z_0, a\}$;

$(1, a) \in \delta(1, a, Z_0)$,

$(1, aa) \in \delta(1, a, a)$,

$(2, \epsilon) \in \delta(1, b, a)$,

$(2, \epsilon) \in \delta(2, b, a)$.

The following PDA accepts the language

$$L = \{a^n b^n \mid n \geq 1\}$$

by final state (and also by empty stack).

$Q = \{1, 2, 3\}$, $\Gamma = \{Z_0, A, a\}$, $F = \{3\}$;

$(1, A) \in \delta(1, a, Z_0)$,

$(1, aA) \in \delta(1, a, A)$,

$(1, aa) \in \delta(1, a, a)$,

$(2, \epsilon) \in \delta(1, b, a)$,

$(2, \epsilon) \in \delta(2, b, a)$,

$(3, \epsilon) \in \delta(1, b, A)$,

$(3, \epsilon) \in \delta(2, b, A)$.

DPDA's are defined as follows.

Definition 6.19. A PDA

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

is a *deterministic PDA* (for short, *DPDA*), iff the following conditions hold for all $(p, Z) \in Q \times \Gamma$: either

(1) $|\delta(p, a, Z)| = 1$ for all $a \in \Sigma$, and $\delta(p, \epsilon, Z) = \emptyset$, or

(2) $\delta(p, a, Z) = \emptyset$ for all $a \in \Sigma$, and $|\delta(p, \epsilon, Z)| = 1$.

A DPDA *operates in realtime* iff it has no ϵ-transitions.

It turns out that for DPDA's the most general acceptance mode is by final state. Indeed, there are language that can only be accepted deterministically as $T(M)$. The language

$$L = \{a^m b^n \mid m \geq n \geq 1\}$$

is such an example. The problem is that $a^m b$ is a prefix of all strings $a^m b^n$, with $m \geq n \geq 2$.

A language L is a *deterministic context-free language* iff $L = T(M)$ for some DPDA M.

It is easily shown that if $L = N(M)$ (or $L = L(M)$) for some DPDA M, then $L = T(M')$ for some DPDA M' easily constructed from M.

A PDA is *unambiguous* iff for every $w \in \Sigma^*$, there is at most one computation

$$(q_0, w, Z_0) \vdash^* ID_n,$$

where ID_n is an accepting ID.

There are context-free languages that are not accepted by any DPDA. For example, it can be shown that the languages

$$L_1 = \{a^n b^n \mid n \geq 1\} \cup \{a^n b^{2n} \mid n \geq 1\},$$

and

$$L_2 = \{ww^R \mid w \in \{a, b\}^*\},$$

are not accepted by any DPDA.

Also note that unambiguous grammars for these languages can be easily given.

We now show that every context-free language is accepted by a PDA.

6.14 From Context-Free Grammars To PDA's

We show how a PDA can be easily constructed from a context-free grammar. Although simple, the construction is not practical for parsing purposes, since the resulting PDA is horribly nondeterministic.

Given a context-free grammar $G = (V, \Sigma, P, S)$, we define a one-state PDA M as follows:

$Q = \{q_0\}$; $\Gamma = V$; $Z_0 = S$; $F = \emptyset$;

For every rule $(A \to \alpha) \in P$, there is a transition

$$(q_0, \alpha) \in \delta(q_0, \epsilon, A).$$

For every $a \in \Sigma$, there is a transition

$$(q_0, \epsilon) \in \delta(q_0, a, a).$$

The intuition is that a computation of M mimics a leftmost derivation in G. One might say that we have a "pop/expand" PDA.

Proposition 6.16. *Given any context-free grammar $G = (V, \Sigma, P, S)$, the PDA M just described accepts $L(G)$ by empty stack, i.e., $L(G) = N(M)$.*

Proof sketch. The following two claims are proved by induction.

Claim 1:

For all $u, v \in \Sigma^*$ and all $\alpha \in NV^* \cup \{\epsilon\}$, if $S \overset{*}{\underset{lm}{\Longrightarrow}} u\alpha$, then

$$(q_0, uv, S) \vdash^* (q_0, v, \alpha).$$

Claim 2:

For all $u, v \in \Sigma^*$ and all $\alpha \in V^*$, if

$$(q_0, uv, S) \vdash^* (q_0, v, \alpha)$$

then $S \overset{*}{\underset{lm}{\Longrightarrow}} u\alpha$. ☐

We now show how a PDA can be converted to a context-free grammar

6.15 From PDA's To Context-Free Grammars

The construction of a context-free grammar from a PDA is not really difficult, but it is quite messy. The construction is simplified if we first convert a PDA to an equivalent PDA such that for every move $(q, \gamma) \in \delta(p, a, Z)$ (where $a \in \Sigma \cup \{\epsilon\}$), we have $|\gamma| \leq 2$. In some sense, we form a kind of PDA in Chomsky Normal Form.

Proposition 6.17. *Given any PDA*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F),$$

another PDA

$$M' = (Q', \Sigma, \Gamma', \delta', q'_0, Z'_0, F')$$

can be constructed, such that $L(M) = L(M')$ and the following conditions hold:

(1) There is a one-to-one correspondence between accepting computations of M and M';

(2) If M has no ϵ-moves, then M' has no ϵ-moves; If M is unambiguous, then M' is unambiguous;

(3) For all $p \in Q'$, all $a \in \Sigma \cup \{\epsilon\}$, and all $Z \in \Gamma'$, if $(q, \gamma) \in \delta'(p, a, Z)$, then $q \neq q'_0$ and $|\gamma| \leq 2$.

The crucial point of the construction is that accepting computations of a PDA accepting by empty stack and final state can be decomposed into *subcomputations* of the form

$$(p, uv, Z\alpha) \vdash^* (q, v, \alpha),$$

where for every intermediate ID (s, w, β), we have $\beta = \gamma\alpha$ for some $\gamma \neq \epsilon$.

The nonterminals of the grammar constructed from the PDA M are triples of the form $[p, Z, q]$ such that

$$(p, u, Z) \vdash^+ (q, \epsilon, \epsilon)$$

for some $u \in \Sigma^*$.

Given a PDA

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

satisfying the conditions of Proposition 6.17, we construct a context-free grammar $G = (V, \Sigma, P, S)$ as follows:

$$V = \{[p, Z, q] \mid p, q \in Q, Z \in \Gamma\} \cup \Sigma \cup \{S\},$$

where S is a new symbol, and the productions are defined as follows: for all $p, q \in Q$, all $a \in \Sigma \cup \{\epsilon\}$, all $X, Y, Z \in \Gamma$, we have:

(1) $S \to \epsilon \in P$, if $q_0 \in F$;

(2) $S \to a \in P$, if $(f, \epsilon) \in \delta(q_0, a, Z_0)$, and $f \in F$;

(3) $S \to a[p, X, f] \in P$, for every $f \in F$, if $(p, X) \in \delta(q_0, a, Z_0)$;

(4) $S \to a[p, X, s][s, Y, f] \in P$, for every $f \in F$, for every $s \in Q$, if $(p, XY) \in \delta(q_0, a, Z_0)$;

(5) $[p, Z, q] \to a \in P$, if $(q, \epsilon) \in \delta(p, a, Z)$ and $p \neq q_0$;

(6) $[p, Z, s] \to a[q, X, s] \in P$, for every $s \in Q$, if $(q, X) \in \delta(p, a, Z)$ and $p \neq q_0$;

(7) $[p, Z, t] \to a[q, X, s][s, Y, t] \in P$, for every $s, t \in Q$, if $(q, XY) \in \delta(p, a, Z)$ and $p \neq q_0$.

Proposition 6.18. *Given any PDA*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

satisfying the conditions of Proposition 6.17, the context-free grammar $G = (V, \Sigma, P, S)$ *constructed as above generates* $L(M)$*, i.e.,* $L(G) = L(M)$*. Furthermore,* G *is unambiguous iff* M *is unambiguous.*

Proof skecth. We have to prove that

$$\begin{aligned} L(G) = \{w \in \Sigma^+ \mid (q_0, w, Z_0) \vdash^+ (f, \epsilon, \epsilon),\ f \in F\} \\ \cup\ \{\epsilon \mid q_0 \in F\}. \end{aligned}$$

For this, the following claim is proved by induction.

Claim:

For all $p, q \in Q$, all $Z \in \Gamma$, all $k \geq 1$, and all $w \in \Sigma^*$,

$$[p, Z, q] \overset{k}{\underset{lm}{\Longrightarrow}} w \quad \text{iff} \quad (p, w, Z) \vdash^+ (q, \epsilon, \epsilon).$$

Using the claim, it is possible to prove that $L(G) = L(M)$. □

In view of Propositions 6.16 and 6.18, the family of context-free languages is exactly the family of languages accepted by PDA's. It is harder to give a grammatical characterization of the deterministic context-free languages. One method is to use Knuth *$LR(k)$-grammars*.

Another characterization can be given in terms of *strict deterministic grammars* due to Harrison and Havel.

6.16 The Chomsky-Schutzenberger Theorem

Unfortunately, there is no characterization of the context-free languages analogous to the characterization of the regular languages in terms of closure properties ($R(\Sigma)$).

However, there is a famous theorem due to Chomsky and Schutzenberger showing that every context-free language can be obtained from a special language, the *Dyck set*, in terms of homomorphisms, inverse homomorphisms and intersection with the regular languages.

Definition 6.20. Given the alphabet $\Sigma_2 = \{a, b, \overline{a}, \overline{b}\}$, define the relation $\simeq$ on Σ_2^* as follows: For all $u, v \in \Sigma_2^*$,

$$\begin{aligned} u \simeq v \quad \text{iff} \quad \exists x, y \in \Sigma_2^*, \qquad & u = xa\overline{a}y, \qquad v = xy \quad \text{or} \\ & u = xb\overline{b}y, \qquad v = xy. \end{aligned}$$

Let $\simeq^*$ be the reflexive and transitive closure of $\simeq$, and let $D_2 = \{w \in \Sigma_2^* \mid w \simeq^* \epsilon\}$. This is the *Dyck set* on two letters.

It is not hard to prove that D_2 is context-free.

Theorem 6.19. *(Chomsky-Schutzenberger) For every PDA, $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, there is a regular language R and two homomorphisms g, h such that*

$$L(M) = h(g^{-1}(D_2) \cap R).$$

Observe that Theorem 6.19 yields another proof of the fact that the language accepted a PDA is context-free.

Indeed, the context-free languages are closed under, homomorphisms, inverse homomorphisms, intersection with the regular languages, and D_2 is context-free.

From the characterization of a-transducers in terms of homomorphisms, inverse homomorphisms, and intersection with regular languages, we deduce that every context-free language is the image of D_2 under some a-transduction.

Chapter 7

A Survey of LR-Parsing Methods

In this chapter, we give a brief survey on LR-parsing methods. We begin with the definition of characteristic strings and the construction of Knuth's $LR(0)$-characteristic automaton. Next, we describe the shift/reduce algorithm. The need for lookahead sets is motivated by the resolution of conflicts. A unified method for computing FIRST, FOLLOW and LALR(1) lookahead sets is presented. The method uses a same graph algorithm *Traverse* which finds all nodes reachable from a given node and computes the union of predefined sets assigned to these nodes. Hence, the only difference between the various algorithms for computing FIRST, FOLLOW and LALR(1) lookahead sets lies in the fact that the initial sets and the graphs are computed in different ways. The method can be viewed as an efficient way for solving a set of simultaneously recursive equations with set variables. The method is inspired by DeRemer and Pennello's method for computing LALR(1) lookahead sets. However, DeRemer and Pennello use a more sophisticated graph algorithm for finding strongly connected components. We use a slightly less efficient but simpler algorithm (a depth-first search). We conclude with a brief presentation of $LR(1)$ parsers.

7.1 $LR(0)$-Characteristic Automata

The purpose of *LR-parsing*, invented by D. Knuth in the mid sixties, is the following: Given a context-free grammar G, for any terminal string $w \in \Sigma^*$, find out whether w belongs to the language $L(G)$ generated by G, and if so, construct a rightmost derivation of w, in a deterministic fashion. Of course, this is not possible for all context-free grammars, but only for those that correspond to languages that can be recognized by a *deterministic* PDA (DPDA). Knuth's major discovery was that for a certain type of grammars, the $LR(k)$-grammars, a certain kind of DPDA could be constructed from the grammar (*shift/reduce parsers*). The k in $LR(k)$ refers to the amount of *lookahead* that is necessary in order to proceed deterministically. It turns out that $k = 1$ is sufficient, but even in this case, Knuth construction produces very large DPDA's, and his original $LR(1)$ method is not practical. Fortunately, around 1969, Frank DeRemer, in his MIT Ph.D. thesis, investigated a practical restriction of Knuth's method, known as $SLR(k)$, and soon after, the $LALR(k)$ method was

discovered. The $SLR(k)$ and the $LALR(k)$ methods are both based on the construction of the $LR(0)$-*characteristic automaton* from a grammar G, and we begin by explaining this construction. The additional ingredient needed to obtain an $SLR(k)$ or an $LALR(k)$ parser from an $LR(0)$ parser is the computation of lookahead sets. In the SLR case, the FOLLOW sets are needed, and in the $LALR$ case, a more sophisticated version of the FOLLOW sets is needed. We will consider the construction of these sets in the case $k = 1$. We will discuss the shift/reduce algorithm and consider briefly ways of building $LR(1)$-parsing tables.

For simplicity of exposition, we first assume that grammars have no ϵ-rules. This restriction will be lifted in Section 7.10. Given a reduced context-free grammar $G = (V, \Sigma, P, S')$ augmented with start production $S' \to S$, where S' does not appear in any other productions, the set C_G of *characteristic strings of* G is the following subset of V^* (watch out, not Σ^*):

$$C_G = \{\alpha\beta \in V^* \mid S' \overset{*}{\underset{rm}{\Longrightarrow}} \alpha B v \underset{rm}{\Longrightarrow} \alpha\beta v, \\ \alpha, \beta \in V^*, v \in \Sigma^*, B \to \beta \in P\}.$$

In words, C_G is a certain set of prefixes of sentential forms obtained in rightmost derivations: Those obtained by truncating the part of the sentential form immediately following the rightmost symbol in the righthand side of the production applied at the last step.

The fundamental property of LR-parsing, due to D. Knuth, is that C_G is a *regular language*. Furthermore, a DFA, DCG, accepting C_G, can be constructed from G.

Conceptually, it is simpler to construct the DFA accepting C_G in two steps:

(1) First, construct a nondeterministic automaton with ϵ-rules, NCG, accepting C_G.

(2) Apply the subset construction (Rabin and Scott's method) to NCG to obtain the DFA DCG.

In fact, careful inspection of the two steps of this construction reveals that it is possible to construct DCG directly in a single step, and this is the construction usually found in most textbooks on parsing.

The nondeterministic automaton NCG accepting C_G is defined as follows:

The states of N_{C_G} are "marked productions", where a marked production is a string of the form $A \to \alpha$"."β, where $A \to \alpha\beta$ is a production, and "." is a symbol not in V called the "dot" and which can appear anywhere within $\alpha\beta$.

The start state is $S' \to$ "."S, and the transitions are defined as follows:

(a) For every terminal $a \in \Sigma$, if $A \to \alpha$"."$a\beta$ is a marked production, with $\alpha, \beta \in V^*$, then there is a transition on input a from state $A \to \alpha$"."$a\beta$ to state $A \to \alpha a$"."β obtained by "shifting the dot." Such a transition is shown in Figure 7.1.

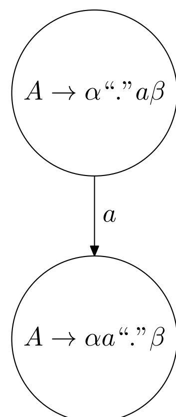


Figure 7.1: Transition on terminal input a

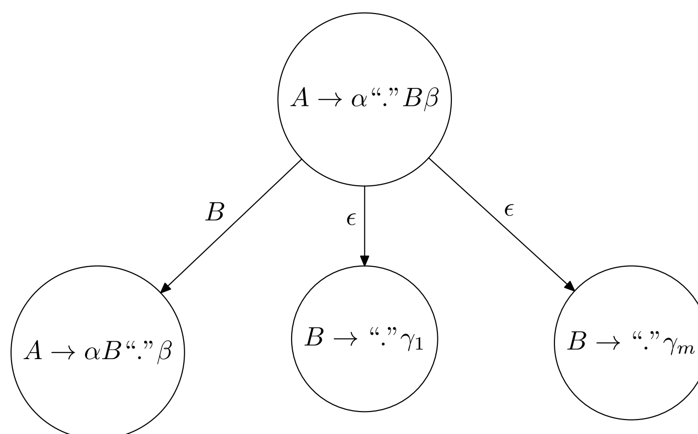


Figure 7.2: Transitions from a state $A \to \alpha``."B\beta$

(b) For every nonterminal $B \in N$, if $A \to \alpha``."B\beta$ is a marked production, with $\alpha, \beta \in V^*$, then there is a transition on input $B$ from state $A \to \alpha``."B\beta$ to state $A \to \alpha B``."\beta$ (obtained by "shifting the dot"), and transitions on input $\epsilon$ (the empty string) to all states $B \to ``."\gamma_i$, for all productions $B \to \gamma_i$ with left-hand side B. Such transitions are shown in Figure 7.2.

(c) A state is *final* if and only if it is of the form $A \to \beta``."$ (that is, the dot is in the rightmost position).

The above construction is illustrated by the following example:

Example 1. Consider the grammar G_1 given by:

$$\begin{aligned} S &\longrightarrow E \\ E &\longrightarrow aEb \\ E &\longrightarrow ab \end{aligned}$$

The NFA for C_{G_1} is shown in Figure 7.3. The result of making the NFA for C_{G_1} deterministic is shown in Figure 7.4 (where transitions to the "dead state" have been omitted). The internal structure of the states $1, \ldots, 6$ is shown below:

$$\begin{aligned} 1: S &\longrightarrow .E \\ E &\longrightarrow .aEb \\ E &\longrightarrow .ab \\ 2: E &\longrightarrow a.Eb \\ E &\longrightarrow a.b \\ E &\longrightarrow .aEb \\ E &\longrightarrow .ab \\ 3: E &\longrightarrow aE.b \\ 4: S &\longrightarrow E. \\ 5: E &\longrightarrow ab. \\ 6: E &\longrightarrow aEb. \end{aligned}$$

The next example is slightly more complicated.

Example 2. Consider the grammar G_2 given by:

$$\begin{aligned} S &\longrightarrow E \\ E &\longrightarrow E + T \\ E &\longrightarrow T \\ T &\longrightarrow T * a \\ T &\longrightarrow a \end{aligned}$$

The result of making the NFA for C_{G_2} deterministic is shown in Figure 7.5 (where transitions to the "dead state" have been omitted). The internal structure of the states $1, \ldots, 8$

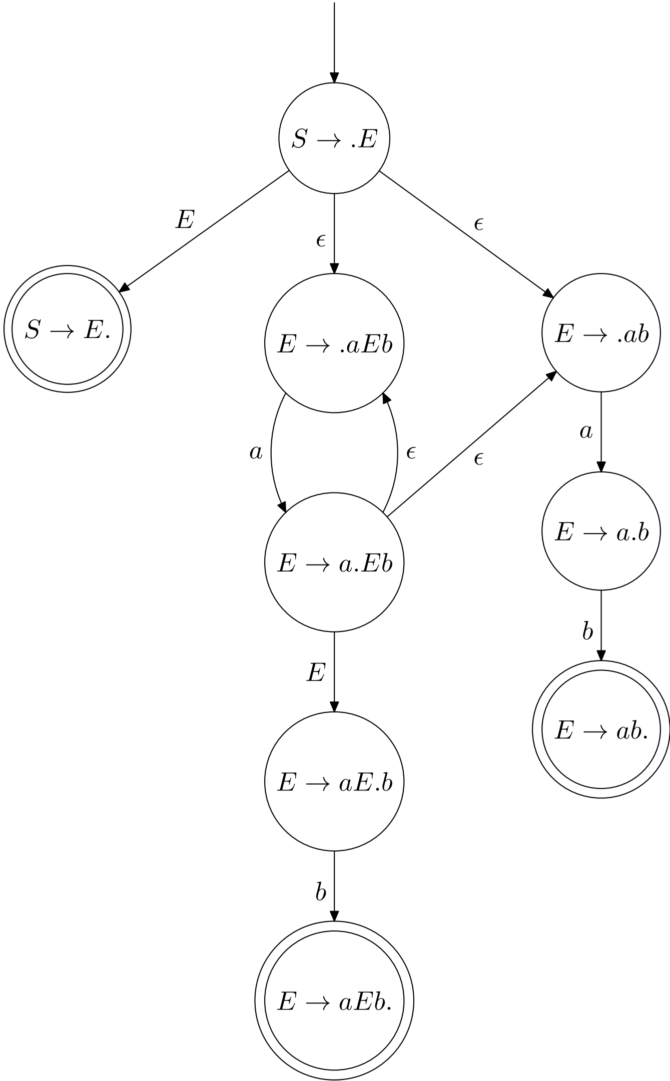


Figure 7.3: NFA for C_{G_1}

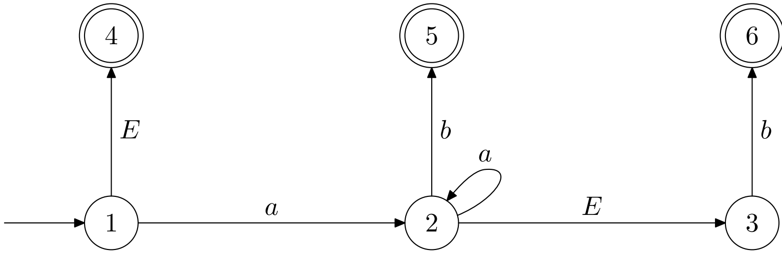


Figure 7.4: DFA for C_{G_1}

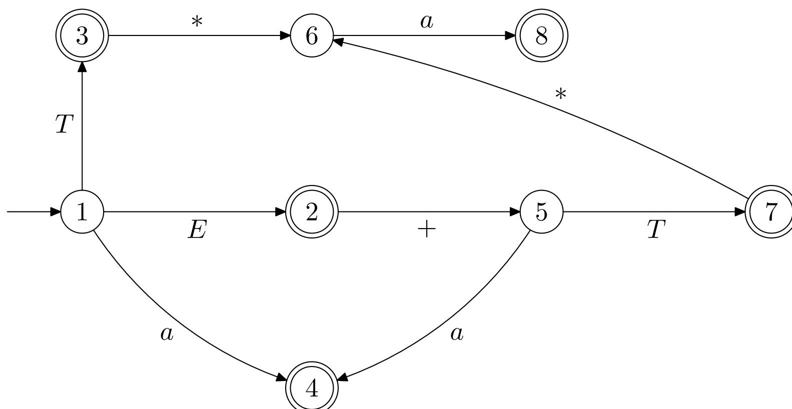


Figure 7.5: DFA for C_{G_2}

is shown below:

$$
\begin{array}{rcl}
1: S & \longrightarrow & .E \\
E & \longrightarrow & .E + T \\
E & \longrightarrow & .T \\
T & \longrightarrow & .T * a \\
T & \longrightarrow & .a \\
2: E & \longrightarrow & E. + T \\
S & \longrightarrow & E. \\
3: E & \longrightarrow & T. \\
T & \longrightarrow & T. * a \\
4: T & \longrightarrow & a. \\
5: E & \longrightarrow & E + .T \\
T & \longrightarrow & .T * a \\
T & \longrightarrow & .a \\
6: T & \longrightarrow & T * .a \\
7: E & \longrightarrow & E + T. \\
T & \longrightarrow & T. * a \\
8: T & \longrightarrow & T * a.
\end{array}
$$

Note that some of the marked productions are more important than others. For example, in state 5, the marked production $E \longrightarrow E + .T$ determines the state. The other two items $T \longrightarrow .T * a$ and $T \longrightarrow .a$ are obtained by ϵ-closure.

We call a marked production of the form $A \longrightarrow \alpha.\beta$, where $\alpha \neq \epsilon$, a *core item*. A marked production of the form $A \longrightarrow \beta.$ is called a *reduce item*. Reduce items only appear in final

states.

If we also call $S' \longrightarrow .S$ a core item, we observe that every state is completely determined by its subset of core items. The other items in the state are obtained via ϵ-closure. We can take advantage of this fact to write a more efficient algorithm to construct in a single pass the $LR(0)$-automaton.

Also observe the so-called *spelling property*: All the transitions entering any given state have the same label.

Given a state s, if s contains both a reduce item $A \longrightarrow \gamma.$ and a shift item $B \longrightarrow \alpha.a\beta$, where $a \in \Sigma$, we say that there is a *shift/reduce conflict* in state s on input a. If s contains two (distinct) reduce items $A_1 \longrightarrow \gamma_1.$ and $A_2 \longrightarrow \gamma_2.$, we say that there is a *reduce/reduce conflict* in state s.

A grammar is said to be $LR(0)$ if the DFA DCG has no conflicts. This is the case for the grammar G_1. However, it should be emphasized that this is extremely rare in practice. The grammar G_1 is just very nice, and a toy example. In fact, G_2 is not $LR(0)$.

To eliminate conflicts, one can either compute $SLR(1)$-lookahead sets, using FOLLOW sets (see Section 7.6), or sharper lookahead sets, the $LALR(1)$ sets (see Section 7.9). For example, the computation of $SLR(1)$-lookahead sets for G_2 will eliminate the conflicts.

We will describe methods for computing $SLR(1)$-lookahead sets and $LALR(1)$-lookahead sets in Sections 7.6, 7.9, and 7.10. A more drastic measure is to compute the $LR(1)$-automaton, in which the states incoporate lookahead symbols (see Section 7.11). However, as we said before, this is not a practical methods for large grammars.

In order to motivate the construction of a shift/reduce parser from the DFA accepting C_G, let us consider a rightmost derivation for $w = aaabbb$ in reverse order for the grammar

$$
\begin{aligned}
0\colon & S \longrightarrow E \\
1\colon & E \longrightarrow aEb \\
2\colon & E \longrightarrow ab
\end{aligned}
$$

$aaabbb$	$\alpha_1\beta_1 v_1$		
$aaEbb$	$\alpha_1 B_1 v_1$		$E \longrightarrow ab$
$aaEbb$	$\alpha_2\beta_2 v_2$		
aEb	$\alpha_2 B_2 v_2$		$E \longrightarrow aEb$
aEb	$\alpha_3\beta_3 v_3$	$\alpha_3 = v_3 = \epsilon$	
E	$\alpha_3 B_3 v_3$	$\alpha_3 = v_3 = \epsilon$	$E \longrightarrow aEb$
E	$\alpha_4\beta_4 v_4$	$\alpha_4 = v_4 = \epsilon$	
S	$\alpha_4 B_4 v_4$	$\alpha_4 = v_4 = \epsilon$	$S \longrightarrow E$

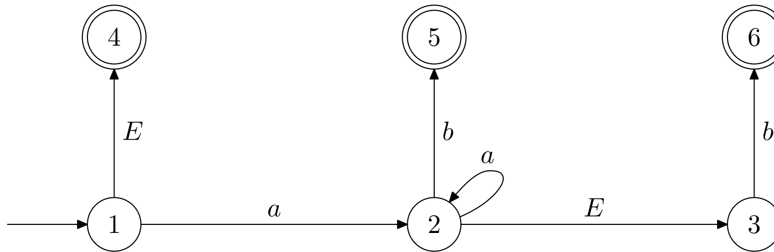


Figure 7.6: DFA for C_G

Observe that the strings $\alpha_i\beta_i$ for $i = 1, 2, 3, 4$ are all accepted by the DFA for C_G shown in Figure 7.6.

Also, every step from $\alpha_i\beta_i v_i$ to $\alpha_i B_i v_i$ is the inverse of the derivation step using the production $B_i \longrightarrow \beta_i$, and the marked production $B_i \longrightarrow \beta_i$" ." is one of the reduce items in the final state reached after processing $\alpha_i\beta_i$ with the DFA for C_G.

This suggests that we can parse $w = aaabbb$ by recursively running the DFA for C_G.

The first time (which correspond to step 1) we run the DFA for C_G on w, some string $\alpha_1\beta_1$ is accepted and the remaining input in v_1.

Then, we "reduce" β_1 to B_1 using a production $B_1 \longrightarrow \beta_1$ corresponding to some reduce item $B_1 \longrightarrow \beta_1$" ." in the final state s_1 reached on input $\alpha_1\beta_1$.

We now run the DFA for C_G on input $\alpha_1 B_1 v_1$. The string $\alpha_2\beta_2$ is accepted, and we have

$$\alpha_1 B_1 v_1 = \alpha_2\beta_2 v_2.$$

We reduce β_2 to B_2 using a production $B_2 \longrightarrow \beta_2$ corresponding to some reduce item $B_2 \longrightarrow \beta_2$" ." in the final state s_2 reached on input $\alpha_2\beta_2$.

We now run the DFA for C_G on input $\alpha_2 B_2 v_2$, and so on.

At the $(i+1)$th step ($i \geq 1$), we run the DFA for C_G on input $\alpha_i B_i v_i$. The string $\alpha_{i+1}\beta_{i+1}$ is accepted, and we have

$$\alpha_i B_i v_i = \alpha_{i+1}\beta_{i+1} v_{i+1}.$$

We reduce β_{i+1} to B_{i+1} using a production $B_{i+1} \longrightarrow \beta_{i+1}$ corresponding to some reduce item $B_{i+1} \longrightarrow \beta_{i+1}$" ." in the final state s_{i+1} reached on input $\alpha_{i+1}\beta_{i+1}$.

The string β_{i+1} in $\alpha_{i+1}\beta_{i+1}v_{i+1}$ if often called a *handle*.

Then we run again the DFA for C_G on input $\alpha_{i+1}B_{i+1}v_{i+1}$.

Now, because the DFA for C_G is *deterministic* there is no need to rerun it on the entire string $\alpha_{i+1}B_{i+1}v_{i+1}$, because *on input* α_{i+1} it will take us to *the same state*, say p_{i+1}, that it reached on input $\alpha_{i+1}\beta_{i+1}v_{i+1}$!

The trick is that we can use a *stack* to keep track of the sequence of states used to process $\alpha_{i+1}\beta_{i+1}$.

Then, to perform the reduction of $\alpha_{i+1}\beta_{i+1}$ to $\alpha_{i+1}B_{i+1}$, we simply *pop* a number of states equal to $|\beta_{i+1}|$, encovering a new state p_{i+1} on top of the stack, and from state p_{i+1} we perform the transition on input B_{i+1} to a state q_{i+1} (in the DFA for C_G), so we *push* state q_{i+1} on the stack which now contains the sequence of states on input $\alpha_{i+1}B_{i+1}$ that takes us to q_{i+1}.

Then we resume scanning v_{i+1} using the DGA for C_G, *pushing* each state being traversed on the stack until we hit a final state.

At this point we find the new string $\alpha_{i+2}\beta_{i+2}$ that leads to a final state and we continue as before.

The process stops when the remaining input v_{i+1} becomes empty and when the reduce item $S' \longrightarrow S.$ (here, $S \longrightarrow E.$) belongs to the final state s_{i+1}.

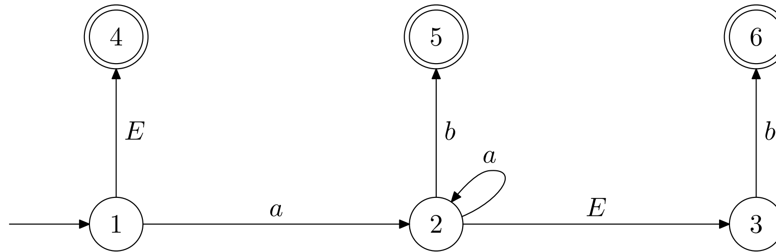


Figure 7.7: DFA for C_G

For example, on input $\alpha_2\beta_2 = aaEbb$, we have the sequence of states:

$$1\ 2\ 2\ 3\ 6$$

State 6 contains the marked production $E \longrightarrow aEb$“.”, so we pop the three topmost states 2 3 6 obtaining the stack

$$1\ 2$$

and then we make the transition from state 2 on input E, which takes us to state 3, so we push 3 on top of the stack, obtaining

$$1\ 2\ 3$$

We continue from state 3 on input b.

Basically, the recursive calls to the DFA for C_G are implemented using a stack.

What is not clear is, during step $i+1$, when reaching a final state s_{i+1}, how do we know which production $B_{i+1} \longrightarrow \beta_{i+1}$ to use in the reduction step?

Indeed, state s_{i+1} could contain several reduce items $B_{i+1} \longrightarrow \beta_{i+1}$ ".".

This is where we assume that we were able to compute some *lookahead information*, that is, for every final state s and every input a, we know which unique production $n\colon B_{i+1} \longrightarrow \beta_{i+1}$ applies. This is recorded in a table name "action," such that $\text{action}(s, a) = rn$, where "r" stands for reduce.

Typically we compute SLR(1) or LALR(1) lookahead sets. Otherwise, we could pick some reducing production nondeterministicallly and use backtracking. This works but the running time may be exponential.

The DFA for C_G and the action table giving us the reductions can be combined to form a bigger action table which specifies completely how the parser using a stack works.

This kind of parser called a *shift-reduce parser* is discussed in the next section.

In order to make it easier to compute the reduce entries in the parsing table, we assume that the end of the input w is signalled by a special endmarker traditionally denoted by \$.

7.2 Shift/Reduce Parsers

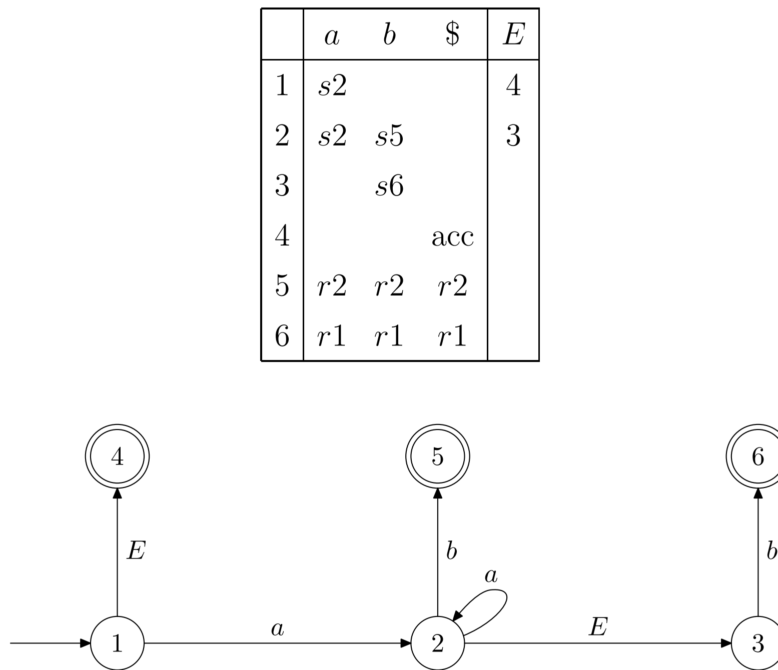
A shift/reduce parser is a modified kind of DPDA. Firstly, push moves, called *shift moves*, are restricted so that exactly one symbol is pushed on top of the stack. Secondly, more powerful kinds of pop moves, called *reduce moves*, are allowed. During a reduce move, a finite number of stack symbols may be popped off the stack, and the last step of a reduce move, called a *goto move*, consists of pushing one symbol on top of new topmost symbol in the stack. Shift/reduce parsers use *parsing tables* constructed from the $LR(0)$-characteristic automaton DCG associated with the grammar. The shift and goto moves come directly from the transition table of DCG, but the determination of the reduce moves requires the computation of *lookahead sets*. The $SLR(1)$ lookahead sets are obtained from some sets called the FOLLOW sets (see Section 7.6), and the $LALR(1)$ lookahead sets $\text{LA}(s, A \longrightarrow \gamma)$ require fancier FOLLOW sets (see Section 7.9).

The construction of shift/reduce parsers is made simpler by assuming that the end of input strings $w \in \Sigma^*$ is indicated by the presence of an *endmarker*, usually denoted \$, and assumed not to belong to Σ.

Consider the grammar G_1 of Example 1, where we have numbered the productions $0, 1, 2$:

$$\begin{aligned} 0: S &\longrightarrow E \\ 1: E &\longrightarrow aEb \\ 2: E &\longrightarrow ab \end{aligned}$$

The parsing tables associated with the grammar G_1 are shown below:



	a	b	\$	E
1	$s2$			4
2	$s2$	$s5$		3
3		$s6$		
4			acc	
5	$r2$	$r2$	$r2$	
6	$r1$	$r1$	$r1$	

Figure 7.8: DFA for C_G

Entries of the form si are *shift actions*, where i denotes one of the states, and entries of the form rn are *reduce actions*, where n denotes a production number (*not* a state). The special action acc means accept, and signals the successful completion of the parse. Entries of the form i, in the rightmost column, are *goto actions*. All blank entries are **error** entries, and mean that the parse should be aborted.

We will use the notation action(s, a) for the entry corresponding to state s and terminal $a \in \Sigma \cup \{\$\}$, and goto$(s, A)$ for the entry corresponding to state s and nonterminal $A \in N - \{S'\}$.

Assuming that the input is $w\$$, we now describe in more detail how a shift/reduce parser proceeds. The parser uses a stack in which states are pushed and popped. Initially, the stack contains state 1 and the cursor pointing to the input is positioned on the leftmost symbol. There are four possibilities:

(1) If action$(s, a) = sj$, then push state j on top of the stack, and advance to the next input symbol in $w\$$. This is a *shift move*.

(2) If action$(s, a) = rn$, then do the following: First, determine the length $k = |\gamma|$ of the righthand side of the production $n\colon A \longrightarrow \gamma$. Then, pop the topmost k symbols off the stack (if $k = 0$, no symbols are popped). If p is the new top state on the stack (after the k pop moves), push the state goto(p, A) on top of the stack, where A is the

lefthand side of the "reducing production" $A \longrightarrow \gamma$. Do not advance the cursor in the current input. This is a *reduce move*.

(3) If action$(s, \$) =$ acc, then accept. The input string w belongs to $L(G)$.

(4) In all other cases, **error**, abort the parse. The input string w does not belong to $L(G)$.

Observe that no explicit state control is needed. The current state is always the current topmost state in the stack. We illustrate below a parse of the input $aaabbb\$$.

stack	remaining input	action
1	$aaabbb\$$	$s2$
12	$aabbb\$$	$s2$
122	$abbb\$$	$s2$
1222	$bbb\$$	$s5$
12225	$bb\$$	$r2$
1223	$bb\$$	$s6$
12236	$b\$$	$r1$
123	$b\$$	$s6$
1236	$\$$	$r1$
14	$\$$	acc

Observe that the sequence of reductions read from bottom-up yields a rightmost derivation of $aaabbb$ from E (or from S, if we view the action acc as the reduction by the production $S \longrightarrow E$). This is a general property of LR-parsers.

The $SLR(1)$ reduce entries in the parsing tables are determined as follows: For every state s containing a reduce item $B \longrightarrow \gamma.$, if $B \longrightarrow \gamma$ is the production number n, enter the action rn for state s and every terminal $a \in \mathrm{FOLLOW}(B)$. If the resulting shift/reduce parser has no conflicts, we say that the grammar is $SLR(1)$. For the $LALR(1)$ reduce entries, enter the action rn for state s and production n: $B \longrightarrow \gamma$, for all $a \in \mathrm{LA}(s, B \longrightarrow \gamma)$. Similarly, if the shift/reduce parser obtained using $LALR(1)$-lookahead sets has no conflicts, we say that the grammar is $LALR(1)$.

7.3 Computation of FIRST

In order to compute the FOLLOW sets, we first need to to compute the FIRST sets! For simplicity of exposition, we first assume that grammars have no ϵ-rules. The general case will be treated in Section 7.10.

Given a context-free grammar $G = (V, \Sigma, P, S')$ (augmented with a start production $S' \longrightarrow S$), for every nonterminal $A \in N = V - \Sigma$, let

$$\mathrm{FIRST}(A) = \{a \mid a \in \Sigma,\ A \stackrel{+}{\Longrightarrow} a\alpha, \text{ for some } \alpha \in V^*\}.$$

For a terminal $a \in \Sigma$, let $\mathrm{FIRST}(a) = \{a\}$. The key to the computation of $\mathrm{FIRST}(A)$ is the following observation: a is in $\mathrm{FIRST}(A)$ if either a is in

$$\mathrm{INITFIRST}(A) = \{a \mid a \in \Sigma,\ A \longrightarrow a\alpha \in P, \text{ for some } \alpha \in V^*\},$$

or a is in

$$\{a \mid a \in \mathrm{FIRST}(B),\ A \longrightarrow B\alpha \in P, \text{ for some } \alpha \in V^*,\ B \neq A\}.$$

Note that the second assertion is true because, if $B \stackrel{+}{\Longrightarrow} a\delta$, then $A \Longrightarrow B\alpha \stackrel{+}{\Longrightarrow} a\delta\alpha$, and so, $\mathrm{FIRST}(B) \subseteq \mathrm{FIRST}(A)$ whenever $A \longrightarrow B\alpha \in P$, with $A \neq B$. Hence, the FIRST sets are the least solution of the following set of recursive equations: For each nonterminal A,

$$\mathrm{FIRST}(A) = \mathrm{INITFIRST}(A) \cup \bigcup \{\mathrm{FIRST}(B) \mid A \longrightarrow B\alpha \in P,\ A \neq B\}.$$

In order to explain the method for solving such systems, we will formulate the problem in more general terms, but first, we describe a "naive" version of the shift/reduce algorithm that hopefully demystifies the "'optimized version" described in Section 7.2.

7.4 The Intuition Behind the Shift/Reduce Algorithm

Let $DCG = (K, V, \delta, q_0, F)$ be the DFA accepting the regular language C_G, and let δ^* be the extension of δ to $K \times V^*$. Let us assume that the grammar G is either $SLR(1)$ or $LALR(1)$, which implies that it has no shift/reduce or reduce/reduce conflicts. We can use the DFA DCG accepting C_G recursively to parse $L(G)$. The function CG is defined as follows: Given any string $\mu \in V^*$,

$$CG(\mu) = \begin{cases} error & \text{if } \delta^*(q_0, \mu) = error; \\ (\delta^*(q_0, \theta), \theta, v) & \text{if } \delta^*(q_0, \theta) \in F,\ \mu = \theta v \text{ and } \theta \text{ is the} \\ & \text{shortest prefix of } \mu \text{ s.t. } \delta^*(q_0, \theta) \in F. \end{cases}$$

The naive shift-reduce algorithm is shown below:

begin
  accept := **true**;
  stop := **false**;
  $\mu := w\$$; {input string}
  while $\neg$*stop* **do**
    if $CG(\mu) = error$ **then**

```
            stop := true; accept := false
        else
            Let (q, θ, v) = CG(μ)
            Let B → β be the production so that
            action(q, FIRST(v)) = B → β and let θ = αβ
            if B → β = S′ → S then
                stop := true
            else
                μ := αBv    {reduction}
            endif
        endif
    endwhile
end
```

The idea is to recursively run the DFA DCG on the sentential form μ, until the first final state q is hit. Then, the sentential form μ must be of the form $\alpha\beta v$, where v is a terminal string ending in \$, and the final state q contains a reduce item of the form $B \longrightarrow \beta$, with $\text{action}(q, \text{FIRST}(v)) = B \longrightarrow \beta$. Thus, we can reduce $\mu = \alpha\beta v$ to $\alpha B v$, since we have found a rightmost derivation step, and repeat the process.

Note that the major inefficiency of the algorithm is that when a reduction is performed, the prefix α of μ is reparsed entirely by DCG. Since DCG is deterministic, the sequence of states obtained on input α is uniquely determined. If we keep the sequence of states produced on input θ by DCG in a stack, then it is possible to avoid reparsing α. Indeed, all we have to do is update the stack so that just before applying DCG to $\alpha A v$, the sequence of states in the stack is the sequence obtained after parsing α. This stack is obtained by popping the $|\beta|$ topmost states and performing an update which is just a goto move. This is the standard version of the shift/reduce algorithm!

7.5 The Graph Method for Computing Fixed Points

Let X be a finite set representing the domain of the problem (in Section 7.3 above, $X = \Sigma$), let $F(1), \ldots, F(N)$ be N sets to be computed and let $I(1), \ldots, I(N)$ be N given subsets of X. The sets $I(1), \ldots, I(N)$ are the initial sets. We also have a directed graph G whose set of nodes is $\{1, \ldots, N\}$ and which represents relationships among the sets $F(i)$, where $1 \leq i \leq N$. The graph G has no parallel edges and no loops, but it may have cycles. If there is an edge from i to j, this is denoted by iGj (note that the absense of loops means that iGi never holds). Also, the existence of a path from i to j is denoted by iG^+j. The graph G represents a relation, and G^+ is the graph of the transitive closure of this relation. The existence of a path from i to j, including the null path, is denoted by iG^*j. Hence, G^* is the

reflexive and transitive closure of G. We want to solve for the least solution of the system of recursive equations:

$$F(i) = I(i) \cup \{F(j) \mid iGj,\, i \neq j\}, \quad 1 \leq i \leq N.$$

Since $(2^X)^N$ is a complete lattice under the inclusion ordering (which means that every family of subsets has a least upper bound, namely, the union of this family), it is an ω-complete poset, and since the function $F\colon (2^X)^N \to (2^X)^N$ induced by the system of equations is easily seen to preserve least upper bounds of ω-chains, the least solution of the system can be computed by the standard fixed point technique (as explained in Section 3.7 of the class notes). We simply compute the sequence of approximations $(F^k(1), \ldots, F^k(N))$, where

$$F^0(i) = \emptyset, \quad 1 \leq i \leq N,$$

and

$$F^{k+1}(i) = I(i) \cup \bigcup \{F^k(j) \mid iGj,\, i \neq j\}, \quad 1 \leq i \leq N.$$

It is easily seen that we can stop at $k = N - 1$, and the least solution is given by

$$F(i) = F^1(i) \cup F^2(i) \cup \cdots \cup F^N(i), \quad 1 \leq i \leq N.$$

However, the above expression can be simplified to

$$F(i) = \bigcup \{I(j) \mid iG^*j\}, \quad 1 \leq i \leq N.$$

This last expression shows that in order to compute $F(i)$, it is necessary to compute the union of all the initial sets $I(j)$ reachable from i (including i). Hence, any transitive closure algorithm or graph traversal algorithm will do. For simplicity and for pedagogical reasons, we use a depth-first search algorithm.

Going back to FIRST, we see that all we have to do is to compute the INITFIRST sets, the graph GFIRST, and then use the graph traversal algorithm. The graph GFIRST is computed as follows: The nodes are the nonterminals and there is an edge from A to B ($A \neq B$) if and only if there is a production of the form $A \longrightarrow B\alpha$, for some $\alpha \in V^*$.

Example 1. Computation of the FIRST sets for the grammar G_1 given by the rules:

$$\begin{aligned}
S &\longrightarrow E\$\\
E &\longrightarrow E + T\\
E &\longrightarrow T\\
T &\longrightarrow T * F\\
T &\longrightarrow F\\
F &\longrightarrow (E)\\
F &\longrightarrow -T\\
F &\longrightarrow a.
\end{aligned}$$

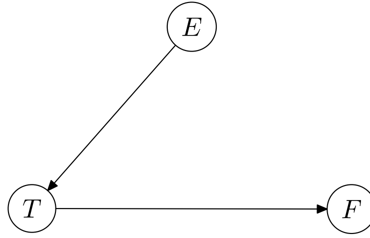


Figure 7.9: Graph GFIRST for G_1

We get

$$\text{INITFIRST}(E) = \emptyset, \quad \text{INITFIRST}(T) = \emptyset, \quad \text{INITFIRST}(F) = \{(, -, a\}.$$

The graph GFIRST is shown in Figure 7.9.
We obtain the following FIRST sets:

$$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{(, -, a\}.$$

7.6 Computation of FOLLOW

Recall the definition of FOLLOW(A) for a nonterminal A:

$$\text{FOLLOW}(A) = \{a \mid a \in \Sigma,\ S \stackrel{+}{\Longrightarrow} \alpha A a \beta, \quad \text{for some } \alpha, \beta \in V^*\}.$$

Note that a is in FOLLOW(A) if either a is in

$$\text{INITFOLLOW}(A) = \{a \mid a \in \Sigma,\ B \longrightarrow \alpha A X \beta \in P,\ a \in \text{FIRST}(X),\ \alpha, \beta \in V^*\}$$

or a is in

$$\{a \mid a \in \text{FOLLOW}(B),\ B \longrightarrow \alpha A \in P,\ \alpha \in V^*,\ A \neq B\}.$$

Indeed, if $S \stackrel{+}{\Longrightarrow} \lambda B a \rho$, then $S \stackrel{+}{\Longrightarrow} \lambda B a \rho \Longrightarrow \lambda \alpha A a \rho$, and so,

$$\text{FOLLOW}(B) \subseteq \text{FOLLOW}(A)$$

whenever $B \longrightarrow \alpha A$ is in P, with $A \neq B$. Hence, the FOLLOW sets are the least solution of the set of recursive equations: For all nonterminals A,

$$\text{FOLLOW}(A) = \text{INITFOLLOW}(A) \cup \bigcup \{\text{FOLLOW}(B) \mid B \longrightarrow \alpha A \in P,\ \alpha \in V^*,\ A \neq B\}.$$

According to the method explained above, we just have to compute the INITFOLLOW sets (using FIRST) and the graph GFOLLOW, which is computed as follows: The nodes are the nonterminals and there is an edge from A to B ($A \neq B$) if and only if there is a production

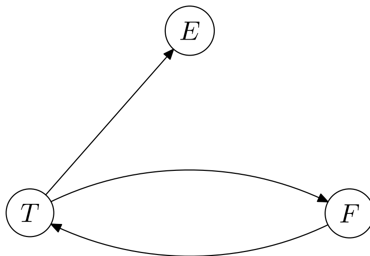


Figure 7.10: Graph GFOLLOW for G_1

of the form $B \longrightarrow \alpha A$ in P, for some $\alpha \in V^*$. Note the duality between the construction of the graph GFIRST and the graph GFOLLOW.

Example 2. Computation of the FOLLOW sets for the grammar G_1.

$$\text{INITFOLLOW}(E) = \{+,), \$\},\ \text{INITFOLLOW}(T) = \{*\},\ \text{INITFOLLOW}(F) = \emptyset.$$

The graph GFOLLOW is shown in Figure 7.10. We have

$$\begin{aligned} \text{FOLLOW}(E) &= \text{INITFOLLOW}(E), \\ \text{FOLLOW}(T) &= \text{INITFOLLOW}(T) \cup \text{INITFOLLOW}(E) \cup \text{INITFOLLOW}(F), \\ \text{FOLLOW}(F) &= \text{INITFOLLOW}(F) \cup \text{INITFOLLOW}(T) \cup \text{INITFOLLOW}(E), \end{aligned}$$

and so

$$\text{FOLLOW}(E) = \{+,), \$\}, \quad \text{FOLLOW}(T) = \{+, *,), \$\}, \quad \text{FOLLOW}(F) = \{+, *,), \$\}.$$

7.7 Algorithm *Traverse*

The input is a directed graph Gr having N nodes, and a family of initial sets $I[i]$, $1 \leq i \leq N$. We assume that a function *successors* is available, which returns for each node n in the graph, the list *successors*$[n]$ of all immediate successors of n. The output is the list of sets $F[i]$, $1 \leq i \leq N$, solution of the system of recursive equations of Section 7.5. Hence,

$$F[i] = \bigcup \{I[j] \mid iG^* j\}, \quad 1 \leq i \leq N.$$

The procedure *Reachable* visits all nodes reachable from a given node. It uses a stack $STACK$ and a boolean array $VISITED$ to keep track of which nodes have been visited. The procedures *Reachable* and *traverse* are shown in Figure 7.11.

```
Procedure Reachable(Gr : graph; startnode : node; I : listofsets;
                    var F : listofsets);
var currentnode, succnode, i : node; STACK : stack;
                    VISITED : array[1..N] of boolean;
  begin
    for i := 1 to N do
      VISITED[i] := false;
    STACK := EMPTY;
    push(STACK, startnode);
    while STACK ≠ EMPTY do
      begin
        currentnode := top(STACK); pop(STACK);
        VISITED[currentnode] := true;
        for each succnode ∈ successors(currentnode) do
          if ¬VISITED[succnode] then
            begin
              push(STACK, succnode);
              F[startnode] := F[startnode] ∪ I[succnode]
            end
      end
  end
```

The sets $F[i]$, $1 \leq i \leq N$, are computed as follows:

```
begin
  for i := 1 to N do
    F[i] := I[i];
  for startnode := 1 to N do
    Reachable(Gr, startnode, I, F)
end
```

Figure 7.11: Algorithm *traverse*

7.8 More on $LR(0)$-Characteristic Automata

Let $G = (V, \Sigma, P, S')$ be an augmented context-free grammar with augmented start production $S' \longrightarrow S\$$ (where S' only occurs in the augmented production). The righmost derivation relation is denoted by $\underset{rm}{\Longrightarrow}$.

Recall that the *set* C_G *of characteristic strings* for the grammar G is defined by

$$C_G = \{\alpha\beta \in V^* \mid S' \overset{*}{\underset{rm}{\Longrightarrow}} \alpha A v \underset{rm}{\Longrightarrow} \alpha\beta v,\ \alpha\beta \in V^*,\ v \in \Sigma^*\}.$$

The fundamental property of LR-parsing, due to D. Knuth, is stated in the following theorem:

Theorem 7.1. *Let G be a context-free grammar and assume that every nonterminal derives some terminal string. The language C_G (over V^*) is a regular language. Furthermore, a deterministic automaton DCG accepting C_G can be constructed from G.*

The construction of DCG can be found in various places, including the book on Compilers by Aho, Sethi and Ullman. We explained this construction in Section 7.1. The proof that the NFA NCG constructed as indicated in Section 7.1 is correct, i.e., that it accepts precisely C_G, is nontrivial, but not really hard either. This will be the object of a homework assignment! However, note a subtle point: The construction of NCG is only correct under the assumption that every nonterminal derives some terminal string. Otherwise, the construction could yield an NFA NCG accepting strings **not in** C_G.

Recall that the states of the characteristic automaton CGA are sets of *items* (or *marked productions*), where an item is a production with a dot anywhere in its right-hand side. Note that in constructing CGA, it is not necessary to include the state $\{S' \longrightarrow S\$.\}$ (the endmarker \$ is only needed to compute the lookahead sets). If a state p contains a marked production of the form $A \longrightarrow \beta.$, where the dot is the rightmost symbol, state p is called a *reduce state* and $A \longrightarrow \beta$ is called a *reducing production* for p. Given any state q, we say that a string $\beta \in V^*$ *accesses* q if there is a path from some state p to the state q on input β in the automaton CGA. Given any two states $p, q \in CGA$, for any $\beta \in V^*$, if there is a sequence of transitions in CGA from p to q on input β, this is denoted by

$$p \overset{\beta}{\longrightarrow} q.$$

The initial state which is the closure of the item $S' \longrightarrow .S\$$ is denoted by 1. The LALR(1)-lookahead sets are defined in the next section.

7.9 LALR(1)-Lookahead Sets

For any reduce state q and any reducing production $A \longrightarrow \beta$ for q, let

$$\mathrm{LA}(q, A \longrightarrow \beta) = \{a \mid a \in \Sigma,\ S' \overset{*}{\underset{rm}{\Longrightarrow}} \alpha A a v \underset{rm}{\Longrightarrow} \alpha\beta a v,\ \alpha, \beta \in V^*,\ v \in \Sigma^*,\ \alpha\beta \textit{ accesses } q\}.$$

In words, $LA(q, A \longrightarrow \beta)$ consists of the terminal symbols for which the reduction by production $A \longrightarrow \beta$ in state q is the correct action (that is, for which the parse will terminate successfully). The LA sets can be computed using the FOLLOW sets defined below.

For any state p and any nonterminal A, let

$$\mathrm{FOLLOW}(p, A) = \{a \mid a \in \Sigma,\ S' \overset{*}{\underset{rm}{\Longrightarrow}} \alpha A a v,\ \alpha \in V^*,\ v \in \Sigma^* \text{ and } \alpha \text{ accesses } p\}.$$

Since for any derivation

$$S' \overset{*}{\underset{rm}{\Longrightarrow}} \alpha A a v \underset{rm}{\Longrightarrow} \alpha\beta a v$$

where $\alpha\beta$ accesses q, there is a state p such that $p \overset{\beta}{\longrightarrow} q$ and α accesses p, it is easy to see that the following result holds:

Proposition 7.2. *For every reduce state q and any reducing production $A \longrightarrow \beta$ for q, we have*

$$\mathrm{LA}(q, A \longrightarrow \beta) = \bigcup \{\mathrm{FOLLOW}(p, A) \mid p \overset{\beta}{\longrightarrow} q\}.$$

Also, we let

$$\mathrm{LA}(\{S' \longrightarrow S.\$\}, S' \longrightarrow S\$) = \mathrm{FOLLOW}(1, S).$$

Intuitively, when the parser makes the reduction by production $A \longrightarrow \beta$ in state q, each state p as above is a possible top of stack after the states corresponding to β are popped. Then the parser must read A in state p, and the next input symbol will be one of the symbols in $\mathrm{FOLLOW}(p, A)$.

The computation of $\mathrm{FOLLOW}(p, A)$ is similar to that of $\mathrm{FOLLOW}(A)$. First, we compute $\mathrm{INITFOLLOW}(p, A)$, given by

$$\mathrm{INITFOLLOW}(p, A) = \{a \mid a \in \Sigma,\ \exists q, r,\ p \overset{A}{\longrightarrow} q \overset{a}{\longrightarrow} r\}.$$

These are the terminals that can be read in CGA after the "goto transition" on nonterminal A has been performed from p. These sets can be easily computed from CGA.

Note that for the state p whose core item is $S' \longrightarrow S.\$$, we have

$$\mathrm{INITFOLLOW}(p, S) = \{\$\}.$$

Next, observe that if $B \longrightarrow \alpha A$ is a production and if

$$S' \overset{*}{\underset{rm}{\Longrightarrow}} \lambda B a v$$

where λ accesses p', then

$$S' \overset{*}{\underset{rm}{\Longrightarrow}} \lambda B a v \underset{rm}{\Longrightarrow} \lambda\alpha A a v$$

where λ accesses p' and $p' \stackrel{\alpha}{\longrightarrow} p$. Hence $\lambda\alpha$ accesses p and

$$\text{FOLLOW}(p', B) \subseteq \text{FOLLOW}(p, A)$$

whenever there is a production $B \longrightarrow \alpha A$ and $p' \stackrel{\alpha}{\longrightarrow} p$. From this, the following recursive equations are easily obtained: For all p and all A,

$$\begin{aligned}\text{FOLLOW}(p, A) = \text{INITFOLLOW}(p, A) \cup \\ \bigcup\{\text{FOLLOW}(p', B) \mid B \longrightarrow \alpha A \in P,\ \alpha \in V^* \text{ and } p' \stackrel{\alpha}{\longrightarrow} p\}.\end{aligned}$$

From Section 7.5, we know that these sets can be computed by using the algorithm *traverse*. All we need is to compute the graph GLA.

The nodes of the graph GLA are the pairs (p, A), where p is a state and A is a nonterminal. There is an edge from (p, A) to (p', B) if and only if there is a production of the form $B \longrightarrow \alpha A$ in P for some $\alpha \in V^*$ and $p' \stackrel{\alpha}{\longrightarrow} p$ in CGA. Note that it is only necessary to consider nodes (p, A) for which there is a nonterminal transition on A from p. Such pairs can be obtained from the parsing table. Also, using the *spelling property*, that is, the fact that all transitions entering a given state have the same label, it is possible to compute the relation *lookback* defined as follows:

$$(q, A)\ \textit{lookback}\ (p, A) \quad \text{iff} \quad p \stackrel{\beta}{\longrightarrow} q$$

for some reduce state q and reducing production $A \longrightarrow \beta$. The above considerations show that the FOLLOW sets of Section 7.6 are obtained by ignoring the state component from FOLLOW(p, A). We now consider the changes that have to be made when ϵ-rules are allowed.

7.10 Computing FIRST, FOLLOW, etc. in the Presence of ϵ-Rules

[Computing FIRST, FOLLOW and LA$(q, A \longrightarrow \beta)$ in the Presence of ϵ-Rules] First, it is necessary to compute the set E of *erasable nonterminals*, that is, the set of nonterminals A such that $A \stackrel{+}{\Longrightarrow} \epsilon$.

We let E be a boolean array and *change* be a boolean flag. An algorithm for computing E is shown in Figure 7.12. Then, in order to compute FIRST, we compute

$$\begin{aligned}\text{INITFIRST}(A) = \{a \mid a \in \Sigma,\ A \longrightarrow a\alpha \in P, \text{ or} \\ A \longrightarrow A_1 \cdots A_k a\alpha \in P, \text{ for some } \alpha \in V^*, \text{ and } E(A_1) = \cdots = E(A_k) = \textbf{true}\}.\end{aligned}$$

The graph GFIRST is obtained as follows: The nodes are the nonterminals, and there is an edge from A to B if and only if either there is a production $A \longrightarrow B\alpha$, or a production $A \longrightarrow A_1 \cdots A_k B\alpha$, for some $\alpha \in V^*$, with $E(A_1) = \cdots = E(A_k) = \textbf{true}$. Then, we extend

```
begin
    for each nonterminal A do
        E(A) := false;
    for each nonterminal A such that A ⟶ ϵ ∈ P do
        E(A) := true;
    change := true;
    while change do
        begin
            change := false;
            for each A ⟶ A_1 ··· A_n ∈ P
                    s.t. E(A_1) = ··· = E(A_n) = true do
                if E(A) = false then
                    begin
                        E(A) := true;
                        change := true
                    end
        end
end
```

Figure 7.12: Algorithm for computing E

FIRST to strings in V^+, in the obvious way. Given any string $\alpha \in V^+$, if $|\alpha| = 1$, then $\beta = X$ for some $X \in V$, and

$$\text{FIRST}(\beta) = \text{FIRST}(X)$$

as before, else if $\beta = X_1 \cdots X_n$ with $n \geq 2$ and $X_i \in V$, then

$$\text{FIRST}(\beta) = \text{FIRST}(X_1) \cup \cdots \cup \text{FIRST}(X_k),$$

where k, $1 \leq k \leq n$, is the largest integer so that

$$E(X_1) = \cdots = E(X_k) = \textbf{true}.$$

To compute FOLLOW, we first compute

$$\text{INITFOLLOW}(A) = \{a \mid a \in \Sigma,\ B \longrightarrow \alpha A \beta \in P,\ \alpha \in V^*,\ \beta \in V^+, \text{ and } a \in \text{FIRST}(\beta)\}.$$

The graph GFOLLOW is computed as follows: The nodes are the nonterminals. There is an edge from A to B if either there is a production of the form $B \longrightarrow \alpha A$, or $B \longrightarrow \alpha A A_1 \cdots A_k$, for some $\alpha \in V^*$, and with $E(A_1) = \cdots = E(A_k) = \textbf{true}$.

The computation of the LALR(1) lookahead sets is also more complicated because another graph is needed in order to compute INITFOLLOW(p, A). First, the graph GLA is defined in the following way: The nodes are still the pairs (p, A), as before, but there is an edge from (p, A) to (p', B) if and only if either there is some production $B \longrightarrow \alpha A$, for some $\alpha \in V^*$ and $p' \stackrel{\alpha}{\longrightarrow} p$, or a production $B \longrightarrow \alpha A \beta$, for some $\alpha \in V^*$, $\beta \in V^+$, $\beta \stackrel{+}{\Longrightarrow} \epsilon$, and $p' \stackrel{\alpha}{\longrightarrow} p$. The sets INITFOLLOW$(p, A)$ are computed in the following way: First, let

$$\text{DR}(p, A) = \{a \mid a \in \Sigma,\ \exists q, r,\ p \stackrel{A}{\longrightarrow} q \stackrel{a}{\longrightarrow} r\}.$$

The sets DR(p, A) are the *direct read* sets. Note that for the state p whose core item is $S' \longrightarrow S.\$$, we have

$$\text{DR}(p, S) = \{\$\}.$$

Then,

$$\begin{aligned} &\text{INITFOLLOW}(p, A) = \text{DR(p, A)} \cup \\ &\quad \bigcup \{a \mid a \in \Sigma,\ S' \underset{rm}{\stackrel{*}{\Longrightarrow}} \alpha A \beta a v \underset{rm}{\Longrightarrow} \alpha A a v,\ \alpha \in V^*,\ \beta \in V^+,\ \beta \stackrel{+}{\Longrightarrow} \epsilon,\ \alpha\ accesses\ p\}. \end{aligned}$$

The set INITFOLLOW(p, A) is the set of terminals that can be read before any handle containing A is reduced. The graph $GREAD$ is defined as follows: The nodes are the pairs (p, A), and there is an edge from (p, A) to (r, C) if and only if $p \stackrel{A}{\longrightarrow} r$ and $r \stackrel{C}{\longrightarrow} s$, for some s, with $E(C) = \textbf{true}$.

Then, it is not difficult to show that the INITFOLLOW sets are the least solution of the set of recursive equations:

$$\text{INITFOLLOW}(p, A) = DR(p, A) \cup \bigcup \{\text{INITFOLLOW}(r, C) \mid (p, A)\ GREAD\ (r, C)\}.$$

Hence the INITFOLLOW sets can be computed using the algorithm traverse on the graph $GREAD$ and the sets $DR(p, A)$, and then, the FOLLOW sets can be computed using traverse again, with the graph GLA and sets INITFOLLOW. Finally, the sets $\mathrm{LA}(q, A \longrightarrow \beta)$ are computed from the FOLLOW sets using the graph *lookback*.

From section 7.5, we note that $F(i) = F(j)$ whenever there is a path from i to j and a path from j to i, that is, whenever i and j are *strongly connected*. Hence, the solution of the system of recursive equations can be computed more efficiently by finding the maximal strongly connected components of the graph G, since F has a same value on each strongly connected component. This is the approach followed by DeRemer and Pennello in: Efficient Computation of LALR(1) Lookahead sets, by F. DeRemer and T. Pennello, *TOPLAS*, Vol. 4, No. 4, October 1982, pp. 615-649.

We now give an example of grammar which is LALR(1) but not SLR(1).

Example 3. The grammar G_2 is given by:

$$
\begin{array}{rcl}
S' & \longrightarrow & S\$ \\
S & \longrightarrow & L = R \\
S & \longrightarrow & R \\
L & \longrightarrow & *R \\
L & \longrightarrow & id \\
R & \longrightarrow & L
\end{array}
$$

The states of the characteristic automaton CGA_2 are:

$$
\begin{array}{rcl}
1: S' & \longrightarrow & .S\$ \\
S & \longrightarrow & .L = R \\
S & \longrightarrow & .R \\
L & \longrightarrow & . * R \\
L & \longrightarrow & .id \\
R & \longrightarrow & .L \\
2: S' & \longrightarrow & S.\$ \\
3: S & \longrightarrow & L. = R \\
R & \longrightarrow & L. \\
4: S & \longrightarrow & R. \\
5: L & \longrightarrow & *.R \\
R & \longrightarrow & .L \\
L & \longrightarrow & . * R \\
L & \longrightarrow & .id \\
6: L & \longrightarrow & id. \\
7: S & \longrightarrow & L = .R \\
R & \longrightarrow & .L \\
L & \longrightarrow & . * R \\
L & \longrightarrow & .id \\
8: L & \longrightarrow & *R. \\
9: R & \longrightarrow & L. \\
10: S & \longrightarrow & L = R.
\end{array}
$$

We find that

$$
\begin{array}{rcl}
\text{INITFIRST}(S) & = & \emptyset \\
\text{INITFIRST}(L) & = & \{*, id\} \\
\text{INITFIRST}(R) & = & \emptyset.
\end{array}
$$

The graph GFIRST is shown in Figure 7.13.
Then, we find that

$$
\begin{array}{rcl}
\text{FIRST}(S) & = & \{*, id\} \\
\text{FIRST}(L) & = & \{*, id\} \\
\text{FIRST}(R) & = & \{*, id\}.
\end{array}
$$

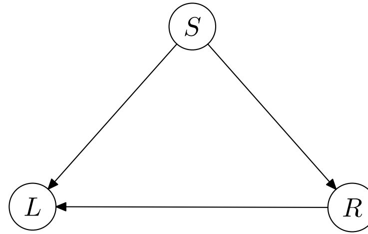


Figure 7.13: The graph GFIRST

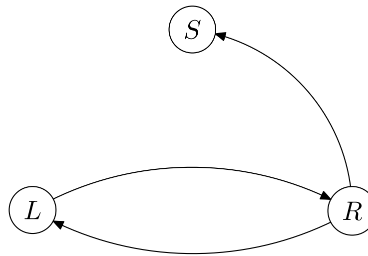


Figure 7.14: The graph GFOLLOW

We also have

$$\begin{aligned} \text{INITFOLLOW}(S) &= \{\$\} \\ \text{INITFOLLOW}(L) &= \{=\} \\ \text{INITFOLLOW}(R) &= \emptyset. \end{aligned}$$

The graph GFOLLOW is shown in Figure 7.14.

Then, we find that

$$\begin{aligned} \text{FOLLOW}(S) &= \{\$\} \\ \text{FOLLOW}(L) &= \{=, \$\} \\ \text{FOLLOW}(R) &= \{=, \$\}. \end{aligned}$$

Note that there is a shift/reduce conflict in state 3 on input $=$, since there is a shift on input $=$ (since $S \longrightarrow L. = R$ is in state 3), and a reduce for $R \rightarrow L$, since $=$ is in FOLLOW(R). However, as we shall see, the conflict is resolved if the LALR(1) lookahead sets are computed.

The graph GLA is shown in Figure 7.15.

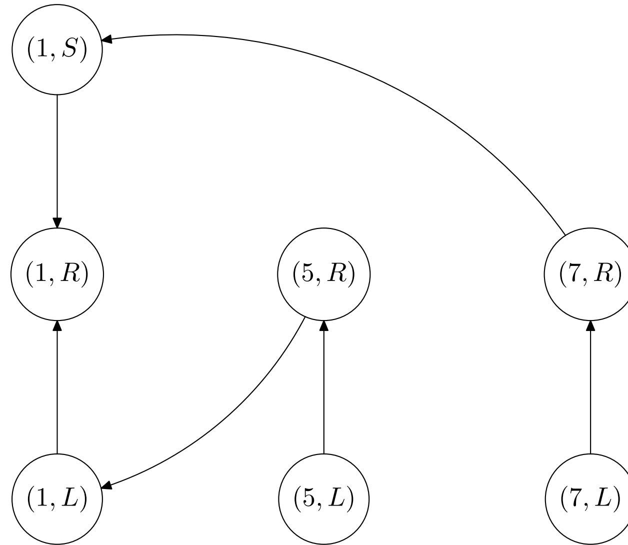


Figure 7.15: The graph GLA

We get the following INITFOLLOW and FOLLOW sets:

$$\begin{array}{rcrc}
\text{INITFOLLOW}(1, S) = & \{\$\} & \text{INITFOLLOW}(1, S) = & \{\$\} \\
\text{INITFOLLOW}(1, R) = & \emptyset & \text{INITFOLLOW}(1, R) = & \{\$\} \\
\text{INITFOLLOW}(1, L) = & \{=\} & \text{INITFOLLOW}(1, L) = & \{=, \$\} \\
\text{INITFOLLOW}(5, R) = & \emptyset & \text{INITFOLLOW}(5, R) = & \{=, \$\} \\
\text{INITFOLLOW}(5, L) = & \emptyset & \text{INITFOLLOW}(5, L) = & \{=, \$\} \\
\text{INITFOLLOW}(7, R) = & \emptyset & \text{INITFOLLOW}(7, R) = & \{\$\} \\
\text{INITFOLLOW}(7, L) = & \emptyset & \text{INITFOLLOW}(7, L) = & \{\$\}.
\end{array}$$

Thus, we get

$$\begin{array}{rcl}
\text{LA}(2, S' \longrightarrow S\$) & = & \text{FOLLOW}(1, S) = \{\$\} \\
\text{LA}(3, R \longrightarrow L) & = & \text{FOLLOW}(1, R) = \{\$\} \\
\text{LA}(4, S \longrightarrow R) & = & \text{FOLLOW}(1, S) = \{\$\} \\
\text{LA}(6, L \longrightarrow id) & = & \text{FOLLOW}(1, L) \cup \text{FOLLOW}(5, L) \cup \text{FOLLOW}(7, L) = \{=, \$\} \\
\text{LA}(8, L \longrightarrow *R) & = & \text{FOLLOW}(1, L) \cup \text{FOLLOW}(5, L) \cup \text{FOLLOW}(7, L) = \{=, \$\} \\
\text{LA}(9, R \longrightarrow L) & = & \text{FOLLOW}(5, R) \cup \text{FOLLOW}(7, R) = \{=, \$\} \\
\text{LA}(10, S \longrightarrow L = R) & = & \text{FOLLOW}(1, S) = \{\$\}.
\end{array}$$

Since $\text{LA}(3, R \longrightarrow L)$ does not contain $=$, the conflict is resolved.

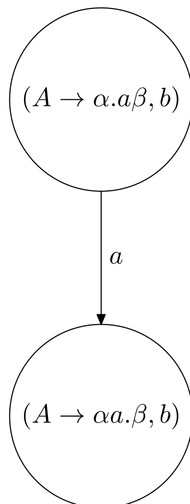


Figure 7.16: Transition on terminal input a

7.11 $LR(1)$-Characteristic Automata

We conclude this brief survey on LR-parsing by describing the construction of $LR(1)$-parsers. The new ingredient is that when we construct an NFA accepting C_G, we incorporate lookahead symbols into the states. Thus, a state is a pair $(A \longrightarrow \alpha.\beta, b)$, where $A \longrightarrow \alpha.\beta$ is a marked production, as before, and $b \in \Sigma \cup \{\$\}$ is a *lookahead symbol*. The new twist in the construction of the nondeterministic characteristic automaton is the following:

The start state is $(S' \rightarrow .S, \$)$, and the transitions are defined as follows:

(a) For every terminal $a \in \Sigma$, then there is a transition on input a from state $(A \rightarrow \alpha.a\beta, b)$ to the state $(A \rightarrow \alpha a.\beta, b)$ obtained by "shifting the dot" (where $a = b$ is possible). Such a transition is shown in Figure 7.16.

(b) For every nonterminal $B \in N$, there is a transition on input B from state $(A \rightarrow \alpha.B\beta, b)$ to state $(A \rightarrow \alpha B.\beta, b)$ (obtained by "shifting the dot"), and transitions on input ϵ (the empty string) to all states $(B \rightarrow .\gamma, a)$, for all productions $B \rightarrow \gamma$ with left-hand side B and all $a \in \mathrm{FIRST}(\beta b)$. Such transitions are shown in Figure 7.17.

(c) A state is *final* if and only if it is of the form $(A \rightarrow \beta., b)$ (that is, the dot is in the rightmost position).

Example 3. Consider the grammar G_3 given by:

$$
\begin{aligned}
0\colon S &\longrightarrow E \\
1\colon E &\longrightarrow aEb \\
2\colon E &\longrightarrow \epsilon
\end{aligned}
$$

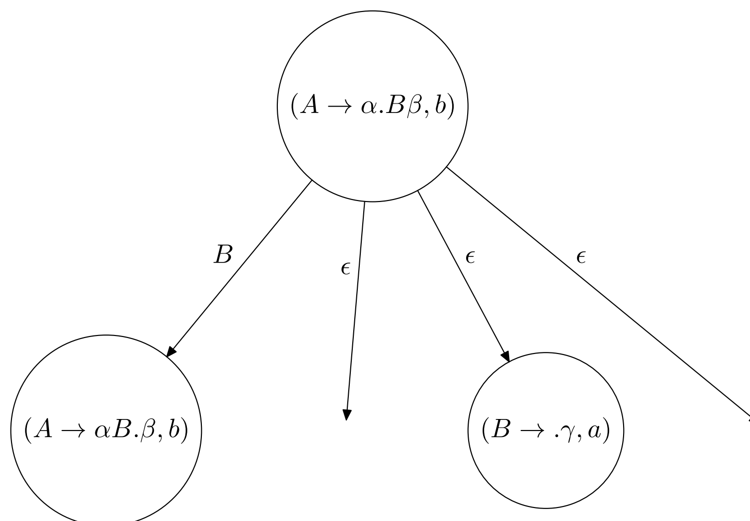


Figure 7.17: Transitions from a state $(A \to \alpha.B\beta, b)$

The result of making the NFA for C_{G_3} deterministic is shown in Figure 7.18 (where transitions to the "dead state" have been omitted). The internal structure of the states $1, \ldots, 8$ is shown below:

$$
\begin{aligned}
1: S &\longrightarrow .E, \$ \\
E &\longrightarrow .aEb, \$ \\
E &\longrightarrow ., \$ \\
2: E &\longrightarrow a.Eb, \$ \\
E &\longrightarrow .aEb, b \\
E &\longrightarrow ., b \\
3: E &\longrightarrow a.Eb, b \\
E &\longrightarrow .aEb, b \\
E &\longrightarrow ., b \\
4: E &\longrightarrow aE.b, \$ \\
5: E &\longrightarrow aEb., \$ \\
6: E &\longrightarrow aE.b, b \\
7: E &\longrightarrow aEb., b \\
8: S &\longrightarrow E., \$
\end{aligned}
$$

The $LR(1)$-shift/reduce parser associated with DCG is built as follows: The shift and goto entries come directly from the transitions of DCG, and for every state s, for every item

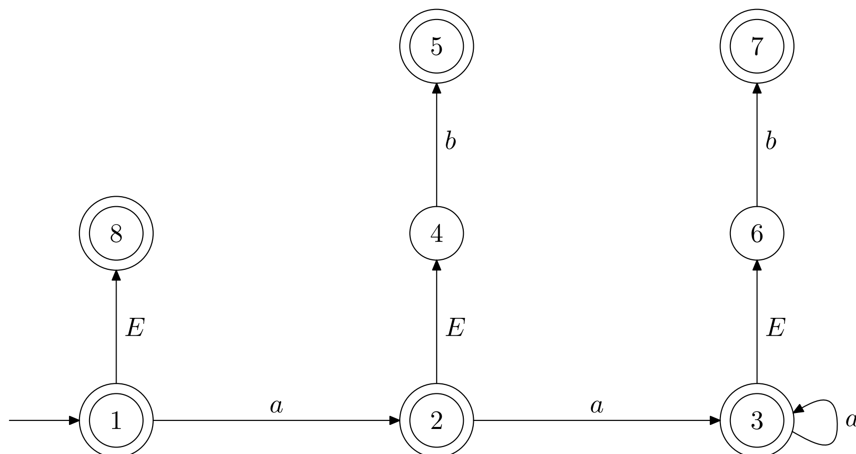


Figure 7.18: DFA for C_{G_3}

$(A \longrightarrow \gamma, b)$ in s, enter an entry rn for state s and input b, where $A \longrightarrow \gamma$ is production number n. If the resulting parser has no conflicts, we say that the grammar is an $LR(1)$ grammar. The $LR(1)$-shift/reduce parser for G_3 is shown below:

	a	b	$\$$	E
1	$s2$		$r2$	8
2	$s3$	$r2$		4
3	$s3$	$r2$		6
4		$r5$		
5			$r1$	
6	$r1$	$s7$		
7		$r1$		
8			acc	

Observe that there are three pairs of states, $(2, 3)$, $(4, 6)$, and $(5, 7)$, where both states in a common pair only differ by the lookahead symbols. We can merge the states corresponding to each pair, because the marked items are the same, but now, we have to allow lookahead sets. Thus, the merging of $(2, 3)$ yields

$$\begin{aligned} 2'\colon E &\longrightarrow a.Eb, \{b, \$\} \\ E &\longrightarrow .aEb, \{b\} \\ E &\longrightarrow ., \{b\}, \end{aligned}$$

the merging of $(4, 6)$ yields

$$3'\colon E \longrightarrow aE.b, \{b, \$\},$$

the merging of $(5,7)$ yields

$$4'\colon E \longrightarrow aEb., \{b, \$\}.$$

We obtain a merged DFA with only five states, and the corresponding shift/reduce parser is given below:

	a	b	$\$$	E
1	$s2'$		$r2$	8
$2'$	$s2'$	$r2$		$3'$
$3'$		$s4'$		
$4'$		$r1$	$r1$	
8			acc	

The reader should verify that this is the $LALR(1)$-parser. The reader should also check that that the $SLR(1)$-parser is given below:

	a	b	$\$$	E
1	$s2$	$r2$	$r2$	5
2	$s2$	$r2$	$r2$	3
3		$s4$		
4		$r1$	$r1$	
5			acc	

The difference between the two parsing tables is that the $LALR(1)$-lookahead sets are sharper than the $SLR(1)$-lookahead sets. This is because the computation of the $LALR(1)$-lookahead sets uses a sharper version of FOLLOW sets. It can also be shown that if a grammar is $LALR(1)$, then the merging of states of an $LR(1)$-parser always succeeds and yields the $LALR(1)$ parser. Of course, this is a very inefficient way of producing $LALR(1)$ parsers, and much better methods exist, such as the graph method described in these notes. However, there are cases where the merging fails. Sufficient conditions for successful merging have been investigated, but there is still room for research in this area.

Chapter 8

RAM Programs, Turing Machines, and the Partial Recursive Functions

See the scanned version of this chapter found in the web page for CIS511:

`http://www.cis.upenn.edu/~jean/old511/html/tcbookpdf3a.pdf`

8.1 Partial Functions and RAM Programs

We define an abstract machine model for computing functions

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*,$$

where $\Sigma = \{a_1, \ldots, a_k\}$ s some input alphabet.

Numerical functions $f\colon \mathbb{N}^n \to \mathbb{N}$ can be viewed as functions defined over the one-letter alphabet $\{a_1\}$, using the bijection $m \mapsto a_1^m$.

Let us recall the definition of a partial function.

Definition 8.1. A binary relation $R \subseteq A \times B$ between two sets A and B is *functional* iff, for all $x \in A$ and $y, z \in B$,

$$(x, y) \in R \quad \text{and} \quad (x, z) \in R \quad \text{implies that} \quad y = z.$$

A *partial function* is a triple $f = \langle A, G, B \rangle$, where A and B are arbitrary sets (possibly empty) and G is a functional relation (possibly empty) between A and B, called the *graph* of f.

Hence, a partial function is a functional relation such that every argument has at most one image under f.

The graph of a function f is denoted as $graph(f)$. When no confusion can arise, a function f and its graph are usually identified.

A partial function $f = \langle A, G, B \rangle$ is often denoted as $f \colon A \to B$.

The *domain $dom(f)$* of a partial function $f = \langle A, G, B \rangle$ is the set

$$dom(f) = \{x \in A \mid \exists y \in B,\ (x, y) \in G\}.$$

For every element $x \in dom(f)$, the unique element $y \in B$ such that $(x, y) \in graph(f)$ is denoted as $f(x)$. We say that *$f(x)$ converges*, also denoted as $f(x) \downarrow$.

If $x \in A$ and $x \notin dom(f)$, we say that *$f(x)$ diverges*, also denoted as $f(x) \uparrow$.

Intuitively, if a function is partial, it does not return any output for any input not in its domain. This corresponds to an infinite computation.

A partial function $f \colon A \to B$ is a *total function* iff $dom(f) = A$. It is customary to call a total function simply a function.

We now define a model of computation know as the *RAM programs*, or *Post machines*.

RAM programs are written in a sort of assembly language involving simple instructions manipulating strings stored into registers.

Every RAM program uses a fixed and finite number of *registers* denoted as $R1, \ldots, Rp$, with no limitation on the size of strings held in the registers.

RAM programs can be defined either in flowchart form or in linear form. Since the linear form is more convenient for coding purposes, we present RAM programs in linear form.

A RAM program P (in linear form) consists of a finite sequence of *instructions* using a finite number of registers $R1, \ldots, Rp$.

Instructions may optionally be labeled with line numbers denoted as $N1, \ldots, Nq$.

It is neither mandatory to label all instructions, nor to use distinct line numbers!

Thus, the same line number can be used in more than one line. As we will see later on, this makes it easier to concatenate two different programs without performing a renumbering of line numbers.

Every instruction has four fields, not necessarily all used. The main field is the **op-code**. Here is an example of a RAM program to concatenate two strings x_1 and x_2.

	$R3$	$\leftarrow$	$R1$
	$R4$	$\leftarrow$	$R2$
$N0$	$R4$	$\mathtt{jmp}_a$	$N1b$
	$R4$	$\mathtt{jmp}_b$	$N2b$
		`jmp`	$N3b$
$N1$		$\mathtt{add}_a$	$R3$
		`tail`	$R4$
		`jmp`	$N0a$
$N2$		$\mathtt{add}_b$	$R3$
		`tail`	$R4$
		`jmp`	$N0a$
$N3$	$R1$	$\leftarrow$	$R3$
		`continue`	

Definition 8.2. *RAM programs* are constructed from seven types of *instructions* shown below:

(1_j)	N		$\mathtt{add}_j$	Y
(2)	N		`tail`	Y
(3)	N		`clr`	Y
(4)	N	Y	$\leftarrow$	X
$(5a)$	N		`jmp`	$N1a$
$(5b)$	N		`jmp`	$N1b$
(6_ja)	N	Y	$\mathtt{jmp}_j$	$N1a$
(6_jb)	N	Y	$\mathtt{jmp}_j$	$N1b$
(7)	N		`continue`	

1. An instruction of type (1_j) concatenates the letter a_j to the right of the string held by register Y $(1 \leq j \leq k)$. The effect is the assignment

$$Y := Ya_j.$$

2. An instruction of type (2) deletes the leftmost letter of the string held by the register Y. This corresponds to the function *tail*, defined such that

$$\begin{aligned} tail(\epsilon) &= \epsilon, \\ tail(a_j u) &= u. \end{aligned}$$

The effect is the assignment

$$Y := tail(Y).$$

3. An instruction of type (3) clears register Y, i.e., sets its value to the empty string ϵ. The effect is the assignment

$$Y := \epsilon.$$

4. An instruction of type (4) assigns the value of register X to register Y. The effect is the assignment

$$Y := X.$$

5. An instruction of type (5a) or (5b) is an unconditional jump.

   The effect of (5a) is to jump to the closest line number $N1$ occurring above the instruction being executed, and the effect of (5b) is to jump to the closest line number $N1$ occurring below the instruction being executed.

6. An instruction of type $(6_j a)$ or $(6_j b)$ is a conditional jump. Let *head* be the function defined as follows:

$$\begin{aligned} head(\epsilon) &= \epsilon, \\ head(a_j u) &= a_j. \end{aligned}$$

   The effect of $(6_j a)$ is to jump to the closest line number $N1$ occurring above the instruction being executed iff $head(Y) = a_j$, else to execute the next instruction (the one immediately following the instruction being executed).

   The effect of $(6_j b)$ is to jump to the closest line number $N1$ occurring below the instruction being executed iff $head(Y) = a_j$, else to execute the next instruction.

   When computing over $\mathbb{N}$, instructions of type $(6_j b)$ jump to the closest $N1$ above or below iff Y is nonnull.

7. An instruction of type (7) is a no-op, i.e., the registers are unaffected. If there is a next instruction, then it is executed, else, the program stops.

Obviously, a program is syntactically correct only if certain conditions hold.

Definition 8.3. A *RAM program* P is a finite sequence of instructions as in Definition 8.2, and satisfying the following conditions:

(1) For every jump instruction (conditional or not), the line number to be jumped to must exist in P.

(2) The last instruction of a RAM program is a `continue`.

The reason for allowing multiple occurences of line numbers is to make it easier to concatenate programs without having to perform a renaming of line numbers.

The technical choice of jumping to the closest address $N1$ above or below comes from the fact that it is easy to search up or down using primitive recursion, as we will see later on.

For the purpose of computing a function $f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*$ using a RAM program P, we assume that P has at least n registers called *input registers*, and that these registers $R1, \ldots, Rn$ are initialized with the input values of the function f.

We also assume that the output is returned in register $R1$.

The following RAM program concatenates two strings x_1 and x_2 held in registers $R1$ and $R2$.

$$
\begin{array}{llcl}
 & R3 & \leftarrow & R1 \\
 & R4 & \leftarrow & R2 \\
N0 & R4 & \mathtt{jmp}_a & N1b \\
 & R4 & \mathtt{jmp}_b & N2b \\
 & & \mathtt{jmp} & N3b \\
N1 & & \mathtt{add}_a & R3 \\
 & & \mathtt{tail} & R4 \\
 & & \mathtt{jmp} & N0a \\
N2 & & \mathtt{add}_b & R3 \\
 & & \mathtt{tail} & R4 \\
 & & \mathtt{jmp} & N0a \\
N3 & R1 & \leftarrow & R3 \\
 & & \mathtt{continue} &
\end{array}
$$

Since $\Sigma = \{a, b\}$, for more clarity, we wrote $\mathtt{jmp}_a$ instead of $\mathtt{jmp}_1$, $\mathtt{jmp}_b$ instead of $\mathtt{jmp}_2$, $\mathtt{add}_a$ instead of $\mathtt{add}_1$, and $\mathtt{add}_b$ instead of $\mathtt{add}_2$.

Definition 8.4. A RAM program P *computes the partial function* $\varphi\colon (\Sigma^*)^n \to \Sigma^*$ if the following conditions hold: For every input $(x_1, \ldots, x_n) \in (\Sigma^*)^n$, having initialized the input registers $R1, \ldots, Rn$ with $x_1, \ldots, x_n$, the program eventually halts iff $\varphi(x_1, \ldots, x_n)$ converges, and if and when P halts, the value of $R1$ is equal to $\varphi(x_1, \ldots, x_n)$. A partial function φ is *RAM-computable* iff it is computed by some RAM program.

For example, the following program computes the *erase function* E defined such that

$$E(u) = \epsilon$$

for all $u \in \Sigma^*$:

```
clr       R1
continue
```

The following program computes the *jth successor function* S_j defined such that

$$S_j(u) = ua_j$$

for all $u \in \Sigma^*$:

```
add_j     R1
continue
```

The following program (with n input variables) computes the *projection function* P_i^n defined such that

$$P_i^n(u_1, \ldots, u_n) = u_i,$$

where $n \geq 1$, and $1 \leq i \leq n$:

```
R1 ←       Ri
   continue
```

Note that P_1^1 is the identity function.

Having a programming language, we would like to know how powerful it is, that is, we would like to know what kind of functions are RAM-computable.

At first glance, RAM programs don't do much, but this is not so. Indeed, we will see shortly that the class of RAM-computable functions is quite extensive.

One way of getting new programs from previous ones is via composition. Another one is by primitive recursion.

We will investigate these constructions after introducing another model of computation, *Turing machines*.

Remarkably, the classes of (partial) functions computed by RAM programs and by Turing machines are identical.

This is the class of *partial computable functions*, also called *partial recursive functions*, a term which is now considered old-fashion.

This class can be given several other definitions. We will present the definition of the so-called *μ-recursive functions* (due to Kleene).

The following proposition will be needed to simplify the encoding of RAM programs as numbers.

Proposition 8.1. *Every RAM program can be converted to an equivalent program only using the following type of instructions:*

(1_j)	N		$\mathtt{add}_j$	Y
(2)	N		`tail`	Y
(6_ja)	N	Y	$\mathtt{jmp}_j$	$N1a$
(6_jb)	N	Y	$\mathtt{jmp}_j$	$N1b$
(7)	N		`continue`	

The proof is fairly simple. For example, instructions of the form

$$Ri \leftarrow Rj$$

can be eliminated by transferring the contents of Rj into an auxiliary register Rk, and then by transferring the contents of Rk into Ri and Rj.

8.2 Definition of a Turing Machine

We define a Turing machine model for computing functions

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*,$$

where $\Sigma = \{a_1, \ldots, a_N\}$ is some input alphabet. We only consider deterministic Turing machines.

A Turing machine also uses a *tape alphabet* Γ such that $\Sigma \subseteq \Gamma$. The tape alphabet contains some special symbol $B \notin \Sigma$, the *blank*.

In this model, a Turing machine uses a single tape. This tape can be viewed as a string over Γ. The tape is both an input tape and a storage mechanism.

Symbols on the tape can be overwritten, and the tape can grow either on the left or on the right. There is a read/write head pointing to some symbol on the tape.

Definition 8.5. A (deterministic) *Turing machine* (or *TM*) M is a sextuple $M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0)$, where

- K is a finite set of *states*;
- Σ is a finite *input alphabet*;
- Γ is a finite *tape alphabet*, s.t. $\Sigma \subseteq \Gamma$, $K \cap \Gamma = \emptyset$, and with blank $B \notin \Sigma$;
- $q_0 \in K$ is the *start state* (or *initial state*);
- δ is the *transition function*, a (finite) set of quintuples

  $$\delta \subseteq K \times \Gamma \times \Gamma \times \{L, R\} \times K,$$

  such that for all $(p, a) \in K \times \Gamma$, there is at most one triple $(b, m, q) \in \Gamma \times \{L, R\} \times K$ such that $(p, a, b, m, q) \in \delta$.

A quintuple $(p, a, b, m, q) \in \delta$ is called an *instruction*. It is also denoted as

$$p, a \rightarrow b, m, q.$$

The effect of an instruction is to switch from state p to state q, overwrite the symbol currently scanned a with b, and move the read/write head either left or right, according to m.

Here is an example of a Turing machine.

$K = \{q_0, q_1, q_2, q_3\}$;

$\Sigma = \{a, b\}$;

$\Gamma = \{a, b, B\}$;

The instructions in δ are:

$$\begin{aligned}
q_0, B &\to B, R, q_3, \\
q_0, a &\to b, R, q_1, \\
q_0, b &\to a, R, q_1, \\
q_1, a &\to b, R, q_1, \\
q_1, b &\to a, R, q_1, \\
q_1, B &\to B, L, q_2, \\
q_2, a &\to a, L, q_2, \\
q_2, b &\to b, L, q_2, \\
q_2, B &\to B, R, q_3.
\end{aligned}$$

8.3 Computations of Turing Machines

To explain how a Turing machine works, we describe its action on *Instantaneous descriptions*. We take advantage of the fact that $K \cap \Gamma = \emptyset$ to define instantaneous descriptions.

Definition 8.6. Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

an *instantaneous description* (for short an *ID*) is a (nonempty) string in $\Gamma^* K \Gamma^+$, that is, a string of the form

$$upav,$$

where $u, v \in \Gamma^*$, $p \in K$, and $a \in \Gamma$.

The intuition is that an ID $upav$ describes a snapshot of a TM in the current state p, whose tape contains the string uav, and with the read/write head pointing to the symbol a.

Thus, in $upav$, the state p is just to the left of the symbol presently scanned by the read/write head.

We explain how a TM works by showing how it acts on ID's.

Definition 8.7. Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

the *yield relation (or compute relation)* $\vdash$ is a binary relation defined on the set of ID's as follows. For any two ID's ID_1 and ID_2, we have $ID_1 \vdash ID_2$ iff either

(1) $(p, a, b, R, q) \in \delta$, and either

(a) $ID_1 = upacv$, $c \in \Gamma$, and $ID_2 = ubqcv$, or

(b) $ID_1 = upa$ and $ID_2 = ubqB$;

or

(2) $(p, a, b, L, q) \in \delta$, and either

(a) $ID_1 = ucpav$, $c \in \Gamma$, and $ID_2 = uqcbv$, or

(b) $ID_1 = pav$ and $ID_2 = qBbv$.

Note how the tape is extended by one blank after the rightmost symbol in case (1)(b), and by one blank before the leftmost symbol in case (2)(b).

As usual, we let $\vdash^+$ denote the transitive closure of $\vdash$, and we let $\vdash^*$ denote the reflexive and transitive closure of $\vdash$.

We can now explain how a Turing machine computes a partial function

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*.$$

Since we allow functions taking $n \geq 1$ input strings, we assume that Γ contains the special delimiter **,** not in Σ, used to separate the various input strings.

It is convenient to assume that a Turing machine "cleans up" its tape when it halts, before returning its output. For this, we will define proper ID's.

Definition 8.8. Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

where Γ contains some delimiter **,** not in Σ in addition to the blank B, a *starting ID* is of the form

$$q_0 w_1, w_2, \ldots, w_n$$

where $w_1, \ldots, w_n \in \Sigma^*$ and $n \geq 2$, or $q_0 w$ with $w \in \Sigma^+$, or $q_0 B$.

A *blocking (or halting) ID* is an ID $upav$ such that there are no instructions $(p, a, b, m, q) \in \delta$ for any $(b, m, q) \in \Gamma \times \{L, R\} \times K$.

A *proper ID* is a halting ID of the form

$$B^k p w B^l,$$

where $w \in \Sigma^*$, and $k, l \geq 0$ (with $l \geq 1$ when $w = \epsilon$).

Computation sequences are defined as follows.

Definition 8.9. Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

a *computation sequence (or computation)* is a finite or infinite sequence of ID's

$$ID_0, ID_1, \ldots, ID_i, ID_{i+1}, \ldots,$$

such that $ID_i \vdash ID_{i+1}$ for all $i \geq 0$.

A computation sequence *halts* iff it is a finite sequence of ID's, so that

$$ID_0 \vdash^* ID_n,$$

and ID_n is a halting ID.

A computation sequence *diverges* if it is an infinite sequence of ID's.

We now explain how a Turing machine computes a partial function.

Definition 8.10. A Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0)$$

computes the partial function

$$f \colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*$$

iff the following conditions hold:

(1) For every $w_1, \ldots, w_n \in \Sigma^*$, given the starting ID

$$ID_0 = q_0 w_1 {,} w_2 {,} \ldots {,} w_n$$

or $q_0 w$ with $w \in \Sigma^+$, or $q_0 B$, the computation sequence of M from ID_0 halts in a proper ID
iff $f(w_1, \ldots, w_n)$ is defined.

(2) If $f(w_1, \ldots, w_n)$ is defined, then M halts in a proper ID of the form

$$ID_n = B^k p f(w_1, \ldots, w_n) B^h,$$

which means that it computes the right value.

A function f (over Σ^*) is *Turing computable* iff it is computed by some Turing machine M.

Note that by (1), the TM M may halt in an improper ID, in which case $f(w_1, \ldots, w_n)$ must be undefined. This corresponds to the fact that we only accept to retrieve the output of a computation if the TM has cleaned up its tape, i.e., produced a proper ID. In particular, intermediate calculations have to be erased before halting.

Example.

$K = \{q_0, q_1, q_2, q_3\}$;

$\Sigma = \{a, b\}$;

$\Gamma = \{a, b, B\}$;

The instructions in δ are:

$$
\begin{aligned}
q_0, B &\to B, R, q_3, \\
q_0, a &\to b, R, q_1, \\
q_0, b &\to a, R, q_1, \\
q_1, a &\to b, R, q_1, \\
q_1, b &\to a, R, q_1, \\
q_1, B &\to B, L, q_2, \\
q_2, a &\to a, L, q_2, \\
q_2, b &\to b, L, q_2, \\
q_2, B &\to B, R, q_3.
\end{aligned}
$$

The reader can easily verify that this machine exchanges the a's and b's in a string. For example, on input $w = aaababb$, the output is $bbbabaa$.

8.4 RAM-computable functions are Turing-computable

Turing machines can simulate RAM programs, and as a result, we have the following Theorem.

Theorem 8.2. *Every RAM-computable function is Turing-computable. Furthermore, given a RAM program P, we can effectively construct a Turing machine M computing the same function.*

The idea of the proof is to represent the contents of the registers $R1, \ldots Rp$ on the Turing machine tape by the string

$$\#r1\#r2\# \cdots \#rp\#,$$

Where # is a special marker and ri represents the string held by Ri, We also use Proposition 8.1 to reduce the number of instructions to be dealt with.

The Turing machine M is built of blocks, each block simulating the effect of some instruction of the program P. The details are a bit tedious, and can be found in the notes or in Machtey and Young.

8.5 Turing-computable functions are RAM-computable

RAM programs can also simulate Turing machines.

Theorem 8.3. *Every Turing-computable function is RAM-computable. Furthermore, given a Turing machine M, one can effectively construct a RAM program P computing the same function.*

The idea of the proof is to design a RAM program containing an encoding of the current ID of the Turing machine M in register $R1$, and to use other registers $R2, R3$ to simulate the effect of executing an instruction of M by updating the ID of M in $R1$.

The details are tedious and can be found in the notes.

Another proof can be obtained by proving that the class of Turing computable functions coincides with the class of *partial computable functions* (formerly called *partial recursive functions*).

Indeed, it turns out that both RAM programs and Turing machines compute precisely the class of partial recursive functions. For this, we need to define the *primitive recursive functions*.

Informally, a primitive recursive function is a total recursive function that can be computed using only **for** loops, that is, loops in which the number of iterations is fixed (unlike a **while** loop).

A formal definition of the primitive functions is given in Section 8.7.

Definition 8.11. Let $\Sigma = \{a_1, \ldots, a_N\}$. The class of *partial computable functions* also called *partial recursive functions* is the class of partial functions (over Σ^*) that can be computed by RAM programs (or equivalently by Turing machines).

The class of *computable functions* also called *recursive functions* is the subset of the class of partial computable functions consisting of functions defined for every input (i.e., total functions).

We can also deal with languages.

8.6 Computably Enumerable Languages and Computable Languages

We define the computably enumerable languages, also called listable languages, and the computable languages.

The old-fashion terminology for computably enumerable languages is recursively enumerable languages, and for computable languages is recursive languages.

We assume that the TM's under consideration have a tape alphabet containing the special symbols 0 and 1.

Definition 8.12. Let $\Sigma = \{a_1, \ldots, a_N\}$. A language $L \subseteq \Sigma^*$ is *(Turing) computably enumerable (for short, a c.e. set)*, or *(Turing) listable* (or *recursively enumerable (for short, a r.e. set)*) iff there is some TM M such that for every $w \in L$, M halts in a proper ID with the output 1, and for every $w \notin L$, either M halts in a proper ID with the output 0, or it runs forever.

A language $L \subseteq \Sigma^*$ is *(Turing) computable* (or *recursive*) iff there is some TM M such that for every $w \in L$, M halts in a proper ID with the output 1, and for every $w \notin L$, M halts in a proper ID with the output 0.

Thus, given a computably enumerable language L, for some $w \notin L$, it is possible that a TM accepting L runs forever on input w. On the other hand, for a computable (recursive) language L, a TM accepting L always halts in a proper ID.

When dealing with languages, it is often useful to consider *nondeterministic Turing machines*. Such machines are defined just like deterministic Turing machines, except that their transition function δ is just a (finite) set of quintuples

$$\delta \subseteq K \times \Gamma \times \Gamma \times \{L, R\} \times K,$$

with no particular extra condition.

It can be shown that every nondeterministic Turing machine can be simulated by a deterministic Turing machine, and thus, nondeterministic Turing machines also accept the class of c.e. sets.

It can be shown that a computably enumerable language is the range of some computable (recursive) function. It can also be shown that a language L is computable (recursive) iff both L and its complement are computably enumerable. There are computably enumerable languages that are not computable (recursive).

Turing machines were invented by Turing around 1935. The primitive recursive functions were known to Hilbert circa 1890. Gödel formalized their definition in 1929. The partial recursive functions were defined by Kleene around 1934.

Church also introduced the λ-calculus as a model of computation around 1934. Other models: Post systems, Markov systems. The equivalence of the various models of computation was shown around 1935/36. RAM programs were only defined around 1963 (they are a slight generalization of Post system).

A further study of the partial recursive functions requires the notions of pairing functions and of universal functions (or universal Turing machines).

8.7 The Primitive Recursive Functions

The class of primitive recursive functions is defined in terms of base functions and closure operations.

Definition 8.13. Let $\Sigma = \{a_1, \ldots, a_N\}$. The *base functions* over Σ are the following functions:

(1) The *erase function E*, defined such that $E(w) = \epsilon$, for all $w \in \Sigma^*$;

(2) For every j, $1 \leq j \leq N$, the *j-successor function S_j*, defined such that $S_j(w) = wa_j$, for all $w \in \Sigma^*$;

(3) The *projection functions P_i^n*, defined such that

$$P_i^n(w_1, \ldots, w_n) = w_i,$$

for every $n \geq 1$, every i, $1 \leq i \leq n$, and for all $w_1, \ldots, w_n \in \Sigma^*$.

Note that P_1^1 is the identity function on Σ^*. Projection functions can be used to permute the arguments of another function.

A crucial closure operation is (extended) composition.

Definition 8.14. Let $\Sigma = \{a_1, \ldots, a_N\}$. For any function

$$g\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \to \Sigma^*,$$

and any m functions

$$h_i\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*,$$

the *composition of g and the h_i* is the function

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*,$$

denoted as $g \circ (h_1, \ldots, h_m)$, such that

$$f(w_1, \ldots, w_n) = g(h_1(w_1, \ldots, w_n), \ldots, h_m(w_1, \ldots, w_n)),$$

for all $w_1, \ldots, w_n \in \Sigma^*$.

As an example, $f = g \circ (P_2^2, P_1^2)$ is such that

$$f(w_1, w_2) = g(P_2^2(w_1, w_2), P_1^2(w_1, w_2)) = g(w_2, w_1).$$

Another crucial closure operation is *primitive recursion*.

Definition 8.15. Let $\Sigma = \{a_1, \ldots, a_N\}$. For any function

$$g\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m-1} \to \Sigma^*,$$

where $m \geq 2$, and any N functions

$$h_i\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m+1} \to \Sigma^*,$$

the function

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \to \Sigma^*,$$

is defined by *primitive recursion from g and $h_1, \ldots, h_N$*, if

$$\begin{aligned} f(\epsilon, w_2, \ldots, w_m) &= g(w_2, \ldots, w_m), \\ f(ua_1, w_2, \ldots, w_m) &= h_1(u, f(u, w_2, \ldots, w_m), w_2, \ldots, w_m), \\ \cdots &= \cdots \\ f(ua_N, w_2, \ldots, w_m) &= h_N(u, f(u, w_2, \ldots, w_m), w_2, \ldots, w_m), \end{aligned}$$

for all $u, w_2, \ldots, w_m \in \Sigma^*$.

When $m = 1$, for some fixed $w \in \Sigma^*$, we have

$$\begin{aligned} f(\epsilon) &= w, \\ f(ua_1) &= h_1(u, f(u)), \\ \cdots &= \cdots \\ f(ua_N) &= h_N(u, f(u)), \end{aligned}$$

for all $u \in \Sigma^*$.

For numerical functions (i.e., when $\Sigma = \{a_1\}$), the scheme of primitive recursion is simpler:

$$\begin{aligned} f(0, x_2, \ldots, x_m) &= g(x_2, \ldots, x_m), \\ f(x+1, x_2, \ldots, x_m) &= h_1(x, f(x, x_2, \ldots, x_m), x_2, \ldots, x_m), \end{aligned}$$

for all $x, x_2, \ldots, x_m \in \mathbb{N}$.

The *successor function* S is the function

$$S(x) = x + 1.$$

Addition, multiplication, exponentiation, and super-exponentiation, can be defined by primitive recursion as follows (being a bit loose, we should use some projections ...):

$$\begin{aligned}
add(0,n) &= P_1^1(n) = n, \\
add(m+1,n) &= S \circ P_2^3(m, add(m,n), n) \\
&= S(add(m,n)) \\
mult(0,n) &= E(n) = 0, \\
mult(m+1,n) &= add \circ (P_2^3, P_3^3)(m, mult(m,n), n) \\
&= add(mult(m,n), n), \\
rexp(0,n) &= S \circ E(n) = 1, \\
rexp(m+1,n) &= mult(rexp(m,n), n), \\
exp(m,n) &= rexp \circ (P_2^2, P_1^2)(m,n), \\
supexp(0,n) &= 1, \\
supexp(m+1,n) &= exp(n, supexp(m,n)).
\end{aligned}$$

We usually write $m+n$ for $add(m,n)$, $m*n$ or even mn for $mult(m,n)$, and m^n for $exp(m,n)$.

There is a minus operation on $\mathbb{N}$ named *monus*. This operation denoted by $\dot{-}$ is defined by

$$m \dot{-} n = \begin{cases} m-n & \text{if } m \geq n \\ 0 & \text{if } m < n. \end{cases}$$

To show that it is primitive recursive, we define the function *pred*. Let *pred* be the primitive recursive function given by

$$\begin{aligned}
pred(0) &= 0 \\
pred(m+1) &= P_1^2(m, pred(m)) = m.
\end{aligned}$$

Then *monus* is defined by

$$\begin{aligned}
monus(m,0) &= m \\
monus(m,n+1) &= pred(monus(m,n)),
\end{aligned}$$

except that the above is not a legal primitive recursion. It is left as an exercise to give a proper primitive recursive definition of *monus*.

As an example over $\{a, b\}^*$, the following function $g\colon \Sigma^* \times \Sigma^* \to \Sigma^*$, is defined by primitive recursion:

$$\begin{aligned} g(\epsilon, v) &= P_1^1(v), \\ g(ua_i, v) &= S_i \circ P_2^3(u, g(u, v), v), \end{aligned}$$

where $1 \leq i \leq N$. It is easily verified that $g(u, v) = vu$. Then,

$$f = g \circ (P_2^2, P_1^2)$$

computes the concatenation function, i.e. $f(u, v) = uv$.

The following functions are also primitive recursive:

$$sg(n) = \begin{cases} 1 & \text{if } n > 0 \\ 0 & \text{if } n = 0, \end{cases}$$

$$\overline{sg}(n) = \begin{cases} 0 & \text{if } n > 0 \\ 1 & \text{if } n = 0, \end{cases}$$

as well as

$$abs(m, n) = |m - m| = m \dot{-} n + n \dot{-} m,$$

and

$$eq(m, n) = \begin{cases} 1 & \text{if } m = n \\ 0 & \text{if } m \neq n. \end{cases}$$

Indeed

$$\begin{aligned} sg(0) &= 0 \\ sg(n + 1) &= S \circ E \circ P_1^2(n, sg(n)), \end{aligned}$$

$$\overline{sg}(n) = S(E(n)) \dot{-} sg(n) = 1 \dot{-} sg(n),$$

and

$$eq(m, n) = sg(|m - n|).$$

Finally, the function

$$cond(m, n, p, q) = \begin{cases} p & \text{if } m = n \\ q & \text{if } m \neq n, \end{cases}$$

is primitive recursive since

$$cond(m, n, p, q) = eq(m, n) * p + \overline{sg}(eq(m, n)) * q.$$

We can also design more general version of *cond*. For example, define $compare_{\leq}$ as

$$compare_{\leq}(m,n) = \begin{cases} 1 & \text{if } m \leq n \\ 0 & \text{if } m > n, \end{cases}$$

which is given by

$$compare_{\leq}(m,n) = 1 \dot{-} sg(m \dot{-} n).$$

Then we can define

$$cond_{\leq}(m,n,p,q) = \begin{cases} p & \text{if } m \leq n \\ q & \text{if } m > n, \end{cases}$$

with

$$cond_{\leq}(m,n,n,p) = compare_{\leq}(m,n) * p + \overline{sg}(compare_{\leq}(m,n)) * q.$$

The above allows to define functions by cases.

Definition 8.16. Let $\Sigma = \{a_1, \dots, a_N\}$. The class of *primitive recursive functions* is the smallest class of functions (over Σ^*) which contains the base functions and is closed under composition and primitive recursion.

We leave as an exercise to show that every primitive recursive function is a total function. The class of primitive recursive functions may not seem very big, but it contains all the total functions that we would ever want to compute.

Although it is rather tedious to prove, the following theorem can be shown.

Theorem 8.4. *For an alphabet* $\Sigma = \{a_1, \dots, a_N\}$, *every primitive recursive function is Turing computable.*

The best way to prove the above theorem is to use the computation model of RAM programs. Indeed, it was shown in Theorem 8.2 that every RAM program can be converted to a Turing machine.

It is also rather easy to show that the primitive recursive functions are RAM-computable.

In order to define new functions it is also useful to use predicates.

Definition 8.17. An *n-ary predicate P* (over Σ^*) is any subset of $(\Sigma^*)^n$. We write that a tuple $(x_1, \dots, x_n)$ *satisfies P* as $(x_1, \dots, x_n) \in P$ or as $P(x_1, \dots, x_n)$. The *characteristic function* of a predicate P is the function $C_P \colon (\Sigma^*)^n \to \{a_1\}^*$ defined by

$$C_p(x_1, \dots, x_n) = \begin{cases} a_1 & \text{iff } P(x_1, \dots, x_n) \\ \epsilon & \text{iff } \textbf{not } P(x_1, \dots, x_n). \end{cases}$$

A predicate P is *primitive recursive* iff its characteristic function C_P is primitive recursive.

We leave to the reader the obvious adaptation of the the notion of primitive recursive predicate to functions defined over $\mathbb{N}$. In this case, 0 plays the role of ϵ and 1 plays the role of a_1.

It is easily shown that if P and Q are primitive recursive predicates (over $(\Sigma^*)^n$), then $P \vee Q$, $P \wedge Q$ and $\neg P$ are also primitive recursive.

As an exercise, the reader may want to prove that the predicate (defined over $\mathbb{N}$):
prime(n) iff n is a prime number, is a primitive recursive predicate.

For any fixed $k \geq 1$, the function:
$\mathrm{ord}(k, n) =$ exponent of the kth prime in the prime factorization of n, is a primitive recursive function.

We can also define functions by cases.

Proposition 8.5. *If $P_1, \ldots, P_n$ are pairwise disjoint primitive recursive predicates (which means that $P_i \cap P_j = \emptyset$ for all $i \neq j$) and $f_1, \ldots, f_{n+1}$ are primitive recursive functions, the function g defined below is also primitive recursive:*

$$g(\overline{x}) = \begin{cases} f_1(\overline{x}) & \text{iff } P_1(\overline{x}) \\ \vdots & \\ f_n(\overline{x}) & \text{iff } P_n(\overline{x}) \\ f_{n+1}(\overline{x}) & \text{otherwise.} \end{cases}$$

(writing $\overline{x}$ for $(x_1, \ldots, x_n)$.)

It is also useful to have bounded quantification and bounded minimization.

Definition 8.18. If P is an $(n+1)$-ary predicate, then the *bounded existential predicate* $\exists y/xP(y, \overline{z})$ holds iff some prefix y of x makes $P(y, \overline{z})$ true.

The *bounded universal predicate* $\forall y/xP(y, \overline{z})$ holds iff every prefix y of x makes $P(y, \overline{z})$ true.

Proposition 8.6. *If P is an $(n+1)$-ary primitive recursive predicate, then $\exists y/xP(y, \overline{z})$ and $\forall y/xP(y, \overline{z})$ are also primitive recursive predicates.*

As an application, we can show that the equality predicate, $u = v?$, is primitive recursive.

Definition 8.19. If P is an $(n+1)$-ary predicate, then the *bounded minimization of P*, $\min y/x\, P(y, \overline{z})$, is the function defined such that $\min y/x\, P(y, \overline{z})$ is the shortest prefix of x such that $P(y, \overline{z})$ if such a y exists, xa_1 otherwise.

The *bounded maximization of P*, $\max y/x\, P(y, \overline{z})$, is the function defined such that $\max y/x\, P(y, \overline{z})$ is the longest prefix of x such that $P(y, \overline{z})$ if such a y exists, xa_1 otherwise.

Proposition 8.7. *If P is an $(n+1)$-ary primitive recursive predicate, then* $\min y/x\, P(y, \overline{z})$ *and* $\max y/x\, P(y, \overline{z})$ *are primitive recursive functions.*

So far, the primitive recursive functions do not yield all the Turing-computable functions. In order to get a larger class of functions, we need the closure operation known as minimization.

8.8 The Partial Computable Functions

Minimization can be viewed as an abstract version of a while loop.

Let $\Sigma = \{a_1, \ldots, a_N\}$. For any function

$$g\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m+1} \to \Sigma^*,$$

where $m \geq 0$, for every j, $1 \leq j \leq N$, the function

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \to \Sigma^*$$

looks for the shortest string u over a_j^* (for a given j) such that

$$g(u, w_1, \ldots, w_m) = \epsilon :$$

$u := \epsilon$;
while $g(u, w_1, \ldots, w_m) \neq \epsilon$ **do**
$u := ua_j$;
endwhile
let $f(w_1, \ldots, w_m) = u$

The operation of minimization (sometimes called minimalization) is defined as follows.

Definition 8.20. Let $\Sigma = \{a_1, \ldots, a_N\}$. For any function

$$g\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m+1} \to \Sigma^*,$$

where $m \geq 0$, for every j, $1 \leq j \leq N$, the function

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \to \Sigma^*,$$

is defined by *minimization over $\{a_j\}^*$ from g*, if the following conditions hold for all $w_1, \ldots, w_m \in \Sigma^*$:

(1) $f(w_1, \ldots, w_m)$ is defined iff there is some $n \geq 0$ such that $g(a_j^p, w_1, \ldots, w_m)$ is defined for all p, $0 \leq p \leq n$, and
$$g(a_j^n, w_1, \ldots, w_m) = \epsilon.$$

(2) When $f(w_1, \ldots, w_m)$ is defined,
$$f(w_1, \ldots, w_m) = a_j^n,$$
where n is such that
$$g(a_j^n, w_1, \ldots, w_m) = \epsilon$$
and
$$g(a_j^p, w_1, \ldots, w_m) \neq \epsilon$$
for every p, $0 \leq p \leq n - 1$.

We also write
$$f(w_1, \ldots, w_m) = min_j u[g(u, w_1, \ldots, w_m) = \epsilon].$$

Note: When $f(w_1, \ldots, w_m)$ is defined,
$$f(w_1, \ldots, w_m) = a_j^n,$$
where n is the smallest integer such that condition (1) holds. It is very important to require that all the values $g(a_j^p, w_1, \ldots, w_m)$ be defined for all p, $0 \leq p \leq n$, when defining $f(w_1, \ldots, w_m)$. Failure to do so allows non-computable functions.

Remark: Kleene used the *μ-notation*:
$$f(w_1, \ldots, w_m) = \mu_j u[g(u, w_1, \ldots, w_m) = \epsilon],$$
actually, its numerical form:
$$f(x_1, \ldots, x_m) = \mu x[g(x, x_1, \ldots, x_m) = 0].$$

The class of partial computable functions is defined as follows.

Definition 8.21. Let $\Sigma = \{a_1, \ldots, a_N\}$. The class of *partial computable functions* also called *partial recursive functions* is the smallest class of partial functions (over Σ^*) which contains the base functions and is closed under composition, primitive recursion, and minimization.

The class of *computable functions* also called *recursive functions* is the subset of the class of partial computable functions consisting of functions defined for every input (i.e., total functions).

One of the major results of computability theory is the following theorem.

Theorem 8.8. *For an alphabet $\Sigma = \{a_1, \ldots, a_N\}$, every partial computable function (partial recursive function) is Turing-computable. Conversely, every Turing-computable function is a partial computable function (partial recursive function). Similarly, the class of computable functions (recursive functions) is equal to the class of Turing-computable functions that halt in a proper ID for every input.*

To prove that every partial computable function is indeed Turing-computable, since by Theorem 8.2, every RAM program can be converted to a Turing machine, the simplest thing to do is to show that every partial computable function is RAM-computable.

For the converse, one can show that given a Turing machine, there is a primitive recursive function describing how to go from one ID to the next. Then, minimization is used to guess whether a computation halts. The proof shows that every partial computable function needs minimization at most once. The characterization of the computable functions in terms of TM's follows easily.

There are computable functions (recursive functions) that are not primitive recursive. Such an example is given by Ackermann's function.

Ackermann's function.

This is a function $A\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ which is defined by the following recursive clauses:

$$\begin{aligned} A(0,\, y) &= y + 1, \\ A(x+1,\, 0) &= A(x,\, 1), \\ A(x+1,\, y+1) &= A(x,\, A(x+1, y)). \end{aligned}$$

It turns out that A is a computable function which is **not** primitive recursive.

It can be shown that:

$$\begin{aligned} A(0,\, x) &= x + 1, \\ A(1,\, x) &= x + 2, \\ A(2,\, x) &= 2x + 3, \\ A(3,\, x) &= 2^{x+3} - 3, \end{aligned}$$

and

$$A(4, x) = \left.2^{2^{\cdot^{\cdot^{2^{16}}}}}\right\}^{x} - 3,$$

with $A(4, 0) = 16 - 3 = 13$.

For example

$$A(4, 1) = 2^{16} - 3, \quad A(4, 2) = 2^{2^{16}} - 3.$$

Actually, it is not so obvious that A is a total function. This can be shown by induction, using the lexicographic ordering $\preceq$ on $\mathbb{N} \times \mathbb{N}$, which is defined as follows:

$$\begin{aligned}
&(m, n) \preceq (m', n') \quad \text{iff either} \\
&m = m' \text{ and } n = n', \text{ or} \\
&m < m', \text{ or} \\
&m = m' \text{ and } n < n'.
\end{aligned}$$

We write $(m, n) \prec (m', n')$ when $(m, n) \preceq (m', n')$ and $(m, n) \neq (m', n')$.

We prove that $A(m, n)$ is defined for all $(m, n) \in \mathbb{N} \times \mathbb{N}$ by complete induction over the lexicographic ordering on $\mathbb{N} \times \mathbb{N}$.

In the base case, $(m, n) = (0, 0)$, and since $A(0, n) = n + 1$, we have $A(0, 0) = 1$, and $A(0, 0)$ is defined.

For $(m, n) \neq (0, 0)$, the induction hypothesis is that $A(m', n')$ is defined for all $(m', n') \prec (m, n)$. We need to conclude that $A(m, n)$ is defined.

If $m = 0$, since $A(0, n) = n + 1$, $A(0, n)$ is defined.

If $m \neq 0$ and $n = 0$, since

$$(m - 1, 1) \prec (m, 0),$$

by the induction hypothesis, $A(m - 1, 1)$ is defined, but $A(m, 0) = A(m - 1, 1)$, and thus $A(m, 0)$ is defined.

If $m \neq 0$ and $n \neq 0$, since

$$(m, n - 1) \prec (m, n),$$

by the induction hypothesis, $A(m, n - 1)$ is defined. Since

$$(m - 1, A(m, n - 1)) \prec (m, n),$$

by the induction hypothesis, $A(m - 1, A(m, n - 1))$ is defined. But $A(m, n) = A(m - 1, A(m, n - 1))$, and thus $A(m, n)$ is defined.

Thus, $A(m, n)$ is defined for all $(m, n) \in \mathbb{N} \times \mathbb{N}$. It is possible to show that A is a recursive function, although the quickest way to prove it requires some fancy machinery (the recursion theorem).

Proving that A is not primitive recursive is harder.

The following proposition shows that restricting ourselves to total functions is too limiting.

Let $\mathcal{F}$ be any set of total functions that contains the base functions and is closed under composition and primitive recursion (and thus, $\mathcal{F}$ contains all the primitive recursive functions).

Definition 8.22. We say that a function $f\colon \Sigma^* \times \Sigma^* \to \Sigma^*$ is *universal* for the one-argument functions in $\mathcal{F}$ iff for every function $g\colon \Sigma^* \to \Sigma^*$ in $\mathcal{F}$, there is some $n \in \mathbb{N}$ such that

$$f(a_1^n, u) = g(u)$$

for all $u \in \Sigma^*$.

Proposition 8.9. *For any countable set $\mathcal{F}$ of total functions containing the base functions and closed under composition and primitive recursion, if f is a universal function for the functions $g\colon \Sigma^* \to \Sigma^*$ in $\mathcal{F}$, then $f \notin \mathcal{F}$.*

Proof. Assume that the universal function f is in $\mathcal{F}$. Let g be the function such that

$$g(u) = f(a_1^{|u|}, u)a_1$$

for all $u \in \Sigma^*$. We claim that $g \in \mathcal{F}$. It it enough to prove that the function h such that

$$h(u) = a_1^{|u|}$$

is primitive recursive, which is easily shown.

Then, because f is universal, there is some m such that

$$g(u) = f(a_1^m, u)$$

for all $u \in \Sigma^*$. Letting $u = a_1^m$, we get

$$g(a_1^m) = f(a_1^m, a_1^m) = f(a_1^m, a_1^m)a_1,$$

a contradiction. ∎

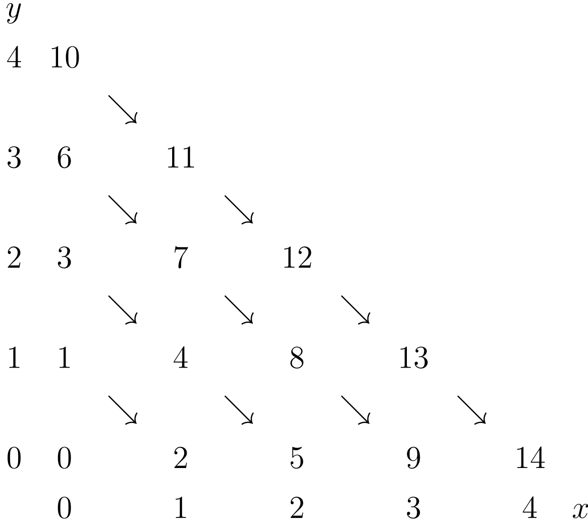
Thus, either a universal function for $\mathcal{F}$ is partial, or it is not in $\mathcal{F}$.

Chapter 9

Universal RAM Programs and Undecidability of the Halting Problem

9.1 Pairing Functions

Pairing functions are used to encode pairs of integers into single integers, or more generally, finite sequences of integers into single integers. We begin by exhibiting a bijective pairing function $J\colon \mathbb{N}^2 \to \mathbb{N}$. The function J has the graph partially showed below:



The function J corresponds to a certain way of enumerating pairs of integers (x, y). Note that the value of $x + y$ is constant along each descending diagonal, and consequently, we have

$$\begin{aligned} J(x, y) &= 1 + 2 + \cdots + (x + y) + x, \\ &= ((x + y)(x + y + 1) + 2x)/2, \\ &= ((x + y)^2 + 3x + y)/2, \end{aligned}$$

that is,

$$J(x, y) = ((x + y)^2 + 3x + y)/2.$$

For example, $J(0, 3) = 6,\ J(1, 2) = 7,\ J(2, 2) = 12,\ J(3, 1) = 13,\ J(4, 0) = 14.$

Let $K\colon \mathbb{N} \to \mathbb{N}$ and $L\colon \mathbb{N} \to \mathbb{N}$ be the projection functions onto the axes, that is, the unique functions such that

$$K(J(a, b)) = a \quad \text{and} \quad L(J(a, b)) = b,$$

for all $a, b \in \mathbb{N}$. For example, $K(11) = 1$, and $L(11) = 3$; $K(12) = 2$, and $L(12) = 2$; $K(13) = 3$ and $L(13) = 1$.

The functions J, K, L are called *Cantor's pairing functions*. They were used by Cantor to prove that the set $\mathbb{Q}$ of rational numbers is countable.

Clearly, J is primitive recursive, since it is given by a polynomial. It is not hard to prove that J is injective and surjective, and that it is strictly monotonic in each argument, which means that for all $x, x', y, y' \in \mathbb{N}$, if $x < x'$ then $J(x, y) < J(x', y)$, and if $y < y'$ then $J(x, y) < J(x, y')$.

The projection functions can be computed explicitly, although this is a bit tricky. We only need to observe that by monotonicity of J,

$$x \leq J(x, y) \quad \text{and} \quad y \leq J(x, y),$$

and thus,

$$K(z) = \min(x \leq z)(\exists y \leq z)[J(x, y) = z],$$

and

$$L(z) = \min(y \leq z)(\exists x \leq z)[J(x, y) = z].$$

Therefore, K and L are primitive recursive. It can be verified that $J(K(z), L(z)) = z$, for all $z \in \mathbb{N}$.

More explicit formulae can be given for K and L. If we define

$$\begin{aligned} Q_1(z) &= \lfloor (\lfloor \sqrt{8z + 1} \rfloor + 1)/2 \rfloor - 1 \\ Q_2(z) &= 2z - (Q_1(z))^2, \end{aligned}$$

then it can be shown that

$$\begin{aligned} K(z) &= \frac{1}{2}(Q_2(z) - Q_1(z)) \\ L(z) &= Q_1(z) - \frac{1}{2}(Q_2(z) - Q_1(z)). \end{aligned}$$

In the above formula, the function $m \mapsto \lfloor \sqrt{m} \rfloor$ yields the largest integer s such that $s^2 \leq m$. It can be computed by a RAM program.

The pairing function $J(x, y)$ is also denoted as $\langle x, y\rangle$, and K and L are also denoted as Π_1 and Π_2.

By induction, we can define bijections between $\mathbb{N}^n$ and $\mathbb{N}$ for all $n \geq 1$. We let $\langle z\rangle_1 = z$,

$$\langle x_1, x_2\rangle_2 = \langle x_1, x_2\rangle,$$

and

$$\langle x_1, \ldots, x_n, x_{n+1}\rangle_{n+1} = \langle x_1, \ldots, x_{n-1}, \langle x_n, x_{n+1}\rangle\rangle_n.$$

For example.

$$\begin{aligned}
\langle x_1, x_2, x_3\rangle_3 &= \langle x_1, \langle x_2, x_3\rangle\rangle_2 \\
&= \langle x_1, \langle x_2, x_3\rangle\rangle \\
\langle x_1, x_2, x_3, x_4\rangle_4 &= \langle x_1, x_2, \langle x_3, x_4\rangle\rangle_3 \\
&= \langle x_1, \langle x_2, \langle x_3, x_4\rangle\rangle\rangle \\
\langle x_1, x_2, x_3, x_4, x_5\rangle_5 &= \langle x_1, x_2, x_3, \langle x_4, x_5\rangle\rangle_4 \\
&= \langle x_1, \langle x_2, \langle x_3, \langle x_4, x_5\rangle\rangle\rangle\rangle.
\end{aligned}$$

It can be shown by induction on n that

$$\langle x_1, \ldots, x_n, x_{n+1}\rangle_{n+1} = \langle x_1, \langle x_2, \ldots, x_{n+1}\rangle_n\rangle.$$

The function $\langle -, \ldots, -\rangle_n \colon \mathbb{N}^n \to \mathbb{N}$ is called an *extended pairing function*.

Observe that if $z = \langle x_1, \ldots, x_n\rangle_n$, then $x_1 = \Pi_1(z)$, $x_2 = \Pi_1(\Pi_2(z))$, $x_3 = \Pi_1(\Pi_2(\Pi_2(z))$, $x_4 = \Pi_1(\Pi_2(\Pi_2(\Pi_2(z))))$, $x_5 = \Pi_2(\Pi_2(\Pi_2(\Pi_2(z))))$.

We can also define a uniform projection function Π with the following property: if $z = \langle x_1, \ldots, x_n\rangle$, with $n \geq 2$, then

$$\Pi(i, n, z) = x_i$$

for all i, where $1 \leq i \leq n$. The idea is to view z as a n-tuple, and $\Pi(i, n, z)$ as the i-th component of that n-tuple. The function Π is defined by cases as follows:

$$\begin{aligned}
\Pi(i, 0, z) &= 0, \quad \text{for all } i \geq 0, \\
\Pi(i, 1, z) &= z, \quad \text{for all } i \geq 0, \\
\Pi(i, 2, z) &= \Pi_1(z), \quad \text{if } 0 \leq i \leq 1, \\
\Pi(i, 2, z) &= \Pi_2(z), \quad \text{for all } i \geq 2,
\end{aligned}$$

and for all $n \geq 2$,

$$\Pi(i, n+1, z) = \begin{cases} \Pi(i, n, z) & \text{if } 0 \leq i < n, \\ \Pi_1(\Pi(n, n, z)) & \text{if } i = n, \\ \Pi_2(\Pi(n, n, z)) & \text{if } i > n. \end{cases}$$

By a previous exercise, this is a legitimate primitive recursive definition.

Some basic properties of Π are given as exercises. In particular, the following properties are easily shown:

(a) $\langle 0,\ldots,0\rangle_n = 0$, $\langle x,0\rangle = \langle x,0,\ldots,0\rangle_n$;

(b) $\Pi(0,n,z) = \Pi(1,n,z)$ and $\Pi(i,n,z) = \Pi(n,n,z)$, for all $i \geq n$ and all $n,z \in \mathbb{N}$;

(c) $\langle \Pi(1,n,z),\ldots,\Pi(n,n,z)\rangle_n = z$, for all $n \geq 1$ and all $z \in \mathbb{N}$;

(d) $\Pi(i,n,z) \leq z$, for all $i,n,z \in \mathbb{N}$;

(e) There is a primitive recursive function Large, such that,

$$\Pi(i,n+1,\mathrm{Large}(n+1,z)) = z,$$

for $i,n,z \in \mathbb{N}$.

As a first application, we observe that we need only consider partial computable functions (partial recursive functions)[1] of a single argument. Indeed, let $\varphi\colon \mathbb{N}^n \to \mathbb{N}$ be a partial computable function of $n \geq 2$ arguments. Let

$$\overline{\varphi}(z) = \varphi(\Pi(1,n,z),\ldots,\Pi(n,n,z)),$$

for all $z \in \mathbb{N}$. Then, $\overline{\varphi}$ is a partial computable function of a single argument, and φ can be recovered from $\overline{\varphi}$, since

$$\varphi(x_1,\ldots,x_n) = \overline{\varphi}(\langle x_1,\ldots,x_n\rangle).$$

Thus, using $\langle -,-\rangle$ and Π as coding and decoding functions, we can restrict our attention to functions of a single argument.

Next, we show that there exist coding and decoding functions between Σ^* and $\{a_1\}^*$, and that partial computable functions over Σ^* can be recoded as partial computable functions over $\{a_1\}^*$. Since $\{a_1\}^*$ is isomorphic to $\mathbb{N}$, this shows that we can restrict out attention to functions defined over $\mathbb{N}$.

9.2 Equivalence of Alphabets

Given an alphabet $\Sigma = \{a_1,\ldots,a_k\}$, strings over Σ can be ordered by viewing strings as numbers in a number system where the digits are $a_1,\ldots,a_k$. In this number system, which is almost the number system with base k, the string a_1 corresponds to zero, and a_k to $k-1$. Hence, we have a kind of shifted number system in base k. For example, if $\Sigma = \{a,b,c\}$, a listing of Σ^* in the ordering corresponding to the number system begins with

$$a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc,$$
$$aaa, aab, aac, aba, abb, abc, \ldots.$$

[1]The term *partial recursive* is now considered old-fashion. Many researchers have switched to the term *partial computable*.

Clearly, there is an ordering function from Σ^* to $\mathbb{N}$ which is a bijection. Indeed, if $u = a_{i_1} \cdots a_{i_n}$, this function $f\colon \Sigma^* \to \mathbb{N}$ is given by

$$f(u) = i_1 k^{n-1} + i_2 k^{n-2} + \cdots + i_{n-1} k + i_n.$$

Since we also want a decoding function, we define the coding function $C_k\colon \Sigma^* \to \Sigma^*$ as follows:

$C_k(\epsilon) = \epsilon$, and if $u = a_{i_1} \cdots a_{i_n}$, then

$$C_k(u) = a_1^{i_1 k^{n-1} + i_2 k^{n-2} + \cdots + i_{n-1} k + i_n}.$$

The function C_k is primitive recursive, because

$$\begin{aligned} C_k(\epsilon) &= \epsilon, \\ C_k(xa_i) &= C_k(x)^k a_1^i. \end{aligned}$$

The inverse of C_k is a function $D_k\colon \{a_1\}^* \to \Sigma^*$. However, primitive recursive functions are total, and we need to extend D_k to Σ^*. This is easily done by letting

$$D_k(x) = D_k(a_1^{|x|})$$

for all $x \in \Sigma^*$. It remains to define D_k by primitive recursion over $\{a_1\}^*$. For this, we introduce three auxiliary functions p, q, r, defined as follows. Let

$$\begin{aligned} p(\epsilon) &= \epsilon, \\ p(xa_i) &= xa_i, \quad \text{if } i \neq k, \\ p(xa_k) &= p(x). \end{aligned}$$

Note that $p(x)$ is the result of deteting consecutive a_k's in the tail of x. Let

$$\begin{aligned} q(\epsilon) &= \epsilon, \\ q(xa_i) &= q(x)a_1. \end{aligned}$$

Note that $q(x) = a_1^{|x|}$. Finally, let

$$\begin{aligned} r(\epsilon) &= a_1, \\ r(xa_i) &= xa_{i+1}, \quad \text{if } i \neq k, \\ r(xa_k) &= xa_k. \end{aligned}$$

The function r is almost the successor function, for the ordering. Then, the trick is that $D_k(xa_i)$ is the successor of $D_k(x)$ in the ordering, and if

$$D_k(x) = ya_j a_k^n$$

with $j \neq k$, since the successor of $ya_j a_k^n$ is $ya_{j+1}a_k^n$, we can use r. Thus, we have

$$\begin{aligned} D_k(\epsilon) &= \epsilon, \\ D_k(xa_i) &= r(p(D_k(x)))q(D_k(x) - p(D_k(x))). \end{aligned}$$

Then, both C_k and D_k are primitive recursive, and $C_k \circ D_k = D_k \circ C_k = \mathrm{id}$.

Let $\varphi\colon \Sigma^* \to \Sigma^*$ be a partial function over Σ^*, and let

$$\varphi^+(x_1, \ldots, x_n) = C_k(\varphi(D_k(x_1), \ldots, D_k(x_n))).$$

The function φ^+ is defined over $\{a_1\}^*$. Also, for any partial function ψ over $\{a_1\}^*$, let

$$\psi^\sharp(x_1, \ldots, x_n) = D_k(\psi(C_k(x_1), \ldots, C_k(x_n))).$$

We claim that if ψ is a partial computable function over $\{a_1\}^*$, then $\psi^\sharp$ is partial computable over Σ^*, and that if φ is a partial computable function over Σ^*, then φ^+ is partial computable over $\{a_1\}^*$.

First, ψ can be extended to Σ^* by letting

$$\psi(x) = \psi(a_1^{|x|})$$

for all $x \in \Sigma^*$, and so, if ψ is partial computable, then so is $\psi^\sharp$ by composition. This seems equally obvious for φ and φ^+, but there is a difficulty. The problem is that φ^+ is defined as a composition of functions over Σ^*. We have to show how φ^+ can be defined directly over $\{a_1\}^*$ without using any additional alphabet symbols. This is done in Machtey and Young [12], see Section 2.2, Lemma 2.2.3.

Pairing functions can also be used to prove that certain functions are primitive recursive, even though their definition is not a legal primitive recursive definition. For example, consider the *Fibonacci function* defined as follows:

$$\begin{aligned} f(0) &= 1, \\ f(1) &= 1, \\ f(n+2) &= f(n+1) + f(n), \end{aligned}$$

for all $n \in \mathbb{N}$. This is not a legal primitive recursive definition, since $f(n+2)$ depends both on $f(n+1)$ and $f(n)$. In a primitive recursive definition, $g(y+1, \overline{x})$ is only allowed to depend upon $g(y, \overline{x})$.

Definition 9.1. Given any function $f\colon \mathbb{N}^n \to \mathbb{N}$, the function $\overline{f}\colon \mathbb{N}^{n+1} \to \mathbb{N}$ defined such that

$$\overline{f}(y, \overline{x}) = \langle f(0, \overline{x}), \ldots, f(y, \overline{x}) \rangle_{y+1}$$

is called the *course-of-value function* for f.

The following lemma holds.

Proposition 9.1. *Given any function $f\colon \mathbb{N}^n \to \mathbb{N}$, if f is primitive recursive, then so is $\overline{f}$.*

Proof. First, it is necessary to define a function *con* such that if $x = \langle x_1, \ldots, x_m\rangle$ and $y = \langle y_1, \ldots, y_n\rangle$, where $m, n \geq 1$, then

$$con(m, x, y) = \langle x_1, \ldots, x_m, y_1, \ldots, y_n\rangle.$$

This fact is left as an exercise. Now, if f is primitive recursive, let

$$\begin{aligned}\overline{f}(0, \overline{x}) &= f(0, \overline{x}),\\ \overline{f}(y+1, \overline{x}) &= con(y+1, \overline{f}(y, \overline{x}), f(y+1, \overline{x})),\end{aligned}$$

showing that $\overline{f}$ is primitive recursive. Conversely, if $\overline{f}$ is primitive recursive, then

$$f(y, \overline{x}) = \Pi(y+1, y+1, \overline{f}(y, \overline{x})),$$

and so, f is primitive recursive. □

Remark: Why is it that

$$\overline{f}(y+1, \overline{x}) = \langle \overline{f}(y, \overline{x}), f(y+1, \overline{x})\rangle$$

does not work?

We define *course-of-value recursion* as follows.

Definition 9.2. Given any two functions $g\colon \mathbb{N}^n \to \mathbb{N}$ and $h\colon \mathbb{N}^{n+2} \to \mathbb{N}$, the function $f\colon \mathbb{N}^{n+1} \to \mathbb{N}$ is defined by *course-of-value recursion* from g and h if

$$\begin{aligned}f(0, \overline{x}) &= g(\overline{x}),\\ f(y+1, \overline{x}) &= h(y, \overline{f}(y, \overline{x}), \overline{x}).\end{aligned}$$

The following lemma holds.

Proposition 9.2. *If $f\colon \mathbb{N}^{n+1} \to \mathbb{N}$ is defined by course-of-value recursion from g and h and g, h are primitive recursive, then f is primitive recursive.*

Proof. We prove that $\overline{f}$ is primitive recursive. Then, by Proposition 9.1, f is also primitive recursive. To prove that $\overline{f}$ is primitive recursive, observe that

$$\begin{aligned}\overline{f}(0, \overline{x}) &= g(\overline{x}),\\ \overline{f}(y+1, \overline{x}) &= con(y+1, \overline{f}(y, \overline{x}), h(y, \overline{f}(y, \overline{x}), \overline{x})).\end{aligned}$$

□

When we use Proposition 9.2 to prove that a function is primitive recursive, we rarely bother to construct a formal course-of-value recursion. Instead, we simply indicate how the value of $f(y+1, \overline{x})$ can be obtained in a primitive recursive manner from $f(0, \overline{x})$ through $f(y, \overline{x})$. Thus, an informal use of Proposition 9.2 shows that the Fibonacci function is primitive recursive. A rigorous proof of this fact is left as an exercise.

9.3 Coding of RAM Programs

In this Section, we present a specific encoding of RAM programs which allows us to treat programs as integers. Encoding programs as integers also allows us to have programs that take other programs as input, and we obtain a *universal program*. Universal programs have the property that given two inputs, the first one being the code of a program and the second one an input data, the universal program simulates the actions of the encoded program on the input data. A coding scheme is also called an indexing or a Gödel numbering, in honor to Gödel, who invented this technique.

From results of the previous Chapter, without loss of generality, we can restrict out attention to RAM programs computing partial functions of one argument over $\mathbb{N}$. Furthermore, we only need the following kinds of instructions, each instruction being coded as shown below. Since we are considering functions over the natural numbers, which corresponds to a one-letter alphabet, there is only one kind of instruction of the form `add` and `jmp` (and `add` increments by 1 the contents of the specified register Rj).

Ni		`add`	Rj	$\mathit{code} = \langle 1, i, j, 0\rangle$
Ni		`tail`	Rj	$\mathit{code} = \langle 2, i, j, 0\rangle$
Ni		`continue`		$\mathit{code} = \langle 3, i, 1, 0\rangle$
Ni	Rj	`jmp`	Nka	$\mathit{code} = \langle 4, i, j, k\rangle$
Ni	Rj	`jmp`	Nkb	$\mathit{code} = \langle 5, i, j, k\rangle$

Recall that a conditional jump causes a jump to the closest address Nk above or below iff Rj is nonzero, and if Rj is null, the next instruction is executed. We assume that all lines in a RAM program are numbered. This is always feasible, by labeling unnamed instructions with a new and unused line number.

The code of an instruction I is denoted as $\#I$. To simplify the notation, we introduce the following decoding primitive recursive functions Typ, Nam, Reg, and Jmp, defined as follows:

$$\begin{aligned} \mathrm{Typ}(x) &= \Pi(1,4,x), \\ \mathrm{Nam}(x) &= \Pi(2,4,x), \\ \mathrm{Reg}(x) &= \Pi(3,4,x), \\ \mathrm{Jmp}(x) &= \Pi(4,4,x). \end{aligned}$$

The functions yield the type, line number, register name, and line number jumped to, if any, for an instruction coded by x. Note that we have no need to interpret the values of these functions if x does not code an instruction.

We can define the primitive recursive predicate INST, such that $\mathrm{INST}(x)$ holds iff x codes an instruction. First, we need the connective $\supset$ (*implies*), defined such that

$$P \supset Q \quad \text{iff} \quad \neg P \vee Q.$$

Then, INST(x) holds iff:

$$\begin{aligned}
&[1 \leq \mathrm{Typ}(x) \leq 5] \wedge [1 \leq \mathrm{Reg}(x)] \wedge \\
&[\mathrm{Typ}(x) \leq 3 \supset \mathrm{Jmp}(x) = 0] \wedge \\
&[\mathrm{Typ}(x) = 3 \supset \mathrm{Reg}(x) = 1].
\end{aligned}$$

Program are coded as follows. If P is a RAM program composed of the n instructions $I_1, \ldots, I_n$, the code of P, denoted as $\#P$, is

$$\#P = \langle n, \#I_1, \ldots, \#I_n \rangle.$$

Recall from a previous exercise that

$$\langle n, \#I_1, \ldots, \#I_n \rangle = \langle n, \langle \#I_1, \ldots, \#I_n \rangle \rangle.$$

Also recall that

$$\langle x, y \rangle = ((x + y)^2 + 3x + y)/2.$$

Consider the following program Padd2 computing the function add2: $\mathbb{N} \to \mathbb{N}$ given by

$$\mathrm{add2}(n) = n + 2.$$

I_1 :	1	`add`	$R1$
I_2 :	2	`add`	$R1$
I_3 :	3	`continue`	

We have

$$\begin{aligned}
\#I1 &= \langle 1, 1, 1, 0 \rangle_4 = \langle 1, \langle 1, \langle 1, 0 \rangle \rangle \rangle = 37 \\
\#I2 &= \langle 1, 2, 1, 0 \rangle_4 = \langle 1, \langle 2, \langle 1, 0 \rangle \rangle \rangle = 92 \\
\#I3 &= \langle 3, 3, 1, 0 \rangle_4 = \langle 3, \langle 3, \langle 1, 0 \rangle \rangle \rangle = 234
\end{aligned}$$

and

$$\begin{aligned}
\#\mathrm{Padd2} &= \langle 3, \#I1, \#I2, \#I3 \rangle_4 = \langle 3, \langle 37, \langle 92, 234 \rangle \rangle \\
&= 1\,018\,748\,519\,973\,070\,618.
\end{aligned}$$

The codes get big fast!

We define the primitive recursive functions Ln, Pg, and Line, such that:

$$\begin{aligned}
\mathrm{Ln}(x) &= \Pi(1, 2, x), \\
\mathrm{Pg}(x) &= \Pi(2, 2, x), \\
\mathrm{Line}(i, x) &= \Pi(i, \mathrm{Ln}(x), \mathrm{Pg}(x)).
\end{aligned}$$

The function Ln yields the length of the program (the number of instructions), Pg yields the sequence of instructions in the program (really, a code for the sequence), and $\mathrm{Line}(i, x)$ yields the code of the ith instruction in the program. Again, if x does not code a program, there is no need to interpret these functions. However, note that by a previous exercise, it happens that

$$\begin{aligned} \mathrm{Line}(0, x) &= \mathrm{Line}(1, x), \quad \text{and} \\ \mathrm{Line}(\mathrm{Ln}(x), x) &= \mathrm{Line}(i, x), \quad \text{for all } i \geq x. \end{aligned}$$

The primitive recursive predicate PROG is defined such that $\mathrm{PROG}(x)$ holds iff x codes a program. Thus, $\mathrm{PROG}(x)$ holds if each line codes an instruction, each jump has an instruction to jump to, and the last instruction is a `continue`. Thus, $\mathrm{PROG}(x)$ holds iff

$$\begin{aligned} &\forall i \leq \mathrm{Ln}(x)[i \geq 1 \supset \\ &[\mathrm{INST}(\mathrm{Line}(i, x)) \wedge \mathrm{Typ}(\mathrm{Line}(\mathrm{Ln}(x), x)) = 3 \\ &\wedge [\mathrm{Typ}(\mathrm{Line}(i, x)) = 4 \supset \\ &\exists j \leq i - 1[j \geq 1 \wedge \mathrm{Nam}(\mathrm{Line}(j, x)) = \mathrm{Jmp}(\mathrm{Line}(i, x))]] \wedge \\ &[\mathrm{Typ}(\mathrm{Line}(i, x)) = 5 \supset \\ &\exists j \leq \mathrm{Ln}(x)[j > i \wedge \mathrm{Nam}(\mathrm{Line}(j, x)) = \mathrm{Jmp}(\mathrm{Line}(i, x))]]]] \end{aligned}$$

Note that we have used the fact proved as an exercise that if f is a primitive recursive function and P is a primitive recursive predicate, then $\exists x \leq f(y) P(x)$ is primitive recursive.

We are now ready to prove a fundamental result in the theory of algorithms. This result points out some of the limitations of the notion of algorithm.

Theorem 9.3. *(Undecidability of the halting problem) There is no RAM program* **Decider** *which halts for all inputs and has the following property when started with input x in register $R1$ and with input i in register $R2$ (the other registers being set to zero):*

(1) **Decider** *halts with output 1 iff i codes a program that eventually halts when started on input x (all other registers set to zero).*

(2) **Decider** *halts with output 0 in $R1$ iff i codes a program that runs forever when started on input x in $R1$ (all other registers set to zero).*

(3) If i does not code a program, then **Decider** *halts with output 2 in $R1$.*

Proof. Assume that **Decider** is such a RAM program, and let Q be the following program with a single input:

$$\text{Program } Q \text{ (code } q) \left\{ \begin{array}{llll} & R2 & \leftarrow & R1 \\ & & P & \\ N1 & & \texttt{continue} & \\ & R1 & \texttt{jmp} & N1a \\ & & \texttt{continue} & \end{array} \right.$$

Let i be the code of some program P. The key point is that *the termination behavior of Q on input i is exactly the opposite of the termination behavior of* **Decider** *on input i and code i*.

(1) If **Decider** says that program P coded by i *halts* on input i, then $R1$ just after the `continue` in line $N1$ contains 1, and Q *loops forever*.

(2) If **Decider** says that program P coded by i *loops forever* on input i, then $R1$ just after `continue` in line $N1$ contains 0, and Q *halts*.

The program Q can be translated into a program using only instructions of type 1, 2, 3, 4, 5, described previously, and let q *be the code of the program* Q.

Let us see what happens if we run the program Q on input q in $R1$ (all other registers set to zero).

Just after execution of the assignment $R2 \leftarrow R1$, the program **Decider** is started with q in both $R1$ and $R2$. Since **Decider** is supposed to halt for all inputs, it eventually halts with output 0 or 1 in $R1$. If **Decider** halts with output 1 in $R1$, then Q goes into an infinite loop, while if **Decider** halts with output 0 in $R1$, then Q halts. But then, because of the definition of **Decider**, we see that **Decider** says that Q halts when started on input q iff Q loops forever on input q, and that Q loops forever on input q iff Q halts on input q, a contradiction. Therefore, **Decider** cannot exist. $\square$

If we identify the notion of algorithm with that of a RAM program which halts for all inputs, the above theorem says that there is no algorithm for deciding whether a RAM program eventually halts for a given input. We say that the halting problem for RAM programs is *undecidable* (or *unsolvable*).

The above theorem also implies that the halting problem for Turing machines is undecidable. Indeed, if we had an algorithm for solving the halting problem for Turing machines, we could solve the halting problem for RAM programs as follows: first, apply the algorithm for translating a RAM program into an equivalent Turing machine, and then apply the algorithm solving the halting problem for Turing machines.

The argument is typical in computability theory and is called a "reducibility argument."

Our next goal is to define a primitive recursive function that describes the computation of RAM programs. Assume that we have a RAM program P using n registers $R1, \ldots, Rn$, whose contents are denoted as $r_1, \ldots, r_n$. We can code $r_1, \ldots, r_n$ into a single integer $\langle r_1, \ldots, r_n \rangle$. Conversely, every integer x can be viewed as coding the contents of $R1, \ldots, Rn$, by taking the sequence $\Pi(1, n, x), \ldots, \Pi(n, n, x)$.

Actually, it is not necessary to know n, the number of registers, if we make the following observation:

$$\mathrm{Reg}(\mathrm{Line}(i, x)) \leq \mathrm{Line}(i, x) \leq \mathrm{Pg}(x)$$

for all $i, x \in \mathbb{N}$. Then, if x codes a program, then $R1, \ldots, Rx$ certainly include all the registers in the program. Also note that from a previous exercise,

$$\langle r_1, \ldots, r_n, 0, \ldots, 0\rangle = \langle r_1, \ldots, r_n, 0\rangle.$$

We now define the primitive recursive functions Nextline, Nextcont, and Comp, describing the computation of RAM programs.

Definition 9.3. Let x code a program and let i be such that $1 \leq i \leq \mathrm{Ln}(x)$. The following functions are defined:

(1) $\mathrm{Nextline}(i, x, y)$ is the number of the next instruction to be executed after executing the ith instruction in the program coded by x, where the contents of the registers is coded by y.

(2) $\mathrm{Nextcont}(i, x, y)$ is the code of the contents of the registers after executing the ith instruction in the program coded by x, where the contents of the registers is coded by y.

(3) $\mathrm{Comp}(x, y, m) = \langle i, z\rangle$, where i and z are defined such that after running the program coded by x for m steps, where the initial contents of the program registers are coded by y, the next instruction to be executed is the ith one, and z is the code of the current contents of the registers.

Proposition 9.4. *The functions* Nextline*,* Nextcont*, and* Comp*, are primitive recursive.*

Proof. (1) $\mathrm{Nextline}(i, x, y) = i + 1$, unless the ith instruction is a jump and the contents of the register being tested is nonzero:

$$\begin{aligned}
&\mathrm{Nextline}(i, x, y) = \\
&\qquad \max j \leq \mathrm{Ln}(x)[j < i \wedge \mathrm{Nam}(\mathrm{Line}(j, x)) = \mathrm{Jmp}(\mathrm{Line}(i, x))] \\
&\qquad \text{if } \mathrm{Typ}(\mathrm{Line}(i, x)) = 4 \wedge \Pi(\mathrm{Reg}(\mathrm{Line}(i, x)), x, y) \neq 0 \\
&\qquad \min j \leq \mathrm{Ln}(x)[j > i \wedge \mathrm{Nam}(\mathrm{Line}(j, x)) = \mathrm{Jmp}(\mathrm{Line}(i, x))] \\
&\qquad \text{if } \mathrm{Typ}(\mathrm{Line}(i, x)) = 5 \wedge \Pi(\mathrm{Reg}(\mathrm{Line}(i, x)), x, y) \neq 0 \\
&\qquad i + 1 \text{ otherwise.}
\end{aligned}$$

Note that according to this definition, if the ith line is the final `continue`, then Nextline signals that the program has halted by yielding

$$\mathrm{Nextline}(i, x, y) > \mathrm{Ln}(x).$$

(2) We need two auxiliary functions Add and Sub defined as follows.

$\mathrm{Add}(j, x, y)$ is the number coding the contents of the registers used by the program coded by x after register Rj coded by $\Pi(j, x, y)$ has been increased by 1, and

$\text{Sub}(j, x, y)$ codes the contents of the registers after register Rj has been decremented by 1 (y codes the previous contents of the registers). It is easy to see that

$$\begin{aligned}\text{Sub}(j, x, y) = \min z \leq y[&\Pi(j, x, z) = \Pi(j, x, y) - 1 \\ &\land \forall k \leq x[0 < k \neq j \supset \Pi(k, x, z) = \Pi(k, x, y)]].\end{aligned}$$

The definition of Add is slightly more tricky. We leave as an exercise to the reader to prove that:

$$\begin{aligned}\text{Add}(j, x, y) = \min z \leq \text{Large}(x, y + 1)& \\ [\Pi(j, x, z) = \Pi(j, x, y) + 1 \land \forall k \leq x[0 < k \neq j \supset \Pi(k, x, z) = \Pi(k, x, y)]],&\end{aligned}$$

where the function Large is the function defined in an earlier exercise. Then

$$\begin{aligned}&\text{Nextcont}(i, x, y) = \\ &\qquad \text{Add}(\text{Reg}(\text{Line}(i, x), x, y) \quad \text{if} \quad \text{Typ}(\text{Line}(i, x)) = 1 \\ &\qquad \text{Sub}(\text{Reg}(\text{Line}(i, x), x, y) \quad \text{if} \quad \text{Typ}(\text{Line}(i, x)) = 2 \\ &\qquad y \quad \text{if} \quad \text{Typ}(\text{Line}(i, x)) \geq 3.\end{aligned}$$

(3) Recall that $\Pi_1(z) = \Pi(1, 2, z)$ and $\Pi_2(z) = \Pi(2, 2, z)$. The function Comp is defined by primitive recursion as follows:

$$\begin{aligned}\text{Comp}(x, y, 0) &= \langle 1, y\rangle \\ \text{Comp}(x, y, m + 1) &= \langle \text{Nextline}(\Pi_1(\text{Comp}(x, y, m)), x, \Pi_2(\text{Comp}(x, y, m))), \\ &\qquad \text{Nextcont}(\Pi_1(\text{Comp}(x, y, m)), x, \Pi_2(\text{Comp}(x, y, m)))\rangle.\end{aligned}$$

Recall that $\Pi_1(\text{Comp}(x, y, m))$ is the number of the next instruction to be executed and that $\Pi_2(\text{Comp}(x, y, m))$ codes the current contents of the registers. ☐

We can now reprove that every RAM computable function is partial computable. Indeed, assume that x codes a program P.

We define the partial function End so that for all x, y, where x codes a program and y codes the contents of its registers, $\text{End}(x, y)$ is the number of steps for which the computation runs before halting, if it halts. If the program does not halt, then $\text{End}(x, y)$ is undefined. Since

$$\text{End}(x, y) = \min m[\Pi_1(\text{Comp}(x, y, m)) = \text{Ln}(x)],$$

If y is the value of the register $R1$ before the program P coded by x is started, recall that the contents of the registers is coded by $\langle y, 0\rangle$. Noticing that 0 and 1 do not code programs, we note that if x codes a program, then $x \geq 2$, and $\Pi_1(z) = \Pi(1, x, z)$ is the contents of $R1$ as coded by z.

Since $\mathrm{Comp}(x, y, m) = \langle i, z\rangle$, we have

$$\Pi_1(\mathrm{Comp}(x, y, m)) = i,$$

where i is the number (index) of the instruction reached after running the program P coded by x with initial values of the registers coded by y for m steps. Thus, P halts if i is the last instruction in P, namely $\mathrm{Ln}(x)$, iff

$$\Pi_1(\mathrm{Comp}(x, y, m)) = \mathrm{Ln}(x).$$

End is a partial computable function; it can be computed by a RAM program involving only one while loop searching for the number of steps m. However, in general, End is not a total function.

If φ is the partial computable function computed by the program P coded by x, then we have

$$\varphi(y) = \Pi_1(\Pi_2(\mathrm{Comp}(x, \langle y, 0\rangle, \mathrm{End}(x, \langle y, 0\rangle)))).$$

This is because if $m = \mathrm{End}(x, \langle y, 0\rangle)$ is the number of steps after which the program P coded by x halts on input y, then

$$\mathrm{Comp}(x, \langle y, 0\rangle, m)) = \langle \mathrm{Ln}(x), z\rangle,$$

where z is the code of the register contents when the program stops. Consequently

$$\begin{aligned} z &= \Pi_2(\mathrm{Comp}(x, \langle y, 0\rangle, m)) \\ z &= \Pi_2(\mathrm{Comp}(x, \langle y, 0\rangle, \mathrm{End}(x, \langle y, 0\rangle))). \end{aligned}$$

The value of the register $R1$ is $\Pi_1(z)$, that is

$$\varphi(y) = \Pi_1(\Pi_2(\mathrm{Comp}(x, \langle y, 0\rangle, \mathrm{End}(x, \langle y, 0\rangle)))).$$

Observe that φ is written in the form $\varphi = g \circ \min f$, for some primitive recursive functions f and g.

We can also exhibit a partial computable function which enumerates all the unary partial computable functions. It is a *universal function*.

Abusing the notation slightly, we will write $\varphi(x, y)$ for $\varphi(\langle x, y\rangle)$, viewing φ as a function of two arguments (however, φ is really a function of a single argument). We define the function φ_{univ} as follows:

$$\varphi_{univ}(x, y) = \begin{cases} \Pi_1(\Pi_2(\mathrm{Comp}(x, \langle y, 0\rangle, \mathrm{End}(x, \langle y, 0\rangle)))) & \text{if } \mathrm{PROG}(x), \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The function φ_{univ} is a partial computable function with the following property: for every x coding a RAM program P, for every input y,

$$\varphi_{univ}(x, y) = \varphi_x(y),$$

the value of the partial computable function φ_x computed by the RAM program P coded by x. If x does not code a program, then $\varphi_{univ}(x, y)$ is undefined for all y.

By Proposition 8.9, the partial function φ_{univ} is not computable (recursive).[2] Indeed, being an enumerating function for the partial computable functions, it is an enumerating function for the total computable functions, and thus, it cannot be computable. Being a partial function saves us from a contradiction.

The existence of the function φ_{univ} leads us to the notion of an *indexing* of the RAM programs.

We can define a listing of the RAM programs as follows. If x codes a program (that is, if PROG(x) holds) and P is the program that x codes, we call this program P the xth RAM program and denote it as P_x. If x does not code a program, we let P_x be the program that diverges for every input:

```
N1        add       R1
N1   R1   jmp       N1a
N1        continue
```

Therefore, in all cases, P_x stands for the xth RAM program. Thus, we have a listing of RAM programs, $P_0, P_1, P_2, P_3, \ldots$, such that every RAM program (of the restricted type considered here) appears in the list exactly once, except for the "infinite loop" program. For example, the program Padd2 (adding 2 to an integer) appears as

$$P_{1\,018\,748\,519\,973\,070\,618}.$$

In particular, note that φ_{univ} being a partial computable function, it is computed by some RAM program UNIV that has a code *univ* and is the program P_{univ} in the list.

Having an indexing of the RAM programs, we also have an indexing of the partial computable functions.

Definition 9.4. For every integer $x \geq 0$, we let P_x be the RAM program coded by x as defined earlier, and φ_x be the partial computable function computed by P_x.

For example, the function add2 (adding 2 to an integer) appears as

$$\varphi_{1\,018\,748\,519\,973\,070\,618}.$$

Remark: Kleene used the notation $\{x\}$ for the partial computable function coded by x. Due to the potential confusion with singleton sets, we follow Rogers, and use the notation φ_x.

[2] The term *recursive function* is now considered old-fashion. Many researchers have switched to the term *computable function*.

It is important to observe that *different programs* P_x and P_y may compute the *same function*, that is, while $P_x \neq P_y$ for all $x \neq y$, it is possible that $\varphi_x = \varphi_y$. In fact, it is *undecidable* whether $\varphi_x = \varphi_y$.

The existence of the universal function φ_{univ} is sufficiently important to be recorded in the following Lemma.

Proposition 9.5. *For the indexing of RAM programs defined earlier, there is a universal partial computable function φ_{univ} such that, for all $x, y \in \mathbb{N}$, if φ_x is the partial computable function computed by P_x, then*

$$\varphi_x(y) = \varphi_{univ}(\langle x, y\rangle).$$

The program UNIV computing φ_{univ} can be viewed as an *interpreter* for RAM programs. By giving the universal program UNIV the "program" x and the "data" y, we get the result of executing program P_x on input y. We can view the RAM model as a *stored program computer*.

By Theorem 9.3 and Proposition 9.5, the halting problem for the single program UNIV is undecidable. Otherwise, the halting problem for RAM programs would be decidable, a contradiction. It should be noted that the program UNIV can actually be written (with a certain amount of pain).

The object of the next Section is to show the existence of Kleene's T-predicate. This will yield another important normal form. In addition, the T-predicate is a basic tool in recursion theory.

9.4 Kleene's T-Predicate

In Section 9.3, we have encoded programs. The idea of this Section is to also encode *computations* of RAM programs. Assume that x codes a program, that y is some input (not a code), and that z codes a computation of P_x on input y. The predicate $T(x, y, z)$ is defined as follows:

$T(x, y, z)$ holds iff x codes a RAM program, y is an input, and z codes a halting computation of P_x on input y.

We will show that T is primitive recursive. First, we need to encode computations. We say that z codes a computation of length $n \geq 1$ if

$$z = \langle n + 2, \langle 1, y_0\rangle, \langle i_1, y_1\rangle, \ldots, \langle i_n, y_n\rangle\rangle,$$

where each i_j is the physical location of the next instruction to be executed and each y_j codes the contents of the registers just before execution of the instruction at the location i_j. Also, y_0 codes the initial contents of the registers, that is, $y_0 = \langle y, 0\rangle$, for some input y. We let $\mathrm{Ln}(z) = \Pi_1(z)$. Note that i_j denotes the physical location of the next instruction to

be executed in the sequence of instructions constituting the program coded by x, and not the line number (label) of this instruction. Thus, the first instruction to be executed is in location 1, $1 \leq i_j \leq \mathrm{Ln}(x)$, and $i_{n-1} = \mathrm{Ln}(x)$. Since the last instruction which is executed is the last physical instruction in the program, namely, a `continue`, there is no next instruction to be executed after that, and i_n is irrelevant. Writing the definition of T is a little simpler if we let $i_n = \mathrm{Ln}(x) + 1$.

Definition 9.5. The T-predicate is the primitive recursive predicate defined as follows:

$$\begin{aligned}
&T(x, y, z) \quad \text{iff} \quad \mathrm{PROG}(x) \text{ and } (\mathrm{Ln}(z) \geq 3) \text{ and} \\
&\forall j \leq \mathrm{Ln}(z) - 3[0 \leq j \supset \\
&\mathrm{Nextline}(\Pi_1(\Pi(j+2, \mathrm{Ln}(z), z)), x, \Pi_2(\Pi(j+2, \mathrm{Ln}(z), z))) = \Pi_1(\Pi(j+3, \mathrm{Ln}(z), z)) \text{ and} \\
&\mathrm{Nextcont}(\Pi_1(\Pi(j+2, \mathrm{Ln}(z), z)), x, \Pi_2(\Pi(j+2, \mathrm{Ln}(z), z))) = \Pi_2(\Pi(j+3, \mathrm{Ln}(z), z)) \text{ and} \\
&\Pi_1(\Pi(\mathrm{Ln}(z) - 1, \mathrm{Ln}(z), z)) = \mathrm{Ln}(x) \text{ and} \\
&\Pi_1(\Pi(2, \mathrm{Ln}(z), z)) = 1 \text{ and} \\
&y = \Pi_1(\Pi_2(\Pi(2, \mathrm{Ln}(z), z))) \text{ and } \Pi_2(\Pi_2(\Pi(2, \mathrm{Ln}(z), z))) = 0]
\end{aligned}$$

The reader can verify that $T(x, y, z)$ holds iff x codes a RAM program, y is an input, and z codes a halting computation of P_x on input y. In order to extract the output of P_x from z, we define the primitive recursive function Res as follows:

$$\mathrm{Res}(z) = \Pi_1(\Pi_2(\Pi(\mathrm{Ln}(z), \mathrm{Ln}(z), z))).$$

The explanation for this formula is that $\mathrm{Res}(z)$ are the contents of register $R1$ when P_x halts, that is, $\Pi_1(y_{\mathrm{Ln}(z)})$. Using the T-predicate, we get the so-called Kleene normal form.

Theorem 9.6. *(Kleene Normal Form) Using the indexing of the partial computable functions defined earlier, we have*

$$\varphi_x(y) = \mathrm{Res}[\min z(T(x, y, z))],$$

where $T(x, y, z)$ *and* Res *are primitive recursive.*

Note that the universal function φ_{univ} can be defined as

$$\varphi_{univ}(x, y) = \mathrm{Res}[\min z(T(x, y, z))].$$

There is another important property of the partial computable functions, namely, that composition is effective. We need two auxiliary primitive recursive functions. The function Conprogs creates the code of the program obtained by concatenating the programs P_x and P_y, and for $i \geq 2$, $\mathrm{Cumclr}(i)$ is the code of the program which clears registers $R2, \ldots, Ri$. To get Cumclr, we can use the function $\mathrm{clr}(i)$ such that $\mathrm{clr}(i)$ is the code of the program

$N1$		`tail`	Ri
$N1$	Ri	`jmp`	$N1a$
N		`continue`	

We leave it as an exercise to prove that clr, Conprogs, and Cumclr, are primitive recursive.

Theorem 9.7. *There is a primitive recursive function c such that*

$$\varphi_{c(x,y)} = \varphi_x \circ \varphi_y.$$

Proof. If both x and y code programs, then $\varphi_x \circ \varphi_y$ can be computed as follows: Run P_y, clear all registers but $R1$, then run P_x. Otherwise, let loop be the index of the infinite loop program:

$$c(x,y) = \begin{cases} \text{Conprogs}(y, \text{Conprogs}(\text{Cumclr}(y), x)) & \text{if PROG}(x) \text{ and PROG}(y) \\ \text{loop} & \text{otherwise.} \end{cases}$$

∎

9.5 A Simple Function Not Known to be Computable

The "$3n+1$ problem" proposed by Collatz around 1937 is the following:

Given any positive integer $n \geq 1$, construct the sequence $c_i(n)$ as follows starting with $i = 1$:

$$c_1(n) = n$$
$$c_{i+1}(n) = \begin{cases} c_i(n)/2 & \text{if } c_i(n) \text{ is even} \\ 3c_i(n) + 1 & \text{if } c_i(n) \text{ is odd.} \end{cases}$$

Observe that for $n = 1$, we get the infinite periodic sequence

$$1 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1 \Longrightarrow \cdots,$$

so we may assume that we stop the first time that the sequence $c_i(n)$ reaches the value 1 (if it actually does). Such an index i is called the *stopping time* of the sequence. And this is the problem:

Conjecture (Collatz):

For any starting integer value $n \geq 1$, the sequence $(c_i(n))$ always reaches 1.

Starting with $n = 3$, we get the sequence

$$3 \Longrightarrow 10 \Longrightarrow 5 \Longrightarrow 16 \Longrightarrow 8 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1.$$

Starting with $n = 5$, we get the sequence

$$5 \Longrightarrow 16 \Longrightarrow 8 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1.$$

Starting with $n = 6$, we get the sequence

$$6 \Longrightarrow 3 \Longrightarrow 10 \Longrightarrow 5 \Longrightarrow 16 \Longrightarrow 8 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1.$$

Starting with $n = 7$, we get the sequence

$$\begin{aligned} 7 &\Longrightarrow 22 \Longrightarrow 11 \Longrightarrow 34 \Longrightarrow 17 \Longrightarrow 52 \Longrightarrow 26 \Longrightarrow 13 \Longrightarrow 40 \\ &\Longrightarrow 20 \Longrightarrow 10 \Longrightarrow 25 \Longrightarrow 16 \Longrightarrow 8 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1. \end{aligned}$$

One might be surprised to find that for $n = 27$, it takes 111 steps to reach 1, and for $n = 97$, it takes 118 steps. I computed the stopping times for n up to 10^7 and found that the largest stopping time, 525 (524 steps), is obtained for $n = 837799$. The terms of this sequence reach values over 2.9×10^9. The graph of the sequence $s(837799)$ is shown in Figure 9.1.

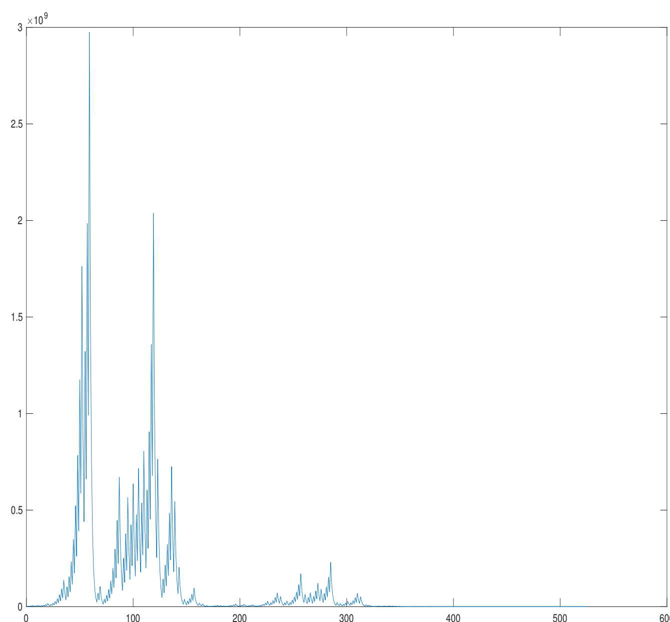


Figure 9.1: Graph of the sequence for $n = 837799$.

We can define the partial computable function C (with positive integer inputs) defined by

$$C(n) = \text{the smallest } i \text{ such that } c_i(n) = 1 \text{ if it exists.}$$

Then the Collatz conjecture is equivalent to asserting that the function C is (total) computable. The graph of the function C for $1 \leq n \leq 10^7$ is shown in Figure 9.2.

So far, the conjecture remains open. It has been checked by computer for all integers $\leq 87 \times 2^{60}$.

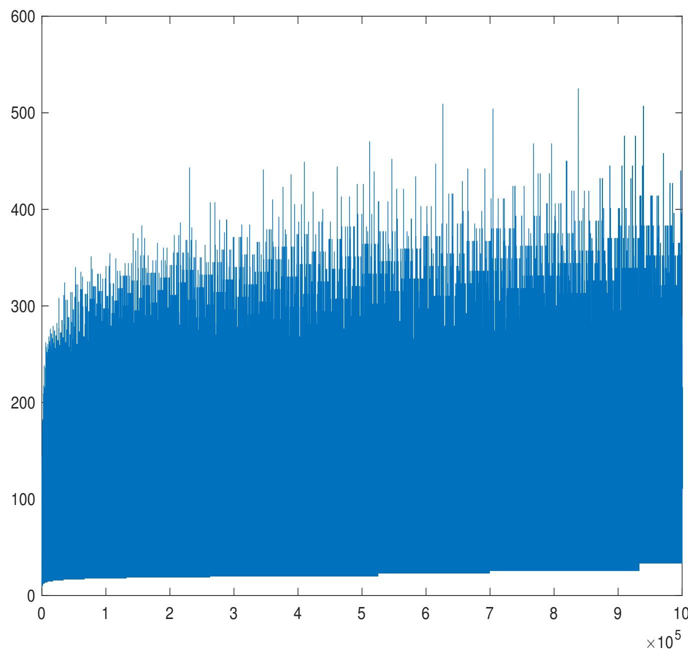


Figure 9.2: Graph of the function C for $1 \leq n \leq 10^7$.

9.6 A Non-Computable Function; Busy Beavers

Total functions that are not computable must grow very fast and thus are very complicated. Yet, in 1962, Radó published a paper in which he defined two functions Σ and S (involving computations of Turing machines) that are total and not computable.

Consider Turing machines with a tape alphabet $\Gamma = \{1, B\}$ with two symbols (B being the blank). We also assume that these Turing machines have a special final state q_F, which is a blocking state (there are no transitions from q_F). We do not count this state when counting the number of states of such Turing machines. The game is to run such Turing machines with a fixed number of states n starting on a blank tape, with the goal of producing the maximum number of (not necessarily consecutive) ones (1).

Definition 9.6. The function Σ (defined on the positive natural numbers) is defined as the maximum number $\Sigma(n)$ of (not necessarily consecutive) 1's written on the tape after a Turing machine with $n \geq 1$ states started on the blank tape halts. The function S is defined as the maximum number $S(n)$ of moves that can be made by a Turing machine of the above type with n states before it halts, started on the blank tape.

A Turing machine with n states that writes the maximum number $\Sigma(n)$ of 1's when started on the blank tape is called a *busy beaver*.

Busy beavers are hard to find, even for small n. First, it can be shown that the number of distinct Turing machines of the above kind with n states is $(4(n+1))^{2n}$. Second, since it is undecidable whether a Turing machine halts on a given input, it is hard to tell which machines loop or halt after a very long time.

Here is a summary of what is known for $1 \leq n \leq 6$. Observe that the exact value of $\Sigma(5), \Sigma(6), S(5)$ and $S(6)$ is unknown.

n	$\Sigma(n)$	$S(n)$
1	1	1
2	4	6
3	6	21
4	13	107
5	≥ 4098	$\geq 47,176,870$
6	$\geq 95,524,079$	$\geq 8,690,333,381,690,951$
6	$\geq 3.515 \times 10^{18267}$	$\geq 7.412 \times 10^{36534}$

The first entry in the table for $n = 6$ corresponds to a machine due to Heiner Marxen (1999). This record was surpassed by Pavel Kropitz in 2010, which corresponds to the second entry for $n = 6$. The machines achieving the record in 2017 for $n = 4, 5, 6$ are shown below, where the blank is denoted Δ instead of B, and where the special halting states is denoted H:

4-state busy beaver:

	A	B	C	D
Δ	$(1, R, B)$	$(1, L, A)$	$(1, R, H)$	$(1, R, D)$
1	$(1, L, B)$	(Δ, L, C)	$(1, L, D)$	(Δ, R, A)

The above machine output 13 ones in 107 steps. In fact, the output is

$$\Delta\,\Delta\,1\,\Delta\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,\Delta\,\Delta.$$

5-state best contender:

	A	B	C	D	E
Δ	$(1, R, B)$	$(1, R, C)$	$(1, R, D)$	$(1, L, A)$	$(1, R, H)$
1	$(1, L, C)$	$(1, R, B)$	(Δ, L, E)	$(1, L, D)$	(Δ, L, A)

The above machine output 4098 ones in $47,176,870$ steps.

6-state contender (Heiner Marxen):

	A	B	C	D	E	F
Δ	$(1, R, B)$	$(1, L, C)$	(Δ, R, F)	$(1, R, A)$	$(1, L, H)$	(Δ, L, A)
1	$(1, R, A)$	$(1, L, B)$	$(1, L, D)$	(Δ, L, E)	$(1, L, F)$	(Δ, L, C)

The above machine outputs $96, 524, 079$ ones in $8, 690, 333, 381, 690, 951$ steps.

6-state best contender (Pavel Kropitz):

	A	B	C	D	E	F
Δ	$(1, R, B)$	$(1, R, C)$	$(1, L, D)$	$(1, R, E)$	$(1, L, A)$	$(1, L, H)$
1	$(1, L, E)$	$(1, R, F)$	(Δ, R, B)	(Δ, L, C)	(Δ, R, D)	$(1, R, C)$

The above machine output at least 3.515×10^{18267} ones!

The reason why it is so hard to compute Σ and S is that they are not computable!

Theorem 9.8. *The functions Σ and S are total functions that are not computable (not recursive).*

Proof sketch. The proof consists in showing that Σ (and similarly for S) eventually outgrows any computable function. More specifically, we claim that for every computable function f, there is some positive integer k_f such that

$$\Sigma(n + k_f) \geq f(n) \quad \text{for all } n \geq 0.$$

We simply have to pick k_f to be the number of states of a Turing machine M_f computing f. Then, we can create a Turing machine $M_{n,f}$ that works as follows. Using n of its states, it writes n ones on the tape, and then it simulates M_f with input 1^n. Since the ouput of $M_{n,f}$ started on the blank tape consists of $f(n)$ ones, and since $\Sigma(n + k_f)$ is the maximum number of ones that a turing machine with $n + k_f$ states will ouput when it stops, we must have

$$\Sigma(n + k_f) \geq f(n) \quad \text{for all } n \geq 0.$$

Next observe that $\Sigma(n) < \Sigma(n + 1)$, because we can create a Turing machine with $n + 1$ states which simulates a busy beaver machine with n states, and then writes an extra 1 when the busy beaver stops, by making a transition to the $(n + 1)$th state. It follows immediately that if $m < n$ then $\Sigma(m) < \Sigma(n)$. If Σ was computaable, then so would be the function g given by $g(n) = \Sigma(2n)$. By the above, we would have

$$\Sigma(n + k_g) \geq g(n) = \Sigma(2n) \quad \text{for all } n \geq 0,$$

and for $n > k_g$, since $2n > n + k_k$, we would have $\Sigma(n + n_g) < \Sigma(2n)$, contradicting the fact that $\Sigma(n + n_g) \geq \Sigma(2n)$.

Since by definition $S(n)$ is the maximum number of moves that can be made by a Turing machine of the above type with n states before it halts, $S(n) \geq \Sigma(n)$. Then the same reasoning as above shows that S is not a computable function. $\square$

The zoo of comutable and non-computable functions is illustrated in Figure 9.3.

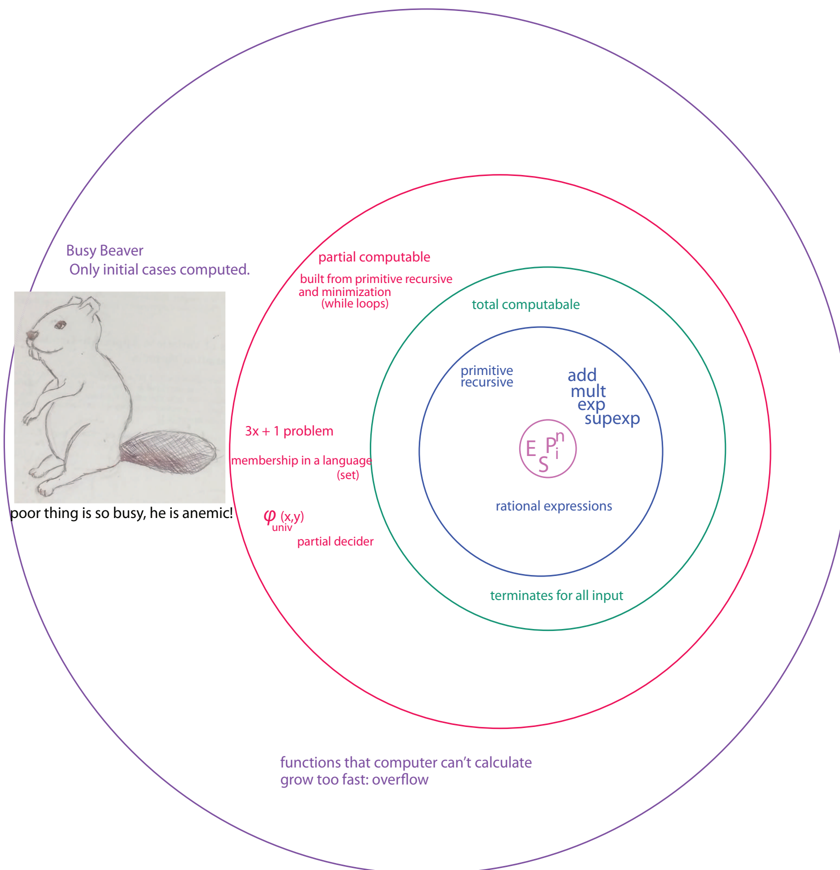


Figure 9.3: Computability Classification of Functions.

Chapter 10

Elementary Recursive Function Theory

10.1 Acceptable Indexings

In Chapter 9, we have exhibited a specific indexing of the partial computable functions by encoding the RAM programs. Using this indexing, we showed the existence of a universal function φ_{univ} and of a computable function c, with the property that for all $x, y \in \mathbb{N}$,

$$\varphi_{c(x,y)} = \varphi_x \circ \varphi_y.$$

It is natural to wonder whether the same results hold if a different coding scheme is used or if a different model of computation is used, for example, Turing machines. In other words, we would like to know if our results depend on a specific coding scheme or not.

Our previous results showing the characterization of the partial computable functions being independennt of the specific model used, suggests that it might be possible to pinpoint certain properties of coding schems which would allow an axiomatic development of recursive function theory. What we are aiming at is to find some simple properties of "nice" coding schemes that allow one to proceed without using explicit coding schemes, as long as the above properties hold.

Remarkably, such properties exist. Furthermore, any two coding schemes having these properties are equivalent in a strong sense (effectively equivalent), and so, one can pick any such coding scheme without any risk of losing anything else because the wrong coding scheme was chosen. Such coding schemes, also called indexings, or Gödel numberings, or even programming systems, are called *acceptable indexings*.

Definition 10.1. An *indexing* of the partial computable functions is an infinite sequence $\varphi_0, \varphi_1, \ldots,$ of partial computable functions that includes all the partial computable functions of one argument (there might be repetitions, this is why we are not using the term enumeration). An indexing is *universal* if it contains the partial computable function φ_{univ}

such that
$$\varphi_{univ}(i, x) = \varphi_i(x)$$
for all $i, x \in \mathbb{N}$. An indexing is *acceptable* if it is universal and if there is a total computable function c for composition, such that
$$\varphi_{c(i,j)} = \varphi_i \circ \varphi_j$$
for all $i, j \in \mathbb{N}$.

From Chapter 9, we know that the specific indexing of the partial computable functions given for RAM programs is acceptable. Another characterization of acceptable indexings left as an exercise is the following: an indexing $\psi_0, \psi_1, \psi_2, \ldots$ of the partial computable functions is acceptable iff there exists a total computable function f translating the RAM indexing of Section 9.3 into the indexing $\psi_0, \psi_1, \psi_2, \ldots$, that is,
$$\varphi_i = \psi_{f(i)}$$
for all $i \in \mathbb{N}$.

A very useful property of acceptable indexings is the so-called "s-m-n Theorem". Using the slightly loose notation $\varphi(x_1, \ldots, x_n)$ for $\varphi(\langle x_1, \ldots, x_n\rangle)$, the s-m-n theorem says the following. Given a function φ considered as having $m + n$ arguments, if we fix the values of the first m arguments and we let the other n arguments vary, we obtain a function ψ of n arguments. Then, the index of ψ depends in a computable fashion upon the index of φ and the first m arguments $x_1, \ldots, x_m$. We can "pull" the first m arguments of φ into the index of ψ.

Theorem 10.1. *(The "s-m-n Theorem") For any acceptable indexing $\varphi_0, \varphi_1, \ldots,$ there is a total computable function s, such that, for all $i, m, n \geq 1$, for all $x_1, \ldots, x_m$ and all $y_1, \ldots, y_n$, we have*
$$\varphi_{s(i,m,x_1,\ldots,x_m)}(y_1, \ldots, y_n) = \varphi_i(x_1, \ldots, x_m, y_1, \ldots, y_n).$$

Proof. First, note that the above identity is really
$$\varphi_{s(i,m,\langle x_1,\ldots,x_m\rangle)}(\langle y_1, \ldots, y_n\rangle) = \varphi_i(\langle x_1, \ldots, x_m, y_1, \ldots, y_n\rangle).$$
Recall that there is a primitive recursive function Con such that
$$\mathrm{Con}(m, \langle x_1, \ldots, x_m\rangle, \langle y_1, \ldots, y_n\rangle) = \langle x_1, \ldots, x_m, y_1, \ldots, y_n\rangle$$
for all $x_1, \ldots, x_m, y_1, \ldots, y_n \in \mathbb{N}$. Hence, a computable function s such that
$$\varphi_{s(i,m,x)}(y) = \varphi_i(\mathrm{Con}(m, x, y))$$
will do. We define some auxiliary primitive recursive functions as follows:
$$P(y) = \langle 0, y\rangle \quad \text{and} \quad Q(\langle x, y\rangle) = \langle x + 1, y\rangle.$$

Since we have an indexing of the partial computable functions, there are indices p and q such that $P = \varphi_p$ and $Q = \varphi_q$. Let R be defined such that

$$R(0) = p,$$
$$R(x+1) = c(q, R(x)),$$

where c is the computable function for composition given by the indexing. We leave as an exercise to prove that

$$\varphi_{R(x)}(y) = \langle x, y \rangle$$

for all $x, y \in \mathbb{N}$. Also, recall that $\langle x, y, z \rangle = \langle x, \langle y, z \rangle \rangle$, by definition of pairing. Then, we have

$$\varphi_{R(x)} \circ \varphi_{R(y)}(z) = \varphi_{R(x)}(\langle y, z \rangle) = \langle x, y, z \rangle.$$

Finally, let k be an index for the function Con, that is, let

$$\varphi_k(\langle m, x, y \rangle) = \mathrm{Con}(m, x, y).$$

Define s by

$$s(i, m, x) = c(i, c(k, c(R(m), R(x)))).$$

Then, we have

$$\varphi_{s(i,m,x)}(y) = \varphi_i \circ \varphi_k \circ \varphi_{R(m)} \circ \varphi_{R(x)}(y) = \varphi_i(\mathrm{Con}(m, x, y)),$$

as desired. Notice that if the composition function c is primitive recursive, then s is also primitive recursive. In particular, for the specific indexing of the RAM programs given in Section 9.3, the function s is primitive recursive. ☐

As a first application of the s-m-n Theorem, we show that any two acceptable indexings are effectively inter-translatable.

Theorem 10.2. *Let $\varphi_0, \varphi_1, \ldots,$ be a universal indexing, and let $\psi_0, \psi_1, \ldots,$ be any indexing with a total computable s-1-1 function, that is, a function s such that*

$$\psi_{s(i,1,x)}(y) = \psi_i(x, y)$$

for all $i, x, y \in \mathbb{N}$. Then, there is a total computable function t such that $\varphi_i = \psi_{t(i)}$.

Proof. Let φ_{univ} be a universal partial computable function for the indexing $\varphi_0, \varphi_1, \ldots$. Since $\psi_0, \psi_1, \ldots,$ is also an indexing, φ_{univ} occurs somewhere in the second list, and thus, there is some k such that $\varphi_{univ} = \psi_k$. Then, we have

$$\psi_{s(k,1,i)}(x) = \psi_k(i, x) = \varphi_{univ}(i, x) = \varphi_i(x),$$

for all $i, x \in \mathbb{N}$. Therefore, we can take the function t to be the function defined such that

$$t(i) = s(k, 1, i)$$

for all $i \in \mathbb{N}$. ☐

Using Theorem 10.2, if we have two acceptable indexings $\varphi_0, \varphi_1, \ldots,$ and $\psi_0, \psi_1, \ldots$, there exist total computable functions t and u such that

$$\varphi_i = \psi_{t(i)} \quad \text{and} \quad \psi_i = \varphi_{u(i)}$$

for all $i \in \mathbb{N}$. Also note that if the composition function c is primitive recursive, then any s-m-n function is primitive recursive, and the translation functions are primitive recursive. Actually, a stronger result can be shown. It can be shown that for any two acceptable indexings, there exist total computable *injective* and *surjective* translation functions. In other words, any two acceptable indexings are recursively isomorphic (Roger's isomorphism theorem). Next, we turn to algorithmically unsolvable, or *undecidable*, problems.

10.2 Undecidable Problems

We saw in Section 9.3 that the halting problem for RAM programs is undecidable. In this section, we take a slightly more general approach to study the undecidability of problems, and give some tools for resolving decidability questions.

First, we prove again the undecidability of the halting problem, but this time, for *any* indexing of the partial computable functions.

Theorem 10.3. *(Halting Problem, Abstract Version) Let $\psi_0, \psi_1, \ldots,$ be any indexing of the partial computable functions. Then, the function f defined such that*

$$f(x,y) = \begin{cases} 1 & \text{if } \psi_x(y) \text{ is defined,} \\ 0 & \text{if } \psi_x(y) \text{ is undefined,} \end{cases}$$

is not computable.

Proof. Assume that f is computable, and let g be the function defined such that

$$g(x) = f(x,x)$$

for all $x \in \mathbb{N}$. Then g is also computable. Let θ be the function defined such that

$$\theta(x) = \begin{cases} 0 & \text{if } g(x) = 0, \\ \text{undefined} & \text{if } g(x) = 1. \end{cases}$$

We claim that θ is not even partial computable. Observe that θ is such that

$$\theta(x) = \begin{cases} 0 & \text{if } \psi_x(x) \text{ is undefined,} \\ \text{undefined} & \text{if } \psi_x(x) \text{ is defined.} \end{cases}$$

If θ was partial computable, it would occur in the list as some ψ_i, and we would have

$$\theta(i) = \psi_i(i) = 0 \quad \text{iff} \quad \psi_i(i) \text{ is undefined,}$$

a contradiction. Therefore, f and g can't be computable. ∎

Observe that the proof of Theorem 10.3 does not use the fact that the indexing is universal or acceptable, and thus, the theorem holds for any indexing of the partial computable functions. The function g defined in the proof of Theorem 10.3 is the characteristic function of a set denoted as K, where

$$K = \{x \mid \psi_x(x) \text{ is defined}\}.$$

Given any set, X, for any subset, $A \subseteq X$, of X, recall that the *characteristic function, C_A (or χ_A), of A* is the function, $C_A \colon X \to \{0, 1\}$, defined so that, for all $x \in X$,

$$C_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A. \end{cases}$$

The set K is an example of a set which is not computable (or not recursive). Since this fact is quite important, we give the following definition:

Definition 10.2. A subset, A, of Σ^* (or a subset, A, of $\mathbb{N}$) is *computable*, or *recursive*,[1] or *decidable* iff its characteristic function, C_A, is a total computable function.

Using Definition 10.2, Theorem 10.3 can be restated as follows.

Proposition 10.4. *For any indexing $\varphi_0, \varphi_1, \ldots$ of the partial computable functions (over Σ^* or $\mathbb{N}$), the set $K = \{x \mid \varphi_x(x)$ is defined$\}$ is not computable (not recursive).*

Computable (recursive) sets allow us to define the concept of a decidable (or undecidable) problem. The idea is to generalize the situation described in Section 9.3 and Section 9.4, where a set of objects, the RAM programs, is encoded into a set of natural numbers, using a coding scheme.

Definition 10.3. Let C be a countable set of objects, and let P be a property of objects in C. We view P as the set

$$\{a \in C \mid P(a)\}.$$

A *coding-scheme* is an injective function $\# \colon C \to \mathbb{N}$ that assigns a unique code to each object in C. The property P is *decidable (relative to #)* iff the set $\{\#(a) \mid a \in C \text{ and } P(a)\}$ is computable (recursive). The property P is *undecidable (relative to #)* iff the set $\{\#(a) \mid a \in C \text{ and } P(a)\}$ is not computable (not recursive).

Observe that the decidability of a property P of objects in C depends upon the coding scheme #. Thus, if we are cheating in using a non-effective coding scheme, we may declare that a property is decidabe even though it is not decidable in some reasonable coding scheme. Consequently, we require a coding scheme # to be *effective* in the following sense. Given any

[1]Since 1996, the term *recursive* has been considered old-fashioned by many researchers, and the term *computable* has been used instead.

object $a \in C$, we can effectively (i.e.. algorithmically) determine its code $\#(a)$. Conversely, given any integer $n \in \mathbb{N}$, we should be able to tell effectively if n is the code of some object in C, and if so, to find this object. In practice, it is always possible to describe the objects in C as strings over some (possibly complex) alphabet Σ (sets of trees, graphs, etc). In such cases, the coding schemes are computable functions from Σ^* to $\mathbb{N} = \{a_1\}^*$.

For example, let $C = \mathbb{N} \times \mathbb{N}$, where the property P is the equality of the partial functions φ_x and φ_y. We can use the pairing function $\langle -, - \rangle$ as a coding function, and the problem is formally encoded as the computability (recursiveness) of the set

$$\{\langle x, y\rangle \mid x, y \in \mathbb{N},\, \varphi_x = \varphi_y\}.$$

In most cases, we don't even bother to describe the coding scheme explicitly, knowing that such a description is routine, although perhaps tedious.

We now show that most properties about programs (except the trivial ones) are undecidable. First, we show that it is undecidable whether a RAM program halts for every input. In other words, it is undecidable whether a procedure is an algorithm. We actually prove a more general fact.

Proposition 10.5. *For any acceptable indexing $\varphi_0, \varphi_1, \ldots$ of the partial computable functions, the set*

$$\mathrm{TOTAL} = \{x \mid \varphi_x \textit{ is a total function}\}$$

is not computable (not recursive).

Proof. The proof uses a technique known as reducibility. We try to reduce a set A known to be *noncomputable* (*nonrecursive*) to TOTAL via a computable function $f\colon A \to \mathrm{TOTAL}$, so that

$$x \in A \quad \text{iff} \quad f(x) \in \mathrm{TOTAL}.$$

If TOTAL were computable (recursive), its characteristic function g would be computable, and thus, the function $g \circ f$ would be computable, a contradiction, since A is assumed to be noncomputable (nonrecursive. In the present case, we pick $A = K$. To find the computable function $f\colon K \to \mathrm{TOTAL}$, we use the s-m-n Theorem. Let θ be the function defined below: for all $x, y \in \mathbb{N}$,

$$\theta(x, y) = \begin{cases} \varphi_x(x) & \text{if } x \in K, \\ \text{undefined} & \text{if } x \notin K. \end{cases}$$

Note that θ does not depend on y. The function θ is partial computable. Indeed, we have

$$\theta(x, y) = \varphi_x(x) = \varphi_{univ}(x, x).$$

Thus, θ has some index j, so that $\theta = \varphi_j$, and by the s-m-n Theorem, we have

$$\varphi_{s(j,1,x)}(y) = \varphi_j(x, y) = \theta(x, y).$$

Let f be the computable function defined such that

$$f(x) = s(j, 1, x)$$

for all $x \in \mathbb{N}$. Then, we have

$$\varphi_{f(x)}(y) = \begin{cases} \varphi_x(x) & \text{if } x \in K, \\ \text{undefined} & \text{if } x \notin K \end{cases}$$

for all $y \in \mathbb{N}$. Thus, observe that $\varphi_{f(x)}$ is a total function iff $x \in K$, that is,

$$x \in K \quad \text{iff} \quad f(x) \in \text{TOTAL},$$

where f is computable. As we explained earlier, this shows that TOTAL is not computable (not recursive). ☐

The above argument can be generalized to yield a result known as Rice's Theorem. Let $\varphi_0, \varphi_1, \ldots$ be any indexing of the partial computable functions, and let C be any set of partial computable functions. We define the set P_C as

$$P_C = \{x \in \mathbb{N} \mid \varphi_x \in C\}.$$

We can view C as a property of some of the partial computable functions. For example

$$C = \{\text{all total computable functions}\}.$$

We say that C is *nontrivial* if C is neither empty nor the set of all partial computable functions. Equivalently C is nontrivial iff $P_C \neq \emptyset$ and $P_C \neq \mathbb{N}$. We may think of P_C as the set of programs computing the functions in C.

Theorem 10.6. *(Rice's Theorem) For any acceptable indexing $\varphi_0, \varphi_1, \ldots$ of the partial computable functions, for any set C of partial computable functions, the set*

$$P_C = \{x \in \mathbb{N} \mid \varphi_x \in C\}$$

is not computable (not recursive) unless C is trivial.

Proof. Assume that C is nontrivial. A set is computable (recursive) iff its complement is computable (recursive) (the proof is trivial). Hence, we may assume that the totally undefined function is not in C, and since $C \neq \emptyset$, let ψ be some other function in C. We produce a computable function f such that

$$\varphi_{f(x)}(y) = \begin{cases} \psi(y) & \text{if } x \in K, \\ \text{undefined} & \text{if } x \notin K, \end{cases}$$

for all $y \in \mathbb{N}$. We get f by using the s-m-n Theorem. Let $\psi = \varphi_i$, and define θ as follows:

$$\theta(x, y) = \varphi_{univ}(i, y) + (\varphi_{univ}(x, x) - \varphi_{univ}(x, x)),$$

where $-$ is the primitive recursive function for truncated subtraction (monus). Clearly, θ is partial computable, and let $\theta = \varphi_j$. By the s-m-n Theorem, we have

$$\varphi_{s(j,1,x)}(y) = \varphi_j(x, y) = \theta(x, y)$$

for all $x, y \in \mathbb{N}$. Letting f be the computable function such that

$$f(x) = s(j, 1, x),$$

by definition of θ, we get

$$\varphi_{f(x)}(y) = \theta(x, y) = \begin{cases} \psi(y) & \text{if } x \in K, \\ \text{undefined} & \text{if } x \notin K. \end{cases}$$

Thus, f is the desired reduction function. Now, we have

$$x \in K \quad \text{iff} \quad f(x) \in P_C,$$

and thus, the characteristic function C_K of K is equal to $C_P \circ f$, where C_P is the characteristic function of P_C. Therefore, P_C is not computable (not recursive), since otherwise, K would be computable, a contradiction. $\square$

Rice's Theorem shows that all nontrivial properties of the input/output behavior of programs are undecidable!

The scenario to apply Rice's Theorem to a class C of partial functions is to show that *some partial computable function belongs to* C (C is not empty), *and that some partial computable function does not belong to* C (C is not all the partial computable functions). This demonstrates that C is nontrivial.

In particular, the following properties are undecidable.

Proposition 10.7. *The following properties of partial computable functions are undecidable.*

(a) A partial computable function is a constant function.

(b) Given any integer $y \in \mathbb{N}$, is y in the range of some partial computable function.

(c) Two partial computable functions φ_x and φ_y are identical. More precisely, the set $\{\langle x, y\rangle \mid \varphi_x = \varphi_y\}$ is not computable.

(d) A partial computable function φ_x is equal to a given partial computable function φ_a.

(e) A partial computable function yields output z on input y, for any given $y, z \in \mathbb{N}$.

(f) A partial computable function diverges for some input.

(g) A partial computable function diverges for all input.

The above proposition is left as an easy exercise. For example, in (a), we need to exhibit a constant (partial) computable function, such as $zero(n) = 0$, and a nonconstant (partial) computable function, such as the identity function (or $succ(n) = n + 1$).

A property may be undecidable although it is partially decidable. By partially decidable, we mean that there exists a computable function g that enumerates the set $P_C = \{x \mid \varphi_x \in C\}$. This means that there is a computable function g whose range is P_C. We say that P_C is *listable*, or *computably enumerable*, or *recursively enumerable*. Indeed, g provides a recursive enumeration of P_C, with possible repetitions. Listable sets are the object of the next Section.

10.3 Listable (Recursively Enumerable) Sets

Consider the set

$$A = \{x \in \mathbb{N} \mid \varphi_x(a) \text{ is defined}\},$$

where $a \in \mathbb{N}$ is any fixed natural number. By Rice's Theorem, A is not computable (not recursive); check this. We claim that A is the range of a computable function g. For this, we use the T-predicate. We produce a function which is actually primitive recursive. First, note that A is nonempty (why?), and let x_0 be any index in A. We define g by primitive recursion as follows:

$$\begin{aligned} g(0) &= x_0, \\ g(x+1) &= \begin{cases} \Pi_1(x) & \text{if } T(\Pi_1(x), a, \Pi_2(x)), \\ x_0 & \text{otherwise.} \end{cases} \end{aligned}$$

Since this type of argument is new, it is helpful to explain informally what g does. For every input x, the function g tries finitely many steps of a computation on input a of some partial computable function. The computation is given by $\Pi_2(x)$, and the partial function is given by $\Pi_1(x)$. Since Π_1 and Π_2 are projection functions, when x ranges over $\mathbb{N}$, both $\Pi_1(x)$ and $\Pi_2(x)$ also range over $\mathbb{N}$.

Such a process is called a *dovetailing* computation. Therefore all computations on input a for all partial computable functions will be tried, and the indices of the partial computable functions converging on input a will be selected. This type of argument will be used over and over again.

Definition 10.4. A subset X of $\mathbb{N}$ is *listable*, or *computably enumerable*, or *recursively enumerable*[2] iff either $X = \emptyset$, or X is the range of some total computable function (total recursive function). Similarly, a subset X of Σ^* is *listable* or *computably enumerable*, or *recursively enumerable* iff either $X = \emptyset$, or X is the range of some total computable function (total recursive function).

[2]Since 1996, the term *recursively enumerable* has been considered old-fashioned by many researchers, and the terms *listable* and *computably enumerable* have been used instead.

We will often abbreviate computably enumerable as *c.e*, (and recursively enumerable as *r.e.*). A computably enumerable set is sometimes called a *partially decidable* or *semidecidable* set.

Remark: It should be noted that the definition of a *listable set* (*r.e set* or *c.e. set*) given in Definition 10.4 is *different* from an earlier definition given in terms of acceptance by a Turing machine and it is by no means obvious that these two definitions are equivalent. This equivalence will be proved in Proposition 10.9 ((1) $\Longleftrightarrow$ (4)).

The following proposition relates computable sets and listable sets (recursive sets and recursively enumerable sets).

Proposition 10.8. *A set A is computable (recursive) iff both A and its complement $\overline{A}$ are listable (recursively enumerable).*

Proof. Assume that A is computable. Then, it is trivial that its complement is also computable. Hence, we only have to show that a computable set is listable. The empty set is listable by definition. Otherwise, let $y \in A$ be any element. Then, the function f defined such that

$$f(x) = \begin{cases} x & \text{iff } C_A(x) = 1, \\ y & \text{iff } C_A(x) = 0, \end{cases}$$

for all $x \in \mathbb{N}$ is computable and has range A.

Conversely, assume that both A and $\overline{A}$ are computably enumerable. If either A or $\overline{A}$ is empty, then A is computable. Otherwise, let $A = f(\mathbb{N})$ and $\overline{A} = g(\mathbb{N})$, for some computable functions f and g. We define the function C_A as follows:

$$C_A(x) = \begin{cases} 1 & \text{if } f(\min y[f(y) = x \ \vee \ g(y) = x]) = x, \\ 0 & \text{otherwise.} \end{cases}$$

The function C_A lists A and $\overline{A}$ in parallel, waiting to see whether x turns up in A or in $\overline{A}$. Note that x must eventually turn up either in A or in $\overline{A}$, so that C_A is a total computable function. $\square$

Our next goal is to show that the listable (recursively enumerable) sets can be given several equivalent definitions.

Proposition 10.9. *For any subset A of $\mathbb{N}$, the following properties are equivalent:*

(1) A is empty or A is the range of a primitive recursive function (Rosser, 1936).

(2) A is listable (recursively enumerable).

(3) A is the range of a partial computable function.

(4) A is the domain of a partial computable function.

Proof. The implication (1) $\Rightarrow$ (2) is trivial, since A is r.e. iff either it is empty or it is the range of a (total) computable function.

To prove the implication (2) $\Rightarrow$ (3), it suffices to observe that the empty set is the range of the totally undefined function (computed by an infinite loop program), and that a computable function is a partial computable function.

The implication (3) $\Rightarrow$ (4) is shown as follows. Assume that A is the range of φ_i. Define the function f such that

$$f(x) = \min y[T(i, \Pi_1(y), \Pi_2(y)) \ \wedge \ \mathrm{Res}(\Pi_2(y)) = x]$$

for all $x \in \mathbb{N}$. Clearly, f is partial computable and has domain A.

The implication (4) $\Rightarrow$ (1) is shown as follows. The only nontrivial case is when A is nonempty. Assume that A is the domain of φ_i. Since $A \neq \emptyset$, there is some $a \in \mathbb{N}$ such that $a \in A$, so the quantity

$$\min y[T(i, \Pi_1(y), \Pi_2(y))]$$

is defined and we can pick a to be

$$a = \Pi_1(\min y[T(i, \Pi_1(y), \Pi_2(y))]).$$

We define the primitive recursive function f as follows:

$$\begin{aligned} f(0) &= a, \\ f(x+1) &= \begin{cases} \Pi_1(x) & \text{if } T(i, \Pi_1(x), \Pi_2(x)), \\ a & \text{if } \neg T(i, \Pi_1(x), \Pi_2(x)). \end{cases} \end{aligned}$$

Clearly, A is the range of f and f is primitive recursive. □

More intuitive proofs of the implications (3) $\Rightarrow$ (4) and (4) $\Rightarrow$ (1) can be given. Assume that $A \neq \emptyset$ and that $A = \mathit{range}(g)$, where g is a partial computable function. Assume that g is computed by a RAM program P. To compute $f(x)$, we start computing the sequence

$$g(0), g(1), \ldots$$

looking for x. If x turns up as say $g(n)$, then we output n. Otherwise the computation diverges. Hence, the domain of f is the range of g.

Assume now that A is the domain of some partial computable function g, and that g is computed by some Turing machine M. Since the case where $A = \emptyset$ is trivial, we may assume that $A \neq \emptyset$, and let $n_0 \in A$ be some chosen element in A. We construct another Turing machine performing the following steps: On input n,

(0) Do one step of the computation of $g(0)$

$\ldots$

(n) Do $n+1$ steps of the computation of $g(0)$
Do n steps of the computation of $g(1)$
...
Do 2 steps of the computation of $g(n-1)$
Do 1 step of the computation of $g(n)$

During this process, whenever the computation of $g(m)$ halts for some $m \leq n$, we output m. Otherwise, we output n_0.

In this fashion, we will enumerate the domain of g, and since we have constructed a Turing machine that halts for every input, we have a total computable function.

The following proposition can easily be shown using the proof technique of Proposition 10.9.

Proposition 10.10. *(1) There is a computable function h such that*

$$range(\varphi_x) = dom(\varphi_{h(x)})$$

for all $x \in \mathbb{N}$.

(2) There is a computable function k such that

$$dom(\varphi_x) = range(\varphi_{k(x)})$$

and $\varphi_{k(x)}$ is total computable, for all $x \in \mathbb{N}$ such that $dom(\varphi_x) \neq \emptyset$.

The proof of Proposition 10.10 is left as an exercise. Using Proposition 10.9, we can prove that K is a listable set. Indeed, we have $K = dom(f)$, where

$$f(x) = \varphi_{univ}(x, x)$$

for all $x \in \mathbb{N}$. The set

$$K_0 = \{\langle x, y\rangle \mid \varphi_x(y) \text{ converges}\}$$

is also a listable set, since $K_0 = dom(g)$, where

$$g(z) = \varphi_{univ}(\Pi_1(z), \Pi_2(z)),$$

which is partial computable. It worth recording these facts in the following lemma.

Proposition 10.11. *The sets K and K_0 are listable sets that are not computable (r.e. sets that are not recursive).*

We can now prove that there are sets that are not c.e. (r.e.).

Proposition 10.12. *For any indexing of the partial computable functions, the complement $\overline{K}$ of the set*

$$K = \{x \in \mathbb{N} \mid \varphi_x(x) \ \textit{converges}\}$$

is not listable (not recursively enumerable).

Proof. If $\overline{K}$ was listable, since K is also listable, by Proposition 10.8, the set K would be computable, a contradiction. ☐

The sets $\overline{K}$ and $\overline{K_0}$ are examples of sets that are not c.e. (r.e.). This shows that the c.e. sets (r.e. sets) are not closed under complementation. However, we leave it as an exercise to prove that the c.e. sets (r.e. sets) are closed under union and intersection.

We will prove later on that TOTAL is not c.e. (r.e.). This is rather unpleasant. Indeed, this means that there is no way of effectively listing all algorithms (all total computable functions). Hence, in a certain sense, the concept of partial computable function (procedure) is more natural than the concept of a (total) computable function (algorithm).

The next two propositions give other characterizations of the c.e. sets (r.e. sets) and of the computable sets (recursive sets). The proofs are left as an exercise.

Proposition 10.13. *(1) A set A is c.e. (r.e.) iff either it is finite or it is the range of an injective computable function.*

(2) A set A is c.e. (r.e.) if either it is empty or it is the range of a monotonic partial computable function.

(3) A set A is c.e. (r.e.) iff there is a Turing machine M such that, for all $x \in \mathbb{N}$, M halts on x iff $x \in A$.

Proposition 10.14. *A set A is computable (recursive) iff either it is finite or it is the range of a strictly increasing computable function.*

Another important result relating the concept of partial computable function and that of a c.e. set (r.e. set) is given below.

Theorem 10.15. *For every unary partial function f, the following properties are equivalent:*

(1) f is partial computable.

(2) The set

$$\{\langle x, f(x)\rangle \mid x \in \mathit{dom}(f)\}$$

is c.e. (r.e.).

Proof. Let $g(x) = \langle x, f(x)\rangle$. Clearly, g is partial computable, and

$$\mathit{range}(g) = \{\langle x, f(x)\rangle \mid x \in \mathit{dom}(f)\}.$$

Conversely, assume that

$$range(g) = \{\langle x, f(x)\rangle \mid x \in dom(f)\}$$

for some computable function g. Then, we have

$$f(x) = \Pi_2(g(\min y[\Pi_1(g(y)) = x]))$$

for all $x \in \mathbb{N}$, so that f is partial computable. $\square$

Using our indexing of the partial computable functions and Proposition 10.9, we obtain an indexing of the c.e. sets. (r.e. sets).

Definition 10.5. For any acceptable indexing $\varphi_0, \varphi_1, \ldots$ of the partial computable functions, we define the enumeration $W_0, W_1, \ldots$ of the c.e. sets (r.e. sets) by setting

$$W_x = dom(\varphi_x).$$

We now describe a technique for showing that certain sets are c.e. (r.e.) but not computable (not recursive), or complements of c.e. sets (r.e. sets) that are not computable (not recursive), or not c.e. (not r.e.), or neither c.e. (r.e.) nor the complement of a c.e. set (r.e. set). This technique is known as *reducibility*.

10.4 Reducibility and Complete Sets

We already used the notion of reducibility in the proof of Proposition 10.5 to show that TOTAL is not computable (not recursive).

Definition 10.6. Let A and B be subsets of $\mathbb{N}$ (or Σ^*). We say that the set A is *many-one reducible* to the set B if there is a *total computable* function (or *total recursive* function) $f\colon \mathbb{N} \to \mathbb{N}$ (or $f\colon \Sigma^* \to \Sigma^*$) such that

$$x \in A \quad \text{iff} \quad f(x) \in B \quad \text{for all } x \in \mathbb{N}.$$

We write $A \leq B$, and for short, we say that A is *reducible* to B. Sometimes, the notation $A \leq_m B$ is used to stress that this is a many-to-one reduction (that is, f is not necessarily injective).

Intuitively, deciding membership in B is as hard as deciding membership in A. This is because any method for deciding membership in B can be converted to a method for deciding membership in A by first applying f to the number (or string) to be tested.

The following simple proposition is left as an exercise to the reader.

Proposition 10.16. *Let A, B, C be subsets of $\mathbb{N}$ (or Σ^*). The following properties hold:*

(1) If $A \leq B$ and $B \leq C$, then $A \leq C$.

(2) If $A \leq B$ then $\overline{A} \leq \overline{B}$.

(3) If $A \leq B$ and B is c.e., then A is c.e.

(4) If $A \leq B$ and A is not c.e., then B is not c.e.

(5) If $A \leq B$ and B is computable, then A is computable.

(6) If $A \leq B$ and A is not computable, then B is not computable.

Another important concept is the concept of a complete set.

Definition 10.7. A c.e. set (r.e. set) A is *complete w.r.t. many-one reducibility* iff every c.e. set (r.e. set) B is reducible to A, i.e., $B \leq A$.

For simplicity, we will often say *complete* for *complete w.r.t. many-one reducibility*. Intuitively, a complete c.e. set (r.e. set) is a "hardest" c.e. set (r.e. set) as far as membership is concerned.

Theorem 10.17. *The following properties hold:*

(1) If A is complete, B is c.e (r.e.), and $A \leq B$, then B is complete.

(2) K_0 is complete.

(3) K_0 is reducible to K. Consequently, K is also complete.

Proof. (1) This is left as a simple exercise.

(2) Let W_x be any c.e. set. Then

$$y \in W_x \quad \text{iff} \quad \langle x, y \rangle \in K_0,$$

and the reduction function is the computable function f such that

$$f(y) = \langle x, y \rangle$$

for all $y \in \mathbb{N}$.

(3) We use the s-m-n Theorem. First, we leave it as an exercise to prove that there is a computable function f such that

$$\varphi_{f(x)}(y) = \begin{cases} 1 & \text{if } \varphi_{\Pi_1(x)}(\Pi_2(x)) \text{ converges,} \\ \text{undefined} & \text{otherwise,} \end{cases}$$

for all $x, y \in \mathbb{N}$. Then, for every $z \in \mathbb{N}$,

$$z \in K_0 \quad \text{iff} \quad \varphi_{\Pi_1(z)}(\Pi_2(z)) \text{ converges,}$$

iff $\varphi_{f(z)}(y) = 1$ for all $y \in \mathbb{N}$. However,

$$\varphi_{f(z)}(y) = 1 \quad \text{iff} \quad \varphi_{f(z)}(f(z)) = 1,$$

since $\varphi_{f(z)}$ is a constant function. This means that

$$z \in K_0 \quad \text{iff} \quad f(z) \in K,$$

and f is the desired function. □

As a corollary of Theorem 10.17, the set K is also complete.

Definition 10.8. Two sets A and B have the same *degree of unsolvability* or are *equivalent* iff $A \leq B$ and $B \leq A$.

Since K and K_0 are both complete, they have the same degree of unsolvability. We will now investigate the reducibility and equivalence of various sets. Recall that

$$\text{TOTAL} = \{x \in \mathbb{N} \mid \varphi_x \text{ is total}\}.$$

We define EMPTY and FINITE, as follows:

$$\begin{aligned} \text{EMPTY} &= \{x \in \mathbb{N} \mid \varphi_x \text{ is undefined for all input}\}, \\ \text{FINITE} &= \{x \in \mathbb{N} \mid \varphi_x \text{ is defined only for finitely many input}\}. \end{aligned}$$

Obviously, EMPTY $\subset$ FINITE, and since

$$\text{FINITE} = \{x \in \mathbb{N} \mid \varphi_x \text{ has a finite domain}\},$$

we have

$$\overline{\text{FINITE}} = \{x \in \mathbb{N} \mid \varphi_x \text{ has an infinite domain}\},$$

and thus, TOTAL $\subset \overline{\text{FINITE}}$.

Proposition 10.18. *We have* $K_0 \leq \overline{\text{EMPTY}}$.

The proof of Proposition 10.18 follows from the proof of Theorem 10.17. We also have the following proposition.

Proposition 10.19. *The following properties hold:*

(1) EMPTY *is not c.e. (not r.e.).*

(2) $\overline{\text{EMPTY}}$ *is c.e. (r.e.).*

(3) $\overline{K}$ *and* EMPTY *are equivalent.*

(4) $\overline{\text{EMPTY}}$ *is complete.*

Proof. We prove (1) and (3), leaving (2) and (4) as an exercise (Actually, (2) and (4) follow easily from (3)). First, we show that $\overline{K} \leq$ EMPTY. By the s-m-n Theorem, there exists a computable function f such that

$$\varphi_{f(x)}(y) = \begin{cases} \varphi_x(x) & \text{if } \varphi_x(x) \text{ converges,} \\ \text{undefined} & \text{if } \varphi_x(x) \text{ diverges,} \end{cases}$$

for all $x, y \in \mathbb{N}$. Note that for all $x \in \mathbb{N}$,

$$x \in \overline{K} \quad \text{iff} \quad f(x) \in \text{EMPTY},$$

and thus, $\overline{K} \leq$ EMPTY. Since $\overline{K}$ is not c.e., EMPTY is not c.e.

By the s-m-n Theorem, there is a computable function g such that

$$\varphi_{g(x)}(y) = \min z[T(x, \Pi_1(z), \Pi_2(z))],$$

for all $x, y \in \mathbb{N}$. Note that

$$x \in \text{EMPTY} \quad \text{iff} \quad g(x) \in \overline{K}$$

for all $x \in \mathbb{N}$. Therefore, EMPTY $\leq \overline{K}$, and since we just showed that $\overline{K} \leq$ EMPTY, the sets $\overline{K}$ and EMPTY are equivalent. ☐

Proposition 10.20. *The following properties hold:*

(1) TOTAL *and* $\overline{\text{TOTAL}}$ *are not c.e. (not r.e.).*

(2) FINITE *and* $\overline{\text{FINITE}}$ *are not c.e (not r.e.).*

Proof. Checking the proof of Theorem 10.17, we note that $K_0 \leq$ TOTAL and $K_0 \leq \overline{\text{FINITE}}$. Hence, we get $\overline{K_0} \leq \overline{\text{TOTAL}}$ and $\overline{K_0} \leq$ FINITE, and neither $\overline{\text{TOTAL}}$ nor FINITE is c.e. If TOTAL was c.e., then there would be a computable function f such that TOTAL $=$ $range(f)$. Define g as follows:

$$g(x) = \varphi_{f(x)}(x) + 1 = \varphi_{univ}(f(x), x) + 1$$

for all $x \in \mathbb{N}$. Since f is total and $\varphi_{f(x)}$ is total for all $x \in \mathbb{N}$, the function g is total computable. Let e be an index such that

$$g = \varphi_{f(e)}.$$

Since g is total, $g(e)$ is defined. Then, we have

$$g(e) = \varphi_{f(e)}(e) + 1 = g(e) + 1,$$

a contradiction. Hence, TOTAL is not c.e. Finally, we show that TOTAL $\leq \overline{\text{FINITE}}$. This also shows that $\overline{\text{FINITE}}$ is not c.e. By the s-m-n Theorem, there is a computable function f such that

$$\varphi_{f(x)}(y) = \begin{cases} 1 & \text{if } \forall z \leq y(\varphi_x(z)\downarrow), \\ \text{undefined} & \text{otherwise,} \end{cases}$$

for all $x, y \in \mathbb{N}$. It is easily seen that

$$x \in \text{TOTAL} \quad \text{iff} \quad f(x) \in \overline{\text{FINITE}}$$

for all $x \in \mathbb{N}$. ☐

From Proposition 10.20, we have TOTAL $\leq \overline{\text{FINITE}}$. It turns out that $\overline{\text{FINITE}} \leq$ TOTAL, and TOTAL and $\overline{\text{FINITE}}$ are equivalent.

Proposition 10.21. *The sets* TOTAL *and* $\overline{\text{FINITE}}$ *are equivalent.*

Proof. We show that $\overline{\text{FINITE}} \leq$ TOTAL. By the s-m-n Theorem, there is a computable function f such that

$$\varphi_{f(x)}(y) = \begin{cases} 1 & \text{if } \exists z \geq y(\varphi_x(z) \downarrow), \\ \text{undefined} & \text{if } \forall z \geq y(\varphi_x(z) \uparrow), \end{cases}$$

for all $x, y \in \mathbb{N}$. It is easily seen that

$$x \in \overline{\text{FINITE}} \quad \text{iff} \quad f(x) \in \text{TOTAL}$$

for all $x \in \mathbb{N}$. ☐

We now turn to the recursion Theorem.

10.5 The Recursion Theorem

The recursion Theorem, due to Kleene, is a fundamental result in recursion theory. Let f be a total computable function. Then, it turns out that there is some n such that

$$\varphi_n = \varphi_{f(n)}.$$

Theorem 10.22. *(Recursion Theorem, Version 1) Let $\varphi_0, \varphi_1, \ldots$ be any acceptable indexing of the partial computable functions. For every total computable function f, there is some n such that*

$$\varphi_n = \varphi_{f(n)}.$$

Proof. Consider the function θ defined such that

$$\theta(x, y) = \varphi_{univ}(\varphi_{univ}(x, x), y)$$

for all $x, y \in \mathbb{N}$. The function θ is partial computable, and there is some index j such that $\varphi_j = \theta$. By the s-m-n Theorem, there is a computable function g such that

$$\varphi_{g(x)}(y) = \theta(x, y).$$

Consider the function $f \circ g$. Since it is computable, there is some index m such that $\varphi_m = f \circ g$. Let

$$n = g(m).$$

Since φ_m is total, $\varphi_m(m)$ is defined, and we have

$$\begin{aligned}\varphi_n(y) &= \varphi_{g(m)}(y) = \theta(m, y) = \varphi_{univ}(\varphi_{univ}(m, m), y) = \varphi_{\varphi_{univ}(m,m)}(y) \\ &= \varphi_{\varphi_m(m)}(y) = \varphi_{f \circ g(m)}(y) = \varphi_{f(g(m))}(y) = \varphi_{f(n)}(y),\end{aligned}$$

for all $y \in \mathbb{N}$. Therefore, $\varphi_n = \varphi_{f(n)}$, as desired. ☐

The recursion Theorem can be strengthened as follows.

Theorem 10.23. *(Recursion Theorem, Version 2) Let $\varphi_0, \varphi_1, \ldots$ be any acceptable indexing of the partial computable functions. There is a total computable function h such that for all $x \in \mathbb{N}$, if φ_x is total, then*

$$\varphi_{\varphi_x(h(x))} = \varphi_{h(x)}.$$

Proof. The computable function g obtained in the proof of Theorem 10.22 satisfies the condition

$$\varphi_{g(x)} = \varphi_{\varphi_x(x)},$$

and it has some index i such that $\varphi_i = g$. Recall that c is a computable composition function such that

$$\varphi_{c(x,y)} = \varphi_x \circ \varphi_y.$$

It is easily verified that the function h defined such that

$$h(x) = g(c(x, i))$$

for all $x \in \mathbb{N}$ does the job. ☐

A third version of the recursion Theorem is given below.

Theorem 10.24. *(Recursion Theorem, Version 3) For all $n \geq 1$, there is a total computable function h of $n+1$ arguments, such that for all $x \in \mathbb{N}$, if φ_x is a total computable function of $n+1$ arguments, then*

$$\varphi_{\varphi_x(h(x,x_1,\ldots,x_n),x_1,\ldots,x_n)} = \varphi_{h(x,x_1,\ldots,x_n)},$$

for all $x_1, \ldots, x_n \in \mathbb{N}$.

Proof. Let θ be the function defined such that

$$\theta(x, x_1, \ldots, x_n, y) = \varphi_{\varphi_x(x,x_1,\ldots,x_n)}(y) = \varphi_{univ}(\varphi_{univ}(x, x, x_1, \ldots, x_n), y)$$

for all $x, x_1, \ldots, x_n, y \in \mathbb{N}$. By the s-m-n Theorem, there is a computable function g such that

$$\varphi_{g(x,x_1,\ldots,x_n)} = \varphi_{\varphi_x(x,x_1,\ldots,x_n)}.$$

It is easily shown that there is a computable function c such that

$$\varphi_{c(i,j)}(x, x_1, \ldots, x_n) = \varphi_i(\varphi_j(x, x_1, \ldots, x_n), x_1, \ldots, x_n)$$

for any two partial computable functions φ_i and φ_j (viewed as functions of $n+1$ arguments) and all $x, x_1, \ldots, x_n \in \mathbb{N}$. Let $\varphi_i = g$, and define h such that

$$h(x, x_1, \ldots, x_n) = g(c(x, i), x_1, \ldots, x_n),$$

for all $x, x_1, \ldots, x_n \in \mathbb{N}$. We have

$$\varphi_{h(x,x_1,\ldots,x_n)} = \varphi_{g(c(x,i),x_1,\ldots,x_n)} = \varphi_{\varphi_{c(x,i)}(c(x,i),x_1,\ldots,x_n)},$$

and

$$\begin{aligned}\varphi_{\varphi_{c(x,i)}(c(x,i),x_1,\ldots,x_n)} &= \varphi_{\varphi_x(\varphi_i(c(x,i),x_1,\ldots,x_n),x_1,\ldots,x_n)},\\ &= \varphi_{\varphi_x(g(c(x,i),x_1,\ldots,x_n),x_1,\ldots,x_n)},\\ &= \varphi_{\varphi_x(h(x,x_1,\ldots,x_n),x_1,\ldots,x_n)}.\end{aligned}$$

□

As a first application of the recursion theorem, we can show that there is an index n such that φ_n is the constant function with output n. Loosely speaking, φ_n prints its own name. Let f be the computable function such that

$$f(x, y) = x$$

for all $x, y \in \mathbb{N}$. By the s-m-n Theorem, there is a computable function g such that

$$\varphi_{g(x)}(y) = f(x, y) = x$$

for all $x, y \in \mathbb{N}$. By the recursion Theorem 10.22, there is some n such that

$$\varphi_{g(n)} = \varphi_n,$$

the constant function with value n.

As a second application, we get a very short proof of Rice's Theorem. Let C be such that $P_C \neq \emptyset$ and $P_C \neq \mathbb{N}$, and let $j \in P_C$ and $k \in \mathbb{N} - P_C$. Define the function f as follows:

$$f(x) = \begin{cases} j & \text{if } x \notin P_C,\\ k & \text{if } x \in P_C,\end{cases}$$

If P_C is computable, then f is computable. By the recursion Theorem 10.22, there is some n such that

$$\varphi_{f(n)} = \varphi_n.$$

But then, we have

$$n \in P_C \quad \text{iff} \quad f(n) \notin P_C$$

by definition of f, and thus,

$$\varphi_{f(n)} \neq \varphi_n,$$

a contradiction. Hence, P_C is not computable.

As a third application, we prove the following proposition.

Proposition 10.25. *Let C be a set of partial computable functions and let*

$$A = \{x \in \mathbb{N} \mid \varphi_x \in C\}.$$

The set A is not reducible to its complement $\overline{A}$.

Proof. Assume that $A \leq \overline{A}$. Then, there is a computable function f such that

$$x \in A \quad \text{iff} \quad f(x) \in \overline{A}$$

for all $x \in \mathbb{N}$. By the recursion Theorem, there is some n such that

$$\varphi_{f(n)} = \varphi_n.$$

But then,

$$\varphi_n \in C \quad \text{iff} \quad n \in A \quad \text{iff} \quad f(n) \in \overline{A} \quad \text{iff} \quad \varphi_{f(n)} \in \overline{C},$$

contradicting the fact that

$$\varphi_{f(n)} = \varphi_n.$$

∎

The recursion Theorem can also be used to show that functions defined by recursive definitions other than primitive recursion are partial computable. This is the case for the function known as *Ackermann's function*, defined recursively as follows:

$$\begin{aligned} f(0, y) &= y + 1, \\ f(x + 1, 0) &= f(x, 1), \\ f(x + 1, y + 1) &= f(x, f(x + 1, y)). \end{aligned}$$

It can be shown that this function is not primitive recursive. Intuitively, it outgrows all primitive recursive functions. However, f is computable, but this is not so obvious. We can

use the recursion Theorem to prove that f is computable. Consider the following definition by cases:

$$\begin{aligned} g(n, 0, y) &= y + 1, \\ g(n, x + 1, 0) &= \varphi_{univ}(n, x, 1), \\ g(n, x + 1, y + 1) &= \varphi_{univ}(n, x, \varphi_{univ}(n, x + 1, y)). \end{aligned}$$

Clearly, g is partial computable. By the s-m-n Theorem, there is a computable function h such that

$$\varphi_{h(n)}(x, y) = g(n, x, y).$$

By the recursion Theorem, there is an m such that

$$\varphi_{h(m)} = \varphi_m.$$

Therefore, the partial computable function $\varphi_m(x, y)$ satisfies the definition of Ackermann's function. We showed in a previous Section that $\varphi_m(x, y)$ is a total function, and thus, Ackermann's function is a total computable function.

Hence, the recursion Theorem justifies the use of certain recursive definitions. However, note that there are some recursive definitions that are only satisfied by the completely undefined function.

In the next Section, we prove the extended Rice Theorem.

10.6 Extended Rice Theorem

The extended Rice Theorem characterizes the sets of partial computable functions C such that P_C is c.e. (r.e.). First, we need to discuss a way of indexing the partial computable functions that have a finite domain. Using the uniform projection function Π, we define the primitive recursive function F such that

$$F(x, y) = \Pi(y + 1, \Pi_1(x) + 1, \Pi_2(x)).$$

We also define the sequence of partial functions $P_0, P_1, \ldots$ as follows:

$$P_x(y) = \begin{cases} F(x, y) - 1 & \text{if } 0 < F(x, y) \text{ and } y < \Pi_1(x) + 1, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Proposition 10.26. *Every P_x is a partial computable function with finite domain, and every partial computable function with finite domain is equal to some P_x.*

The proof is left as an exercise. The easy part of the extended Rice Theorem is the following lemma. Recall that given any two partial functions $f\colon A \to B$ and $g\colon A \to B$, we say that g *extends* f iff $f \subseteq g$, which means that $g(x)$ is defined whenever $f(x)$ is defined, and if so, $g(x) = f(x)$.

Proposition 10.27. *Let C be a set of partial computable functions. If there is a c.e. set (r.e. set) A such that, $\varphi_x \in C$ iff there is some $y \in A$ such that φ_x extends P_y, then $P_C = \{x \mid \varphi_x \in C\}$ is c.e. (r.e.).*

Proof. Proposition 10.27 can be restated as

$$P_C = \{x \mid \exists y \in A,\, P_y \subseteq \varphi_x\}$$

is c.e. If A is empty, so is P_C, and P_C is c.e. Otherwise, let f be a computable function such that

$$A = range(f).$$

Let ψ be the following partial computable function:

$$\psi(z) = \begin{cases} \Pi_1(z) & \text{if } P_{f(\Pi_2(z))} \subseteq \varphi_{\Pi_1(z)}, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

It is clear that

$$P_C = range(\psi).$$

To see that ψ is partial computable, write $\psi(z)$ as follows:

$$\psi(z) = \begin{cases} \Pi_1(z) & \text{if } \forall w \leq \Pi_1(f(\Pi_2(z)))[F(f(\Pi_2(z)), w) > 0 \\ & \qquad \supset \varphi_{\Pi_1(z)}(w) = F(f(\Pi_2(z)), w) - 1], \\ \text{undefined} & \text{otherwise.} \end{cases}$$

∎

To establish the converse of Proposition 10.27, we need two propositions.

Proposition 10.28. *If P_C is c.e. (r.e.) and $\varphi \in C$, then there is some $P_y \subseteq \varphi$ such that $P_y \in C$.*

Proof. Assume that P_C is c.e. and that $\varphi \in C$. By an s-m-n construction, there is a computable function g such that

$$\varphi_{g(x)}(y) = \begin{cases} \varphi(y) & \text{if } \forall z \leq y[\neg T(x, x, z)], \\ \text{undefined} & \text{if } \exists z \leq y[T(x, x, z)], \end{cases}$$

for all $x, y \in \mathbb{N}$. Observe that if $x \in K$, then $\varphi_{g(x)}$ is a finite subfunction of φ, and if $x \in \overline{K}$, then $\varphi_{g(x)} = \varphi$. Assume that no finite subfunction of φ is in C. Then,

$$x \in \overline{K} \quad \text{iff} \quad g(x) \in P_C$$

for all $x \in \mathbb{N}$, that is, $\overline{K} \leq P_C$. Since P_C is c.e., $\overline{K}$ would also be c.e., a contradiction. ∎

As a corollary of Proposition 10.28, we note that TOTAL is not c.e.

Proposition 10.29. *If P_C is c.e. (r.e.), $\varphi \in C$, and $\varphi \subseteq \psi$, where ψ is a partial computable function, then $\psi \in C$.*

Proof. Assume that P_C is c.e. We claim that there is a computable function h such that

$$\varphi_{h(x)}(y) = \begin{cases} \psi(y) & \text{if } x \in K, \\ \varphi(y) & \text{if } x \in \overline{K}, \end{cases}$$

for all $x, y \in \mathbb{N}$. Assume that $\psi \notin C$. Then

$$x \in \overline{K} \quad \text{iff} \quad h(x) \in P_C$$

for all $x \in \mathbb{N}$, that is, $\overline{K} \leq P_C$, a contradiction, since P_C is c.e. Therefore, $\psi \in C$. To find the function h we proceed as follows: Let $\varphi = \varphi_j$ and define Θ such that

$$\Theta(x, y, z) = \begin{cases} \varphi(y) & \text{if } T(j, y, z) \wedge \neg T(x, y, w), \text{ for } 0 \leq w < z \\ \psi(y) & \text{if } T(x, x, z) \wedge \neg T(j, y, w), \text{ for } 0 \leq w < z \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Observe that if $x = y = j$, then $\Theta(j, j, z)$ is multiply defined, but since ψ extends φ, we get the same value $\psi(y) = \varphi(y)$, so Θ is a well defined partial function. Clearly, for all $(m, n) \in \mathbb{N}^2$, there is at most one $z \in \mathbb{N}$ so that $\Theta(x, y, z)$ is defined, so the function σ defined by

$$\sigma(x, y) = \begin{cases} z & \text{if } (x, y, z) \in \text{dom}(\Theta) \\ \text{undefined} & \text{otherwise} \end{cases}$$

is a partial computable function. Finally, let

$$\theta(x, y) = \Theta(x, y, \sigma(x, y)),$$

a partial computable function. It is easy to check that

$$\theta(x, y) = \begin{cases} \psi(y) & \text{if } x \in K, \\ \varphi(y) & \text{if } x \in \overline{K}, \end{cases}$$

for all $x, y \in \mathbb{N}$. By the s-m-n Theorem, there is a computable function h such that

$$\varphi_{h(x)}(y) = \theta(x, y)$$

for all $x, y \in \mathbb{N}$. ∎

Observe that Proposition 10.29 yields a new proof that $\overline{\text{TOTAL}}$ is not c.e. (not r.e.). Finally, we can prove the extended Rice Theorem.

Theorem 10.30. *(Extended Rice Theorem) The set P_C is c.e. (r.e.) iff there is a c.e. set (r.e. set) A such that*

$$\varphi_x \in C \quad \textit{iff} \quad \exists y \in A\, (P_y \subseteq \varphi_x).$$

Proof. Let $P_C = dom(\varphi_i)$. Using the s-m-n Theorem, there is a computable function k such that

$$\varphi_{k(y)} = P_y$$

for all $y \in \mathbb{N}$. Define the c.e. set A such that

$$A = dom(\varphi_i \circ k).$$

Then,

$$y \in A \quad \text{iff} \quad \varphi_i(k(y)) \downarrow \quad \text{iff} \quad P_y \in C.$$

Next, using Proposition 10.28 and Proposition 10.29, it is easy to see that

$$\varphi_x \in C \quad \text{iff} \quad \exists y \in A\, (P_y \subseteq \varphi_x).$$

Indeed, if $\varphi_x \in C$, by Proposition 10.28, there is a finite subfunction $P_y \subseteq \varphi_x$ such that $P_y \in C$, but

$$P_y \in C \quad \text{iff} \quad y \in A,$$

as desired. On the other hand, if

$$P_y \subseteq \varphi_x$$

for some $y \in A$, then

$$P_y \in C,$$

and by Proposition 10.29, since φ_x extends P_y, we get

$$\varphi_x \in C.$$

∎

10.7 Creative and Productive Sets

In this section, we discuss some special sets that have important applications in logic: creative and productive sets. The concepts to be described are illustrated by the following situation. Assume that

$$W_x \subseteq \overline{K}$$

for some $x \in \mathbb{N}$. We claim that

$$x \in \overline{K} - W_x.$$

Indeed, if $x \in W_x$, then $\varphi_x(x)$ is defined, and by definition of K, we get $x \notin \overline{K}$, a contradiction. Therefore, $\varphi_x(x)$ must be undefined, that is,

$$x \in \overline{K} - W_x.$$

The above situation can be generalized as follows.

Definition 10.9. A set A is *productive* iff there is a total computable function f such that

$$\text{if} \quad W_x \subseteq A \quad \text{then} \quad f(x) \in A - W_x$$

for all $x \in \mathbb{N}$. The function f is called the *productive function of* A. A set A is *creative* if it is c.e (r.e.) and if its complement $\overline{A}$ is productive.

As we just showed, K is creative and $\overline{K}$ is productive. The following facts are immediate conequences of the definition.

(1) A productive set is not c.e. (r.e.).

(2) A creative set is not computable (not recursive).

Creative and productive sets arise in logic. The set of theorems of a logical theory is often creative. For example, the set of theorems in Peano's arithmetic is creative. This yields incompleteness results.

Proposition 10.31. *If a set A is productive, then it has an infinite c.e. (r.e.) subset.*

Proof. We first give an informal proof. let f be the computable productive function of A. We define a computable function g as follows: Let x_0 be an index for the empty set, and let

$$g(0) = f(x_0).$$

Assuming that

$$\{g(0), g(1), \ldots, g(y)\}$$

is known, let x_{y+1} be an index for this finite set, and let

$$g(y + 1) = f(x_{y+1}).$$

Since $W_{x_{y+1}} \subseteq A$, we have $f(x_{y+1}) \in A$.

For the formal proof, we use the following facts whose proof is left as an exercise:

(1) There is a computable function u such that

$$W_{u(x,y)} = W_x \cup W_y.$$

(2) There is a computable function t such that

$$W_{t(x)} = \{x\}.$$

Letting x_0 be an index for the empty set, we define the function h as follows:

$$h(0) = x_0,$$
$$h(y + 1) = u(t(f(y)), h(y)).$$

We define g such that

$$g = f \circ h.$$

It is easily seen that g does the job. ☐

Another important property of productive sets is the following.

Proposition 10.32. *If a set A is productive, then $\overline{K} \leq A$.*

Proof. Let f be a productive function for A. Using the s-m-n Theorem, we can find a computable function h such that

$$W_{h(y,x)} = \begin{cases} \{f(y)\} & \text{if } x \in K, \\ \emptyset & \text{if } x \in \overline{K}. \end{cases}$$

The above can be restated as follows:

$$\varphi_{h(y,x)}(z) = \begin{cases} 1 & \text{if } x \in K \text{ and } z = f(y), \\ \text{undefined} & \text{if } x \in \overline{K}, \end{cases}$$

for all $x, y, z \in \mathbb{N}$. By the third version of the recursion Theorem (Theorem 10.24), there is a computable function g such that

$$W_{g(x)} = W_{h(g(x),x)}$$

for all $x \in \mathbb{N}$. Let

$$k = f \circ g.$$

We claim that

$$x \in \overline{K} \quad \text{iff} \quad k(x) \in A$$

for all $x \in \mathbb{N}$. The verification of this fact is left as an exercise. Thus, $\overline{K} \leq A$. ☐

Using Proposition 10.32, the following results can be shown.

Proposition 10.33. *The following facts hold.*

(1) If A is productive and $A \leq B$, then B is productive.

(2) A is creative iff A is equivalent to K.

(3) A is creative iff A is complete,

Chapter 11

Listable Sets and Diophantine Sets; Hilbert's Tenth Problem

11.1 Diophantine Equations and Hilbert's Tenth Problem

There is a deep and a priori unexpected connection between the theory of computable and listable sets and the solutions of polynomial equations involving polynomials in several variables with integer coefficients. These are polynomials in $n \geq 1$ variables $x_1, \ldots, x_n$ which are finite sums of *monomials* of the form

$$ax_1^{k_1} \cdots x_n^{k_n},$$

where $k_1, \ldots, k_n \in \mathbb{N}$ are nonnegative integers, and $a \in \mathbb{Z}$ is an integer (possibly negative). The natural number $k_1 + \cdots + k_n$ is called the *degree* of the monomial $ax_1^{k_1} \cdots x_n^{k_n}$.

For example, if $n = 3$, then

1. $5,\ -7$, are monomials of degree 0.
2. $3x_1,\ -2x_2$, are monomials of degree 1.
3. $x_1x_2,\ 2x_1^2,\ 3x_1x_3,\ -5x_2^2$, are monomials of degree 2.
4. $x_1x_2x_3,\ x_1^2x_3,\ -x_2^3$, are monomials of degree 3.
5. $x_1^4,\ -x_1^2x_3^2,\ x_1x_2^2x_3$, are monomials of degree 4.

It is convenient to introduce multi-indices, where an *n-dimensional multi-index* is an n-tuple $\alpha = (k_1, \ldots, k_n)$ with $n \geq 1$ and $k_i \in \mathbb{N}$. Let $|\alpha| = k_1 + \cdots + k_n$. Then we can write

$$x^\alpha = x_1^{k_1} \cdots x_n^{k_n}.$$

For example, for $n = 3$,

$$x^{(1,2,1)} = x_1x_2^2x_3,\ x^{(0,2,2)} = x_2^2x_3^2.$$

Definition 11.1. A *polynomial* $P(x_1, \ldots, x_n)$ in the variables $x_1, \ldots, x_n$ with integer coefficients is a finite sum of monomials of the form

$$P(x_1, \ldots, x_n) = \sum_{\alpha} a_\alpha x^\alpha,$$

where the α's are n-dimensional multi-indices, and with $a_\alpha \in \mathbb{Z}$. The maximum of the degrees $|\alpha|$ of the monomials $a_\alpha x^\alpha$ is called the *total degree* of the polynomial $P(x_1, \ldots, x_n)$. The set of all such polynomials is denoted by $\mathbb{Z}[x_1, \ldots, x_n]$.

Sometimes, we write P instead of $P(x_1, \ldots, x_n)$. We also use variables x, y, z *etc.* instead of $x_1, x_2, x_3, \ldots$.

For example, $2x - 3y - 1$ is a polynomial of total degree 1, $x^2 + y^2 - z^2$ is a polynomial of total degree 2, and $x^3 + y^3 + z^3 - 29$ is a polynomial of total degree 3.

Mathematicians have been interested for a long time in the problem of solving equations of the form

$$P(x_1, \ldots, x_n) = 0,$$

with $P \in \mathbb{Z}[x_1, \ldots, x_n]$, seeking only *integer solutions* for $x_1, \ldots, x_n$.

Diophantus of Alexandria, a Greek mathematician of the 3rd century, was one of the first to investigate such equations. For this reason, seeking integer solutions of polynomials in $\mathbb{Z}[x_1, \ldots, x_n]$ is referred to as *solving Diophantine equations*.

This problem is not as simple as it looks. The equation

$$2x - 3y - 1 = 0$$

obviously has the solution $x = 2, y = 1$, and more generally $x = -1 + 3a$, $y = -1 + 2a$, for any integer $a \in \mathbb{Z}$.

The equation

$$x^2 + y^2 - z^2 = 0$$

has the solution $x = 3$, $y = 4$, $z = 5$, since $3^2 + 4^2 = 9 + 16 = 25 = 5^2$. More generally, the reader should check that

$$x = t^2 - 1,\ y = 2t,\ z = t^2 + 1$$

is a solution for all $t \in \mathbb{Z}$.

The equation

$$x^3 + y^3 + z^3 - 29 = 0$$

has the solution $x = 3$, $y = 1$, $z = 1$.

What about the equation

$$x^3 + y^3 + z^3 - 30 = 0?$$

Amazingly, the only known integer solution is

$$(x, y, z) = (283059965, 2218888517, 2220422932),$$

discovered in 1999 by E. Pine, K. Yarbrough, W. Tarrant, and M. Beck, following an approach suggested by N. Elkies.

And what about solutions of the equation

$$x^3 + y^3 + z^3 - 33 = 0?$$

Well, nobody knows whether this equation is solvable in integers!

In 1900, at the International Congress of Mathematicians held in Paris, the famous mathematician David Hilbert presented a list of ten open mathematical problems. Soon after, Hilbert published a list of 23 problems. The tenth problem is this:

Hilbert's tenth problem (H10)

Find an algorithm that solves the following problem:

Given as input a polynomial $P \in \mathbb{Z}[x_1, \ldots, x_n]$ with integer coefficients, return YES or NO, according to whether there exist integers $a_1, \ldots, a_n \in \mathbb{Z}$ so that $P(a_1, \ldots, a_n) = 0$; that is, the Diophantine equation $P(x_1, \ldots, x_n) = 0$ has a solution.

It is important to note that at the time Hilbert proposed his tenth problem, a rigorous mathematical definition of the notion of algorithm did not exist. In fact, the machinery needed to even define the notion of algorithm did not exist. It is only around 1930 that precise definitions of the notion of computability due to Turing, Church, and Kleene, were formulated, and soon after shown to be all equivalent.

So to be precise, the above statement of Hilbert's tenth should say: find a RAM program (or equivalently a Turing machine) that solves the following problem: ...

In 1970, the following somewhat surprising resolution of Hilbert's tenth problem was reached:

Theorem (Davis-Putnam-Robinson-Matiyasevich)

Hilbert's thenth problem is undecidable; that is, there is no algorithm for solving Hilbert's tenth problem.

In 1962, Davis, Putnam and Robinson had shown that if a fact known as *Julia Robinson hypothesis* could be proved, then Hilbert's tenth problem would be undecidable. At the time, the Julia Robinson hypothesis seemed implausible to many, so it was a surprise when in 1970 Matiyasevich found a set satisfying the Julia Robinson hypothesis, thus completing the proof of the undecidability of Hilbert's tenth problem. It is also a bit startling that Matiyasevich' set involves the Fibonacci numbers.

A detailed account of the history of the proof of the undecidability of Hilbert's tenth problem can be found in Martin Davis' classical paper Davis [6].

Even though Hilbert's tenth problem turned out to have a negative solution, the knowledge gained in developing the methods to prove this result is very significant. What was revealed is that polynomials have considerable expressive powers. This is what we discuss in the next section.

11.2 Diophantine Sets and Listable Sets

We begin by showing that if we can prove that the version of Hilbert's tenth problem *with solutions restricted to belong to* $\mathbb{N}$ is undecidable, then Hilbert's tenth problem (with solutions in $\mathbb{Z}$ is undecidable).

Proposition 11.1. *If we had an algorithm for solving Hilbert's tenth problem (with solutions in* $\mathbb{Z}$*), then we would have an algorithm for solving Hilbert's tenth problem with solutions restricted to belong to* $\mathbb{N}$ *(that is, nonnegative integers).*

Proof. The above statement is not at all obvious, although its proof is short with the help of some number theory. Indeed, by a theorem of Lagrange (Lagrange's four square theorem), *every* natural number m can be represented as the sum of four squares,

$$m = a_0^2 + a_1^2 + a_2^2 + a_3^2, \quad a_0, a_1, a_2, a_3 \in \mathbb{Z}.$$

We reduce Hilbert's tenth problem restricted to solutions in $\mathbb{N}$ to Hilbert's tenth problem (with solutions in $\mathbb{Z}$). Given a Diophantine equation $P(x_1, \ldots, x_n) = 0$, we can form the polynomial

$$Q = P(u_1^2 + v_1^2 + y_1^2 + z_1^2, \ldots, u_n^2 + v_n^2 + y_n^2 + z_n^2)$$

in the $4n$ variables u_i, v_i, y_i, z_i ($1 \leq i \leq n$) obtained by replacing x_i by $u_i^2 + v_i^2 + y_i^2 + z_i^2$ for $i = 1, \ldots, n$. If $Q = 0$ has a solution $(p_1, q_1, r_1, s_1, \ldots, p_n, q_n, r_n, s_n,)$ with $p_i, q_i, r_i, s_i \in \mathbb{Z}$, then if we set $a_i = p_i^2 + q_i^2 + r_i^2 + s_i^2$, obviously $P(a_1, \ldots, a_n) = 0$ with $a_i \in \mathbb{N}$. Conversely, if $P(a_1, \ldots, a_n) = 0$ with $a_i \in \mathbb{N}$, then by Lagrange's theorem there exist some $p_i, q_i, r_i, s_i \in \mathbb{Z}$ (in fact $\mathbb{N}$) such that $a_i = p_i^2 + q_i^2 + r_i^2 + s_i^2$ for $i = 1, \ldots, n$, and the equation $Q = 0$ has the solution $(p_1, q_1, r_1, s_1, \ldots, p_n, q_n, r_n, s_n,)$ with $p_i, q_i, r_i, s_i \in \mathbb{Z}$. Therefore $Q = 0$ has a solution $(p_1, q_1, r_1, s_1, \ldots, p_n, q_n, r_n, s_n,)$ with $p_i, q_i, r_i, s_i \in \mathbb{Z}$ iff $P = 0$ has a solution $(a_!, \ldots, a_n)$ with $a_i \in \mathbb{N}$. If we had an algorithm to decide whether Q has a solution with its components in $\mathbb{Z}$, then we would have an algorithm to decide whether $P = 0$ has a solution with its components in $\mathbb{N}$. ☐

As consequence, the contrapositive of Proposition 11.1 shows that if the version of Hilbert's tenth problem restricted to solutions in $\mathbb{N}$ is undecidable, so is Hilbert's original problem (with solutions in $\mathbb{Z}$).

In fact, the Davis-Putnam-Robinson-Matiyasevich theorem establishes the undecidability of the version of Hilbert's tenth problem restricted to solutions in $\mathbb{N}$. *From now on, we restrict our attention to this version of Hilbert's tenth problem.*

A key idea is to use Diophantine equations with parameters, to *define* sets of numbers.

For example, consider the polynomial

$$P_1(a, y, z) = (y + 2)(z + 2) - a.$$

For $a \in \mathbb{N}$ fixed, the equation $(y + 2)(z + 2) - a = 0$, equivalently

$$a = (y + 2)(z + 2),$$

has a solution with $y, z \in \mathbb{N}$ iff a is composite.

If we now consider the polynomial

$$P_2(a, y, z) = y(2z + 3) - a,$$

for $a \in \mathbb{N}$ fixed, the equation $y(2z + 3) - a = 0$, equivalently

$$a = y(2z + 3),$$

has a solution with $y, z \in \mathbb{N}$ iff a is not a power of 2.

For a slightly more complicated example, consider the polynomial

$$P_3(a, y) = 3y + 1 - a^2.$$

We leave it as an exercise to show that the natural numbers a that satisfy the equation $3y + 1 - a^2 = 0$, equivalently

$$a^2 = 3y + 1,$$

or $(a - 1)(a + 1) = 3y$, are of the form $a = 3k + 1$ or $a = 3k + 2$, for any $k \in \mathbb{N}$.

In the first case, if we let S_1 be the set of composite natural numbers, then we can write

$$S_1 = \{a \in \mathbb{N} \mid (\exists y, z)((y + 2)(z + 2) - a = 0)\},$$

where it is understood that the existentially quantified variables y, z take their values in $\mathbb{N}$.

In the second case, if we let S_2 be the set of natural numbers that are not powers of 2, then we can write

$$S_2 = \{a \in \mathbb{N} \mid (\exists y, z)(y(2z + 3) - a = 0)\}.$$

In the third case, if we let S_3 be the set of natural numbers that are congruent to 1 or 2 modulo 3, then we can write

$$S_3 = \{a \in \mathbb{N} \mid (\exists y)(3y + 1 - a^2 = 0)\}.$$

A more explicit Diophantine definition for S_3 is

$$S_3 = \{a \in \mathbb{N} \mid (\exists y)((a - 3y - 1)(a - 3y - 2) = 0)\}.$$

The natural generalization is as follows.

Definition 11.2. A set $S \subseteq \mathbb{N}$ of natural numbers is *Diophantine* (or *Diophantine definable*) if there is a polynomial $P(a, x_1, \ldots, x_n) \in \mathbb{Z}[a, x_1, \ldots, x_n]$, with $n \geq 0$[1] such that

$$S = \{a \in \mathbb{N} \mid (\exists x_1, \ldots, x_n)(P(a, x_1, \ldots, x_n) = 0)\},$$

where it is understood that the existentially quantified variables $x_1, \ldots, x_n$ take their values in $\mathbb{N}$. More generally, a relation $R \subseteq \mathbb{N}^m$ is *Diophantine* ($m \geq 2$) if there is a polynomial $P(a_1, \ldots, a_m, x_1, \ldots, x_n) \in \mathbb{Z}[a_1, \ldots, a_m, x_1, \ldots, x_n]$, with $n \geq 0$, such that

$$R = \{(a_1, \ldots, a_m) \in \mathbb{N}^m \mid (\exists x_1, \ldots, x_n)(P(a_1, \ldots, a_m, x_1, \ldots, x_n) = 0)\},$$

where it is understood that the existentially quantified variables $x_1, \ldots, x_n$ take their values in $\mathbb{N}$.

For example, the strict order relation $a_1 < a_2$ is defined as follows:

$$a_1 < a_2 \quad \text{iff} \quad (\exists x)(a_1 + 1 + x - a_2 = 0),$$

and the divisibility relation $a_1 \mid a_2$ (a_1 divides a_2) is defined as follows:

$$a_1 \mid a_2 \quad \text{iff} \quad (\exists x)(a_1 x - a_2 = 0).$$

What about the ternary relation $R \subseteq \mathbb{N}^3$ given by

$$(a_1, a_2, a_3) \in R \quad \text{if} \quad a_1 \mid a_2 \quad \text{and} \quad a_1 < a_3?$$

At first glance it is not obvious how to "convert" a conjunction of Diophantine definitions into a single Diophantine definition, but we can do this using the following trick: given any finite number of Diophantine equations in the variables $x_1, \ldots, x_n$,

$$P_1 = 0,\ P_2 = 0,\ \ldots,\ P_m = 0, \tag{$*$}$$

observe that $(*)$ has a solution $(a_1, \ldots, a_n)$, which means that $P_i(a_1, \ldots, a_n) = 0$ for $i = 1, \ldots, m$, iff the single equation

$$P_1^2 + P_2^2 + \cdots + P_m^2 = 0 \tag{$**$}$$

also has the solution $(a_1, \ldots, a_n)$. This is because, since $P_1^2, P_2^2, \ldots, P_m^2$ are all nonnegative, their sum is equal to zero iff they are *all equal to zero*, that is $P_i^2 = 0$ for $i = 1 \ldots, m$, which is equivalent to $P_i = 0$ for $i = 1 \ldots, m$.

Using this trick, we see that

$$(a_1, a_2, a_3) \in R \quad \text{iff} \quad (\exists u, v)((a_1 u - a_2)^2 + (a_1 + 1 + v - a_3)^2 = 0).$$

We can also define the notion of Diophantine function.

[1]We have to allow $n = 0$. Otherwise singleton sets would not be Diophantine.

Definition 11.3. A function $f\colon \mathbb{N}^n \to \mathbb{N}$ is *Diophantine* iff its graph $\{(a_0, a_1, \ldots, a_n) \subseteq \mathbb{N}^{n+1} \mid a_0 = f(a_1, \ldots, a_n)\}$ is Diophantine.

For example, the pairing function J and the projection functions K, L due to Cantor introduced in Section 9.1 are Diophantine, since

$$\begin{aligned} z = J(x,y) \quad &\text{iff} \quad (x+y-1)(x+y) + 2x - 2z = 0 \\ x = K(z) \quad &\text{iff} \quad (\exists y)((x+y-1)(x+y) + 2x - 2z = 0) \\ y = L(z) \quad &\text{iff} \quad (\exists x)((x+y-1)(x+y) + 2x - 2z = 0). \end{aligned}$$

How extensive is the family of Diophantine sets? The remarkable fact proved by Davis-Putnam-Robinson-Matiyasevich is that they coincide with the listable sets (the recursively enumerable sets). This is a highly nontrivial result.

The easy direction is the following result.

Proposition 11.2. *Every Diophantine set is listable (recursively enumerable).*

Proof sketch. Suppose S is given as

$$S = \{a \in \mathbb{N} \mid (\exists x_1, \ldots, x_n)(P(a, x_1, \ldots, x_n) = 0)\},$$

Using the extended pairing function $\langle x_1, \ldots, x_n \rangle_n$ of Section 9.1, we enumerate all n-tuples $(x_1, \ldots, x_n) \in \mathbb{N}^n$, and during this process we compute $P(a, x_1, \ldots, x_n)$. If $P(a, x_1, \ldots, x_n)$ is zero, then we output a, else we go on. This way, S is the range of a computable function, and it is listable. $\square$

It is also easy to see that every Diophantine function is partial computable. The main theorem of the theory of Diophantine sets is the following deep result.

Theorem 11.3. *(Davis-Putnam-Robinson-Matiyasevich, 1970) Every listable subset of* $\mathbb{N}$ *is Diophantine. Every partial computable function is Diophantine.*

Theorem 11.3 is often referred to as the *DPRM theorem.* A complete proof of Theorem 11.3 is provided in Davis [6]. As noted by Davis, although the proof is certainly long and nontrivial, it only uses elementary facts of number theory, nothing more sophisticated than the Chinese remainder theorem. Nevetherless, the proof is a tour de force.

One of the most difficult steps is to show that the exponential function $h(n, k) = n^k$ is Diophantine. This is done using the Pell equation. According to Martin Davis, the proof given in Davis [6] uses a combination of ideas from Matiyasevich and Julia Robinson. Matiyasevich's proof used the Fibonacci numbers.

Using some results from the theory of computation it is now easy to deduce that Hilbert's tenth problem is undecidable. To achieve this, recall that there are listable sets that are not

computable. For example, it is shown in Lemma 10.11 that $K = \{x \in \mathbb{N} \mid \varphi_x(x) \text{ is defined}\}$ is listable but not computable. Since K is listable, by Theorem 11.3, it is defined by some Diophantine equation

$$P(a, x_1, \ldots, x_n) = 0,$$

which means that

$$K = \{a \in \mathbb{N} \mid (\exists x_1 \ldots, x_n)(P(a, x_1, \ldots, x_n) = 0)\}.$$

We have the following strong form of the undecidability of Hilbert's tenth problem, in the sense that it shows that Hilbert's tenth problem is already undecidable for a fixed Diophantine equation in one parameter.

Theorem 11.4. *There is no algorithm which takes as input the polynomial $P(a, x_1, \ldots, x_n)$ defining K and any natural number $a \in \mathbb{N}$ and decides whether*

$$P(a, x_1, \ldots, x_n) = 0.$$

Consequently, Hilbert's tenth problem is undecidable.

Proof. If there was such an algorithm, then K would be decidable, a contradiction.

Any algorithm for solving Hilbert's tenth problem could be used to decide whether or not $P(a, x_1, \ldots, x_n) = 0$, but we just showed that there is no such algorithm. □

It is an open problem whether Hilbert's tenth problem is undecidable if we allow *rational solutions* (that is, $x_1, \ldots, x_n \in \mathbb{Q}$).

Alexandra Shlapentokh proved that various extensions of Hilbert's tenth problem are undecidable. These results deal with some algebraic number theory beyond the scope of these notes. Incidentally, Alexandra was an undegraduate at Penn and she worked on a logic project for me (finding a Gentzen system for a subset of temporal logic).

Having now settled once and for all the undecidability of Hilbert's tenth problem, we now briefly explore some interesting consequences of Theorem 11.3.

11.3 Some Applications of the DPRM Theorem

The first application of the DRPM theorem is a particularly striking way of defining the listable subsets of $\mathbb{N}$ as the nonnegative ranges of polynomials with integer coefficients. This result is due to Hilary Putnam.

Theorem 11.5. *For every listable subset S of $\mathbb{N}$, there is some polynomial $Q(x, x_1, \ldots, x_n)$ with integer coefficients such that*

$$S = \{Q(a, b_1, \ldots, b_n) \mid Q(a, b_1, \ldots, b_n) \in \mathbb{N},\ a, b_1, \ldots, b_n \in \mathbb{N}\}.$$

Proof. By the DPRM theorem (Theorem 11.3), there is some polynomial $P(x, x_1, \ldots, x_n)$ with integer coefficients such that

$$S = \{a \in \mathbb{N} \mid (\exists x_1, \ldots, x_n)(P(a, x_1, \ldots, x_n) = 0)\}.$$

Let $Q(x, x_1, \ldots, x_n)$ be given by

$$Q(x, x_1, \ldots, x_n) = (x + 1)(1 - P^2(x, x_1, \ldots, x_n)) - 1.$$

We claim that Q satisfies the statement of the theorem. If $a \in S$, then $P(a, b_1, \ldots, b_n) = 0$ for some $b_1, \ldots, b_n \in \mathbb{N}$, so

$$Q(a, b_1, \ldots, b_n) = (a + 1)(1 - 0) - 1 = a.$$

This shows that all $a \in S$ show up the the nonnegative range of Q. Conversely, assume that $Q(a, b_1, \ldots, b_n) \geq 0$ for some $a, b_1, \ldots, b_n \in \mathbb{N}$. Then by definition of Q we must have

$$(a + 1)(1 - P^2(a, b_1, \ldots, b_n)) - 1 \geq 0,$$

that is,

$$(a + 1)(1 - P^2(a, b_1, \ldots, b_n)) \geq 1,$$

and since $a \in \mathbb{N}$, this implies that $P^2(a, b_1, \ldots, b_n) < 1$, but since P is a polynomial with integer coefficients and $a, b_1, \ldots, b_n \in \mathbb{N}$, the expression $P^2(a, b_1, \ldots, b_n)$ must be a nonnegative integer, so we must have

$$P(a, b_1, \ldots, b_n) = 0,$$

which shows that $a \in S$. ☐

Remark: It should be noted that in general, the polynomials Q arising in Theorem 11.5 may take on negative integer values, and to obtain all listable sets, we must restrict ourself to their nonnegative range.

As an example, the set S_3 of natural numbers that are congruent to 1 or 2 modulo 3 is given by

$$S_3 = \{a \in \mathbb{N} \mid (\exists y)(3y + 1 - a^2 = 0)\}.$$

so by Theorem 11.5, S_3 is the nonnegative range of the polynomial

$$\begin{aligned} Q(x, y) &= (x + 1)(1 - (3y + 1 - x^2)^2)) - 1 \\ &= -(x + 1)((3y - x^2)^2 + 2(3y - x^2))) - 1 \\ &= (x + 1)(x^2 - 3y)(2 - (x^2 - 3y)) - 1. \end{aligned}$$

Observe that $Q(x, y)$ takes on negative values. For example, $Q(0, 0) = -1$. Also, in order for $Q(x, y)$ to be nonnegative, $(x^2 - 3y)(2 - (x^2 - 3y))$ must be positive, but this can only happen if $x^2 - 3y = 1$, that is, $x^2 = 3y + 1$, which is the original equation defining S_3.

There is no miracle. The nonnegativity of $Q(x, x_1, \ldots, x_n)$ must subsume the solvability of the equation $P(x, x_1, \ldots, x_n) = 0$.

A particularly interesting listable set is the set of primes. By Theorem 11.5, in theory, the set of primes is the positive range of some polynomial with integer coefficients.

Remarkably, some explicit polynomials have been found. This is a nontrivial task. In particular, the process involves showing that the exponential function is definable, which was the stumbling block of the completion of the DPRM theorem for many years.

To give the reader an idea of how the proof begins, observe by the Bezout identity, if $p = s + 1$ and $q = s!$, then we can assert that p and q are relatively prime ($\gcd(p, q) = 1$) as the fact that the Diophantine equation

$$ap - bq = 1$$

is satisfied for some $a, b \in \mathbb{N}$. Then, it is not hard to see that $p \in \mathbb{N}$ is prime iff the following set of equations has a solution for $a, b, s, r, q \in \mathbb{N}$:

$$\begin{aligned} p &= s + 1 \\ p &= r + 2 \\ q &= s! \\ ap - bq &= 1. \end{aligned}$$

The problem with the above is that the equation $q = s!$ is not Diophantine. The next step is to show that the factorial function is Diophantine, and this involves a lot of work. One way to proceed is to show that the above system is equivalent to a system allowing the use of the exponential function. The final step is to show that the exponential function can be eliminated in favor of polynomial equations.

We refer the interested reader to the remarkable expository paper by Davis, Matiyasevich and Robinson [7] for details. Here is a polynomial of total degree 25 in 26 variables (due to J. Jones, D. Sato, H. Wada, D. Wiens) which produces the primes as its positive range:

$$\begin{aligned} &(k+2)\Big[1 - ([wz + h + j - q]^2 + [(gk + 2g + k + 1)(h + j) + h - z]^2 \\ &+ [16(k+1)^3(k+2)(n+1)^2 + 1 - f^2]^2 \\ &+ [2n + p + q + z - e]^2 + [e^3(e+2)(a+1)^2 + 1 - o^2]^2 \\ &+ [(a^2-1)y^2 + 1 - x^2]^2 + [16r^2y^4(a^2-1) + 1 - u^2]^2 \\ &+ [((a + u^2(u^2 - a))^2 - 1)(n + 4dy)^2 + 1 - (x + cu)^2]^2 \\ &+ [(a^2-1)l^2 + 1 - m^2]^2 + [ai + k + 1 - l - i]^2 + [n + l + v - y]^2 \\ &+ [p + l(a - n - 1) + b(2an + 2a - n^2 - 2n - 2) - m]^2 \\ &+ [q + y(a - p - 1) + s(2ap + 2a - p^2 - 2p - 2) - x]^2 \\ &+ [z + pl(a - p) + t(2ap - p^2 - 1) - pm]^2)\Big]. \end{aligned}$$

Around 2004, Nachi Gupta, an undergraduate student at Penn, and I, tried to produce the prime 2 as one of the values of the positive range of the above polynomial. It turns out that this leads to values of the variables that are so large that we never succeeded!

Other interesting applications of the DPRM theorem are the re-statements of famous open problems, such as the Riemann hypothesis, as the unsolvability of certain Diophantine equations. One may also obtain a nice variant of Gödel's incompleteness theorem. For all this, see Davis, Matiyasevich and Robinson [7].

Chapter 12

The Post Correspondence Problem; Applications to Undecidability Results

12.1 The Post Correspondence Problem

The Post correspondence problem (due to Emil Post) is another undecidable problem that turns out to be a very helpful tool for proving problems in logic or in formal language theory to be undecidable.

Let Σ be an alphabet with at least two letters. An instance of the *Post Correspondence problem* (for short, PCP) is given by two sequences $U = (u_1, \ldots, u_m)$ and $V = (v_1, \ldots, v_m)$, of strings $u_i, v_i \in \Sigma^*$.

The problem is to find whether there is a (finite) sequence $(i_1, \ldots, i_p)$, with $i_j \in \{1, \ldots, m\}$ for $j = 1, \ldots, p$, so that

$$u_{i_1} u_{i_2} \cdots u_{i_p} = v_{i_1} v_{i_2} \cdots v_{i_p}.$$

Equivalently, an instance of the PCP is a sequence of pairs

$$(u_1, v_1), \ldots, (u_m, v_m).$$

For example, consider the following problem:

$$(abab, ababaaa), (aaabbb, bb), (aab, baab), (ba, baa), (ab, ba), (aa, a).$$

There is a solution for the string 1234556:

$$abab\,aaabbb\,aab\,ba\,ab\,ab\,aa = ababaaa\,bb\,baab\,baa\,ba\,ba\,a.$$

We are beginning to suspect that this is a hard problem. Indeed, it is undecidable!

Theorem 12.1. *(Emil Post, 1946) The Post correspondence problem is undecidable, provided that the alphabet Σ has at least two symbols.*

There are several ways of proving Theorem 12.1, but the strategy is more or less the same: Reduce the halting problem to the PCP, by encoding sequences of ID's as partial solutions of the PCP.

For instance, this can be done for RAM programs. The first step is to show that every RAM program can be simulated by a single register RAM program.

Then, the halting problem for RAM programs with one register is reduced to the PCP (using the fact that only four kinds of instructions are needed). A proof along these lines was given by Dana Scott.

12.2 Some Undecidability Results for CFG's

Theorem 12.2. *It is undecidable whether a context-free grammar is ambiguous.*

Proof. We reduce the PCP to the ambiguity problem for CFG's. Given any instance $U = (u_1, \ldots, u_m)$ and $V = (v_1, \ldots, v_m)$ of the PCP, let $c_1, \ldots, c_m$ be m new symbols, and consider the following languages:

$$\begin{aligned} L_U &= \{u_{i_1} \cdots u_{i_p} c_{i_p} \cdots c_{i_1} \mid 1 \leq i_j \leq m, \\ &\qquad 1 \leq j \leq p,\, p \geq 1\}, \\ L_V &= \{v_{i_1} \cdots v_{i_p} c_{i_p} \cdots c_{i_1} \mid 1 \leq i_j \leq m, \\ &\qquad 1 \leq j \leq p,\, p \geq 1\}, \end{aligned}$$

and $L_{U,V} = L_U \cup L_V$.

We can easily construct a CFG, $G_{U,V}$, generating $L_{U,V}$. The productions are:

$$\begin{aligned} S &\longrightarrow S_U \\ S &\longrightarrow S_V \\ S_U &\longrightarrow u_i S_U c_i \\ S_U &\longrightarrow u_i c_i \\ S_V &\longrightarrow v_i S_V c_i \\ S_V &\longrightarrow v_i c_i. \end{aligned}$$

It is easily seen that the PCP for (U, V) has a solution iff $L_U \cap L_V \neq \emptyset$ iff G is ambiguous. $\square$

Remark: As a corollary, we also obtain the following result: It is undecidable for arbitrary context-free grammars G_1 and G_2 whether $L(G_1) \cap L(G_2) = \emptyset$ (see also Theorem 12.4).

Recall that the computations of a Turing Machine, M, can be described in terms of instantaneous descriptions, $upav$.

We can encode computations

$$ID_0 \vdash ID_1 \vdash \cdots \vdash ID_n$$

halting in a proper ID, as the language, L_M, consisting all of strings

$$w_0\#w_1^R\#w_2\#w_3^R\#\cdots\#w_{2k}\#w_{2k+1}^R,$$

or

$$w_0\#w_1^R\#w_2\#w_3^R\#\cdots\#w_{2k-2}\#w_{2k-1}^R\#w_{2k},$$

where $k \geq 0$, w_0 is a starting ID, $w_i \vdash w_{i+1}$ for all i with $0 \leq i < 2k+1$ and w_{2k+1} is proper halting ID in the first case, $0 \leq i < 2k$ and w_{2k} is proper halting ID in the second case.

The language L_M turns out to be the intersection of two context-free languages L_M^0 and L_M^1 defined as follows:

(1) The strings in L_M^0 are of the form

$$w_0\#w_1^R\#w_2\#w_3^R\#\cdots\#w_{2k}\#w_{2k+1}^R$$

or

$$w_0\#w_1^R\#w_2\#w_3^R\#\cdots\#w_{2k-2}\#w_{2k-1}^R\#w_{2k},$$

where $w_{2i} \vdash w_{2i+1}$ for all $i \geq 0$, and w_{2k} is a proper halting ID in the second case.

(2) The strings in L_M^1 are of the form

$$w_0\#w_1^R\#w_2\#w_3^R\#\cdots\#w_{2k}\#w_{2k+1}^R$$

or

$$w_0\#w_1^R\#w_2\#w_3^R\#\cdots\#w_{2k-2}\#w_{2k-1}^R\#w_{2k},$$

where $w_{2i+1} \vdash w_{2i+2}$ for all $i \geq 0$, w_0 is a starting ID, and w_{2k+1} is a proper halting ID in the first case.

Theorem 12.3. *Given any Turing machine M, the languages L_M^0 and L_M^1 are context-free, and $L_M = L_M^0 \cap L_M^1$.*

Proof. We can construct PDA's accepting L_M^0 and L_M^1. It is easily checked that $L_M = L_M^0 \cap L_M^1$. ☐

As a corollary, we obtain the following undecidability result:

Theorem 12.4. *It is undecidable for arbitrary context-free grammars* G_1 *and* G_2 *whether* $L(G_1) \cap L(G_2) = \emptyset$.

Proof. We can reduce the problem of deciding whether a partial recursive function is undefined everywhere to the above problem. By Rice's theorem, the first problem is undecidable.

However, this problem is equivalent to deciding whether a Turing machine never halts in a proper ID. By Theorem 12.3, the languages L_M^0 and L_M^1 are context-free. Thus, we can construct context-free grammars G_1 and G_2 so that $L_M^0 = L(G_1)$ and $L_M^1 = L(G_2)$. Then, M never halts in a proper ID iff $L_M = \emptyset$ iff (by Theorem 12.3), $L_M = L(G_1) \cap L(G_2) = \emptyset$. □

Given a Turing machine M, the language L_M is defined over the alphabet $\Delta = \Gamma \cup Q \cup \{\#\}$. The following fact is also useful to prove undecidability:

Theorem 12.5. *Given any Turing machine* M*, the language* $\Delta^* - L_M$ *is context-free.*

Proof. One can easily check that the conditions for not belonging to L_M can be checked by a PDA. □

As a corollary, we obtain:

Theorem 12.6. *Given any context-free grammar,* $G = (V, \Sigma, P, S)$*, it is undecidable whether* $L(G) = \Sigma^*$.

Proof. We can reduce the problem of deciding whether a Turing machine never halts in a proper ID to the above problem.

Indeed, given M, by Theorem 12.5, the language $\Delta^* - L_M$ is context-free. Thus, there is a CFG, G, so that $L(G) = \Delta^* - L_M$. However, M never halts in a proper ID iff $L_M = \emptyset$ iff $L(G) = \Delta^*$. □

As a consequence, we also obtain the following:

Theorem 12.7. *Given any two context-free grammar,* G_1 *and* G_2*, and any regular language,* R*, the following facts hold:*

(1) $L(G_1) = L(G_2)$ *is undecidable.*

(2) $L(G_1) \subseteq L(G_2)$ *is undecidable.*

(3) $L(G_1) = R$ *is undecidable.*

(4) $R \subseteq L(G_2)$ *is undecidable.*

In contrast to (4), the property $L(G_1) \subseteq R$ is decidable!

12.3 More Undecidable Properties of Languages; Greibach's Theorem

We conclude with a nice theorem of S. Greibach, which is a sort of version of Rice's theorem for families of languages.

Let $\mathcal{L}$ be a countable family of languages. We assume that there is a coding function $c\colon \mathcal{L} \to \mathbb{N}$ and that this function can be extended to code the regular languages (all alphabets are subsets of some given countably infinite set).

We also assume that $\mathcal{L}$ is effectively closed under union, and concatenation with the regular languages.

This means that given any two languages L_1 and L_2 in $\mathcal{L}$, we have $L_1 \cup L_2 \in \mathcal{L}$, and $c(L_1 \cup L_2)$ is given by a recursive function of $c(L_1)$ and $c(L_2)$, and that for every regular language R, we have $L_1R \in \mathcal{L}$, $RL_1 \in \mathcal{L}$, and $c(RL_1)$ and $c(L_1R)$ are recursive functions of $c(R)$ and $c(L_1)$.

Given any language, $L \subseteq \Sigma^*$, and any string, $w \in \Sigma^*$, we define L/w by

$$L/w = \{u \in \Sigma^* \mid uw \in L\}.$$

Theorem 12.8. *(Greibach) Let $\mathcal{L}$ be a countable family of languages that is effectively closed under union, and concatenation with the regular languages, and assume that the problem $L = \Sigma^*$ is undecidable for $L \in \mathcal{L}$ and any given sufficiently large alphabet Σ. Let P be any nontrivial property of languages that is true for the regular languages, and so that if $P(L)$ holds for any $L \in \mathcal{L}$, then $P(L/a)$ also holds for any letter a. Then, P is undecidable for $\mathcal{L}$.*

Proof. Since P is nontrivial for $\mathcal{L}$, there is some $L_0 \in \mathcal{L}$ so that $P(L_0)$ is false.

Let Σ be large enough, so that $L_0 \subseteq \Sigma^*$, and the problem $L = \Sigma^*$ is undecidable for $L \in \mathcal{L}$.

We show that given any $L \in \mathcal{L}$, with $L \subseteq \Sigma^*$, we can construct a language $L_1 \in \mathcal{L}$, so that $L = \Sigma^*$ iff $P(L_1)$ holds. Thus, the problem $L = \Sigma^*$ for $L \in \mathcal{L}$ reduces to property P for $\mathcal{L}$, and since for Σ big enough, the first problem is undecidable, so is the second.

For any $L \in \mathcal{L}$, with $L \subseteq \Sigma^*$, let

$$L_1 = L_0\#\Sigma^* \cup \Sigma^*\#L.$$

Since $\mathcal{L}$ is effectively closed under union and concatenation with the regular languages, we have $L_1 \in \mathcal{L}$.

If $L = \Sigma^*$, then $L_1 = \Sigma^*\#\Sigma^*$, a regular language, and thus, $P(L_1)$ holds, since P holds for the regular languages.

Conversely, we would like to prove that if $L \neq \Sigma^*$, then $P(L_1)$ is false.

Since $L \neq \Sigma^*$, there is some $w \notin L$. But then,

$$L_1/\#w = L_0.$$

Since P is preserved under quotient by a single letter, by a trivial induction, if $P(L_1)$ holds, then $P(L_0)$ also holds. However, $P(L_0)$ is false, so $P(L_1)$ must be false.

Thus, we proved that $L = \Sigma^*$ iff $P(L_1)$ holds, as claimed. $\square$

Greibach's theorem can be used to show that it is undecidable whether a context-free grammar generates a regular language.

It can also be used to show that it is undecidable whether a context-free language is inherently ambiguous.

Chapter 13

Computational Complexity; $\mathcal{P}$ and $\mathcal{NP}$

13.1 The Class $\mathcal{P}$

In the previous two chapters, we clarified what it means for a problem to be decidable or undecidable. This chapter is heavily inspired by Lewis and Papadimitriou's excellent treatment [11].

In principle, if a problem is decidable, then there is an algorithm (i.e., a procedure that halts for every input) that decides every instance of the problem.

However, from a practical point of view, knowing that a problem is decidable may be useless, if the number of steps (*time complexity*) required by the algorithm is excessive, for example, exponential in the size of the input, or worse.

For instance, consider the *traveling salesman problem*, which can be formulated as follows:

We have a set $\{c_1, \ldots, c_n\}$ of cities, and an $n \times n$ matrix $D = (d_{ij})$ of nonnegative integers, the *distance matrix*, where d_{ij} denotes the distance between c_i and c_j, which means that $d_{ii} = 0$ and $d_{ij} = d_{ji}$ for all $i \neq j$.

The problem is to find a *shortest tour* of the cities, that is, a permutation π of $\{1, \ldots, n\}$ so that the *cost*

$$C(\pi) = d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + \cdots + d_{\pi(n-1)\pi(n)} + d_{\pi(n)\pi(1)}$$

is as small as possible (minimal).

One way to solve the problem is to consider all possible tours, i.e., $n!$ permutations.

Actually, since the starting point is irrelevant, we need only consider $(n-1)!$ tours, but this still grows very fast. For example, when $n = 40$, it turns out that $39!$ exceeds 10^{45}, a huge number.

Consider the 4×4 symmetric matrix given by

$$D = \begin{pmatrix} 0 & 2 & 1 & 1 \\ 2 & 0 & 1 & 1 \\ 1 & 1 & 0 & 3 \\ 1 & 1 & 3 & 0 \end{pmatrix},$$

and the budget $B = 4$. The tour specified by the permutation

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 2 & 3 \end{pmatrix}$$

has cost 4, since

$$\begin{aligned} c(\pi) &= d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + d_{\pi(3)\pi(4)} + d_{\pi(4)\pi(1)} \\ &= d_{14} + d_{42} + d_{23} + d_{31} \\ &= 1 + 1 + 1 + 1 = 4. \end{aligned}$$

The cities in this tour are traversed in the order

$$(1, 4, 2, 3, 1).$$

Remark: The permutation π shown above is described in Cauchy's *two-line notation*,

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 2 & 3 \end{pmatrix},$$

where every element in the second row is the image of the element immediately above it in the first row: thus

$$\pi(1) = 1,\ \pi(2) = 4,\ \pi(3) = 2,\ \pi(4) = 3.$$

Thus, to capture the essence of practically feasible algorithms, we must limit our computational devices to run only for a number of steps that is bounded by a *polynomial* in the length of the input.

We are led to the definition of polynomially bounded computational models.

Definition 13.1. A deterministic Turing machine M is said to be *polynomially bounded* if there is a polynomial $p(X)$ so that the following holds: For every input $x \in \Sigma^*$, there is no ID ID_n so that

$$ID_0 \vdash ID_1 \vdash^* ID_{n-1} \vdash ID_n, \quad \text{with} \quad n > p(|x|),$$

where $ID_0 = q_0 x$ is the starting ID.

A language $L \subseteq \Sigma^*$ is *polynomially decidable* if there is a polynomially bounded Turing machine that accepts L. The family of all polynomially decidable languages is denoted by $\mathcal{P}$.

Remark: Even though Definition 13.1 is formulated for Turing machines, it can also be formulated for other models, such as RAM programs.

The reason is that the conversion of a Turing machine into a RAM program (and vice versa) produces a program (or a machine) whose size is polynomial in the original device.

The following proposition, although trivial, is useful:

Proposition 13.1. *The class $\mathcal{P}$ is closed under complementation.*

Of course, many languages do not belong to $\mathcal{P}$. One way to obtain such languages is to use a diagonal argument. But there are also many natural languages that are not in $\mathcal{P}$, although this may be very hard to prove for some of these languages.

Let us consider a few more problems in order to get a better feeling for the family $\mathcal{P}$.

13.2 Directed Graphs, Paths

Recall that a *directed graph*, G, is a pair $G = (V, E)$, where $E \subseteq V \times V$. Every $u \in V$ is called a *node* (or *vertex*) and a pair $(u, v) \in E$ is called an *edge* of G.

We will restrict ourselves to *simple graphs*, that is, graphs without edges of the form (u, u); equivalently, $G = (V, E)$ is a simple graph if whenever $(u, v) \in E$, then $u \neq v$.

Given any two nodes $u, v \in V$, a *path from u to v* is any sequence of $n + 1$ edges $(n \geq 0)$

$$(u, v_1), (v_1, v_2), \ldots, (v_n, v).$$

(If $n = 0$, a path from u to v is simply a single edge, (u, v).)

A graph G is *strongly connected* if for every pair $(u, v) \in V \times V$, there is a path from u to v. A *closed path, or cycle*, is a path from some node u to itself.

We will restrict out attention to finite graphs, i.e. graphs (V, E) where V is a finite set.

Definition 13.2. Given a graph G, an *Eulerian cycle* is a cycle in G that passes through all the nodes (possibly more than once) and every edge of G exactly once. A *Hamiltonian cycle* is a cycle that passes through all the nodes exactly once (note, some edges may not be traversed at all).

Eulerian Cycle Problem: Given a graph G, is there an Eulerian cycle in G?

Hamiltonian Cycle Problem: Given a graph G, is there an Hamiltonian cycle in G?

13.3 Eulerian Cycles

The following graph is a directed graph version of the Königsberg bridge problem, solved by Euler in 1736.

The nodes A, B, C, D correspond to four areas of land in Königsberg and the edges to the seven bridges joining these areas of land.

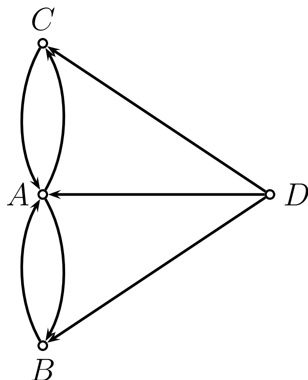


Figure 13.1: A directed graph modeling the Königsberg bridge problem

The problem is to find a closed path that crosses every bridge exactly once and returns to the starting point.

In fact, the problem is unsolvable, as shown by Euler, because some nodes do not have the same number of incoming and outgoing edges (in the undirected version of the problem, some nodes do not have an even degree.)

It may come as a surprise that the Eulerian Cycle Problem does have a polynomial time algorithm, but that so far, not such algorithm is known for the Hamiltonian Cycle Problem.

The reason why the Eulerian Cycle Problem is decidable in polynomial time is the following theorem due to Euler:

Theorem 13.2. *A graph $G = (V, E)$ has an Eulerian cycle iff the following properties hold:*

(1) The graph G is strongly connected.

(2) Every node has the same number of incoming and outgoing edges.

Proving that properties (1) and (2) hold if G has an Eulerian cycle is fairly easy. The converse is harder, but not that bad (try!).

Theorem 13.2 shows that it is necessary to check whether a graph is strongly connected. This can be done by computing the transitive closure of E, which can be done in polynomial time (in fact, $O(n^3)$).

Checking property (2) can clearly be done in polynomial time. Thus, the Eulerian cycle problem is in $\mathcal{P}$.

Unfortunately, no theorem analogous to Theorem 13.2 is know for Hamiltonian cycles.

13.4 Hamiltonian Cycles

A game invented by Sir William Hamilton in 1859 uses a regular solid dodecahedron whose twenty vertices are labeled with the names of famous cities.

The player is challenged to "travel around the world" by finding a closed cycle along the edges of the dodecahedron which passes through every city exactly once (this is the undirected version of the Hamiltonian cycle problem).

In graphical terms, assuming an orientation of the edges between cities, the graph D shown in Figure 13.2 is a plane projection of a regular dodecahedron and we want to know if there is a Hamiltonian cycle in this directed graph.

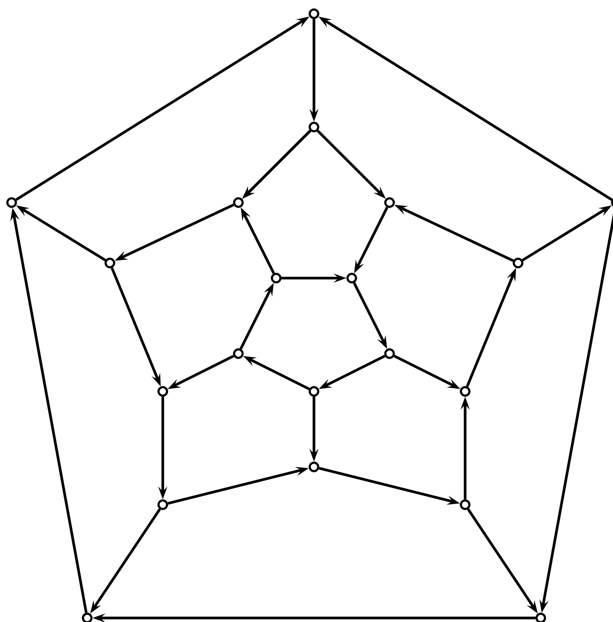


Figure 13.2: A tour "around the world."

Finding a Hamiltonian cycle in this graph does not appear to be so easy!

A solution is shown in Figure 13.3 below:

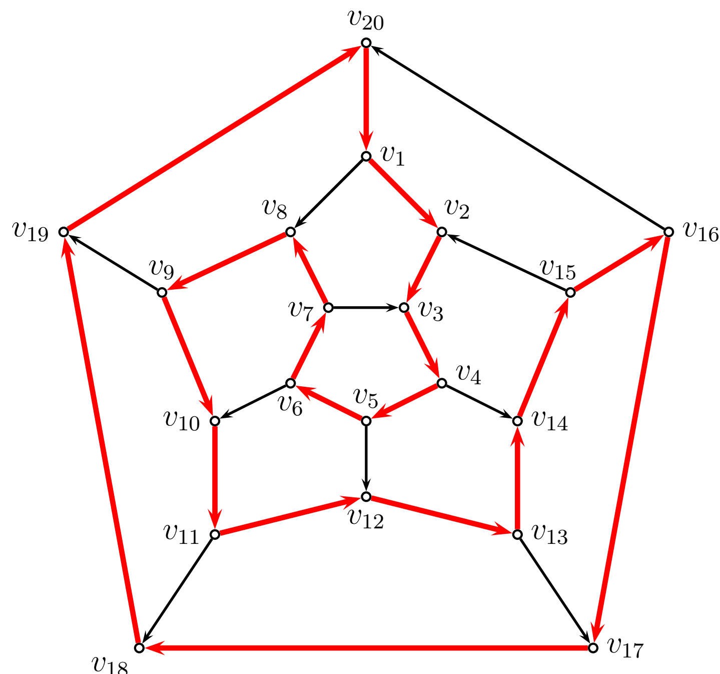


Figure 13.3: A Hamiltonian cycle in D.

A solution!

Remark: We talked about problems being decidable in polynomial time. Obviously, this is equivalent to deciding some property of a certain class of objects, for example, finite graphs.

Our framework requires that we first encode these classes of objects as strings (or numbers), since $\mathcal{P}$ consists of languages.

Thus, when we say that a property is decidable in polynomial time, we are really talking about the encoding of this property as a language. Thus, we have to be careful about these encodings, but it is rare that encodings cause problems.

13.5 Propositional Logic and Satisfiability

We define the syntax and the semantics of propositions in conjunctive normal form (CNF).

The syntax has to do with the legal form of propositions in CNF. Such propositions are interpreted as truth functions, by assigning truth values to their variables.

We begin by defining propositions in CNF. Such propositions are constructed from a countable set, **PV**, of *propositional (or boolean) variables*, say

$$\mathbf{PV} = \{x_1, x_2, \ldots, \},$$

using the connectives $\wedge$ (and), $\vee$ (or) and $\neg$ (negation).

We define a *literal (or atomic proposition)*, L, as $L = x$ or $L = \neg x$, also denoted by $\overline{x}$, where $x \in \mathbf{PV}$.

A *clause*, C, is a disjunction of pairwise distinct literals,

$$C = (L_1 \vee L_2 \vee \cdots \vee L_m).$$

Thus, a clause may also be viewed as a nonempty *set*

$$C = \{L_1, L_2, \ldots, L_m\}.$$

We also have a special clause, the *empty clause*, denoted $\perp$ or $\square$ (or $\{\}$). It corresponds to the truth value false.

A *proposition in CNF, or boolean formula*, P, is a conjunction of pairwise distinct clauses

$$P = C_1 \wedge C_2 \wedge \cdots \wedge C_n.$$

Thus, a boolean formula may also be viewed as a nonempty *set*

$$P = \{C_1, \ldots, C_n\},$$

but this time, the comma is interpreted as conjunction. We also allow the proposition $\perp$, and sometimes the proposition $\top$ (corresponding to the truth value true).

For example, here is a boolean formula:

$$P = \{(x_1 \vee x_2 \vee x_3), (\overline{x_1} \vee x_2), (\overline{x_2} \vee x_3), (\overline{x_3} \vee x_1), (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})\}.$$

In order to interpret boolean formulae, we use truth assignments.

We let $\mathrm{BOOL} = \{\mathbf{F}, \mathbf{T}\}$, the set of truth values, where $\mathbf{F}$ stands for false and $\mathbf{T}$ stands for true.

A *truth assignment (or valuation)*, v, is any function $v \colon \mathbf{PV} \to \mathrm{BOOL}$.

For example, the function $v_F \colon \mathbf{PV} \to \mathrm{BOOL}$ given by

$$v_F(x_i) = \mathbf{F} \quad \text{for all } i \geq 1$$

is a truth assignment, and so is the function $v_T \colon \mathbf{PV} \to \mathrm{BOOL}$ given by

$$v_T(x_i) = \mathbf{T} \quad \text{for all } i \geq 1.$$

The function $v\colon \mathbf{PV} \to \mathrm{BOOL}$ given by

$$\begin{aligned} v(x_1) &= \mathbf{T} \\ v(x_2) &= \mathbf{F} \\ v(x_3) &= \mathbf{T} \\ v(x_i) &= \mathbf{T} \quad \text{for all } i \geq 4 \end{aligned}$$

is also a truth assignment.

Given a truth assignment $v\colon \mathbf{PV} \to \mathrm{BOOL}$, we define the *truth value* $\widehat{v}(X)$ of a literal, clause, and boolean formula, X, using the following recursive definition:

(1) $\widehat{v}(\perp) = \mathbf{F}$, $\widehat{v}(\top) = \mathbf{T}$.

(2) $\widehat{v}(x) = v(x)$, if $x \in \mathbf{PV}$.

(3) $\widehat{v}(\overline{x}) = \overline{v(x)}$, if $x \in \mathbf{PV}$, where $\overline{v(x)} = \mathbf{F}$ if $v(x) = \mathbf{T}$ and $\overline{v(x)} = \mathbf{T}$ if $v(x) = \mathbf{F}$.

(4) $\widehat{v}(C) = \mathbf{F}$ if C is a clause and iff $\widehat{v}(L_i) = \mathbf{F}$ for all literals L_i in C, otherwise $\mathbf{T}$.

(5) $\widehat{v}(P) = \mathbf{T}$ if P is a boolean formula and iff $\widehat{v}(C_j) = \mathbf{T}$ for all clauses C_j in P, otherwise $\mathbf{F}$.

Since a boolean formula P only contains a finite number of variables, say $\{x_{i_1}, \ldots, x_{i_n}\}$, one should expect that its truth value $\widehat{v}(P)$ depends only on the truth values assigned by the truth assignment v to the variables in the set $\{x_{i_1}, \ldots, x_{i_n}\}$, and this is indeed the case. The following proposition is easily shown by induction on the depth of P (viewed as a tree).

Proposition 13.3. *Let P be a boolean formula containing the set of variables $\{x_{i_1}, \ldots, x_{i_n}\}$. If $v_1\colon \mathbf{PV} \to \mathrm{BOOL}$ and $v_2\colon \mathbf{PV} \to \mathrm{BOOL}$ are any truth assignments agreeing on the set of variables $\{x_{i_1}, \ldots, x_{i_n}\}$, which means that*

$$v_1(x_{i_j}) = v_2(x_{i_j}) \quad \textit{for } j = 1, \ldots, n,$$

then $\widehat{v_1}(P) = \widehat{v_2}(P)$.

In view of Proposition 13.3, given any boolean formula P, we only need to specify the values of a truth assignment v for the variables occurring on P. For example, given the boolean formula

$$P = \{(x_1 \vee x_2 \vee x_3), (\overline{x_1} \vee x_2), (\overline{x_2} \vee x_3), (\overline{x_3} \vee x_1), (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})\},$$

we only need to specify $v(x_1), v(x_2), v(x_3)$. Thus there are $2^3 = 8$ distinct truth assignments:

$$\begin{array}{ll} \mathbf{F}, \mathbf{F}, \mathbf{F} & \mathbf{T}, \mathbf{F}, \mathbf{F} \\ \mathbf{F}, \mathbf{F}, \mathbf{T} & \mathbf{T}, \mathbf{F}, \mathbf{T} \\ \mathbf{F}, \mathbf{T}, \mathbf{F} & \mathbf{T}, \mathbf{T}, \mathbf{F} \\ \mathbf{F}, \mathbf{T}, \mathbf{T} & \mathbf{T}, \mathbf{T}, \mathbf{T}. \end{array}$$

In general, there are 2^n distinct truth assignments to n distinct variables.

Here is an example showing the evaluation of the truth value $\widehat{v}(P)$ for the boolean formula

$$\begin{aligned} P &= (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee x_3) \wedge (\overline{x_3} \vee x_1) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) \\ &= \{(x_1 \vee x_2 \vee x_3), (\overline{x_1} \vee x_2), (\overline{x_2} \vee x_3), (\overline{x_3} \vee x_1), (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})\}, \end{aligned}$$

and the truth assignment

$$v(x_1) = \mathbf{T}, \quad v(x_2) = \mathbf{F}, \quad v(x_3) = \mathbf{F}.$$

For the literals, we have

$$\widehat{v}(x_1) = \mathbf{T}, \quad \widehat{v}(x_2) = \mathbf{F} \quad \widehat{v}(x_3) = \mathbf{F}, \quad \widehat{v}(\overline{x_1}) = \mathbf{F}, \quad \widehat{v}(\overline{x_2}) = \mathbf{T} \quad \widehat{v}(\overline{x_3}) = \mathbf{T},$$

for the clauses

$$\begin{aligned} \widehat{v}(x_1 \vee x_2 \vee x_3) &= \widehat{v}(x_1) \vee \widehat{v}(x_2) \vee \widehat{v}(x_3) = \mathbf{T} \vee \mathbf{F} \vee \mathbf{F} = \mathbf{T}, \\ \widehat{v}(\overline{x_1} \vee x_2) &= \widehat{v}(\overline{x_1}) \vee \widehat{v}(x_2) = \mathbf{F} \vee \mathbf{F} = \mathbf{F}, \\ \widehat{v}(\overline{x_2} \vee x_3) &= \widehat{v}(\overline{x_2}) \vee \widehat{v}(x_3) = \mathbf{T} \vee \mathbf{F} = \mathbf{T}, \\ \widehat{v}(\overline{x_3} \vee x_1) &= \widehat{v}(\overline{x_3}) \vee \widehat{v}(x_1) = \mathbf{T} \vee \mathbf{T} = \mathbf{T}, \\ \widehat{v}(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) &= \widehat{v}(\overline{x_1}) \vee \widehat{v}(\overline{x_2}) \vee \widehat{v}(\overline{x_3}) = \mathbf{F} \vee \mathbf{T} \vee \mathbf{T} = \mathbf{T}, \end{aligned}$$

and for the conjunction of the clauses,

$$\begin{aligned} \widehat{v}(P) &= \widehat{v}(x_1 \vee x_2 \vee x_3) \wedge \widehat{v}(\overline{x_1} \vee x_2) \wedge \widehat{v}(\overline{x_2} \vee x_3) \wedge \widehat{v}(\overline{x_3} \vee x_1) \wedge \widehat{v}(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) \\ &= \mathbf{T} \wedge \mathbf{F} \wedge \mathbf{T} \wedge \mathbf{T} \wedge \mathbf{T} = \mathbf{F}. \end{aligned}$$

Therefore, $\widehat{v}(P) = \mathbf{F}$.

Definition 13.3. We say that a truth assignment v *satisfies* a boolean formula P, if $\widehat{v}(P) = \mathbf{T}$. In this case, we also write

$$v \models P.$$

A boolean formula P is *satisfiable* if $v \models P$ for some truth assignment v, otherwise, it is *unsatisfiable*. A boolean formula P is *valid (or a tautology)* if $v \models P$ for all truth assignments v, in which case we write

$$\models P.$$

One should check that the boolean formula

$$P = \{(x_1 \vee x_2 \vee x_3), (\overline{x_1} \vee x_2), (\overline{x_2} \vee x_3), (\overline{x_3} \vee x_1), (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})\}$$

is **un**satisfiable.

One may think that it is easy to test whether a proposition is satisfiable or not. Try it, it is not that easy!

As a matter of fact, the *satisfiability problem*, testing whether a boolean formula is satisfiable, also denoted SAT, is not known to be in $\mathcal{P}$.

Moreover, it is an NP-complete problem. Most people believe that the satisfiability problem is **not** in $\mathcal{P}$, but a proof still eludes us!

Before we explain what is the class $\mathcal{NP}$, let us remark that the satisfiability problem for clauses containing at most two literals (*2-satisfiability*, or 2-SAT) is solvable in polynomial time.

The first step consists in observing that if every clause in P contains at most two literals, then we can reduce the problem to testing satisfiability when every clause has exactly two literals.

Indeed, if P contains some clause (x), then any valuation satisfying P must make x true. Then, all clauses containing x will be true, and we can delete them, whereas we can delete $\overline{x}$ from every clause containing it, since $\overline{x}$ is false.

Similarly, if P contains some clause $(\overline{x})$, then any valuation satisfying P must make x false.

Thus, in a finite number of steps, either we get the empty clause, and P is unsatisfiable, or we get a set of clauses with exactly two literals.

The number of steps is clearly linear in the number of literals in P.

For the second step, we construct a directed graph from P. The nodes of this graph are the literals in P, and edges are defined as follows:

(1) For every clause $(\overline{x} \vee y)$, there is an edge from x to y and an edge from $\overline{y}$ to $\overline{x}$.

(2) For every clause $(x \vee y)$, there is an edge from $\overline{x}$ to y and an edge from $\overline{y}$ to x

(3) For every clause $(\overline{x} \vee \overline{y})$, there is an edge from x to $\overline{y}$ and an edge from y to $\overline{x}$.

Then, it can be shown that P is unsatisfiable iff there is some x so that there is a cycle containing x and $\overline{x}$.

As a consequence, 2-satisfiability is in $\mathcal{P}$.

13.6 The Class $\mathcal{NP}$, Polynomial Reducibility, $\mathcal{NP}$-Completeness

One will observe that the hard part in trying to solve either the Hamiltonian cycle problem or the satisfiability problem, SAT, is to *find* a solution, but that *checking* that a candidate solution is indeed a solution can be done easily in polynomial time.

This is the essence of problems that can be solved *nondetermistically* in polynomial time: A solution can be guessed and then checked in polynomial time.

Definition 13.4. A nondeterministic Turing machine M is said to be *polynomially bounded* if there is a polynomial $p(X)$ so that the following holds: For every input $x \in \Sigma^*$, there is no ID ID_n so that

$$ID_0 \vdash ID_1 \vdash^* ID_{n-1} \vdash ID_n, \quad \text{with} \quad n > p(|x|),$$

where $ID_0 = q_0 x$ is the starting ID.

A language $L \subseteq \Sigma^*$ is *nondeterministic polynomially decidable* if there is a polynomially bounded nondeterministic Turing machine that accepts L. The family of all nondeterministic polynomially decidable languages is denoted by $\mathcal{NP}$.

Of course, we have the inclusion

$$\mathcal{P} \subseteq \mathcal{NP},$$

but whether or not we have equality is one of the most famous open problems of theoretical computer science and mathematics.

In fact, the question $\mathcal{P} \neq \mathcal{NP}$ is one of the open problems listed by the CLAY Institute, together with the Poincaré conjecture and the Riemann hypothesis, among other problems, and for which *one million dollar* is offered as a reward!

It is easy to check that SAT is in $\mathcal{NP}$, and so is the Hamiltonian cycle problem.

As we saw in recursion theory, where we introduced the notion of many-one reducibility, in order to compare the "degree of difficulty" of problems, it is useful to introduce the notion of reducibility and the notion of a complete set.

Definition 13.5. A function $f\colon \Sigma^* \to \Sigma^*$ is *polynomial-time computable* if there is a polynomial $p(X)$ so that the following holds: There is a deterministic Turing machine M computing it so that for every input $x \in \Sigma^*$, there is no ID ID_n so that

$$ID_0 \vdash ID_1 \vdash^* ID_{n-1} \vdash ID_n, \quad \text{with} \quad n > p(|x|),$$

where $ID_0 = q_0 x$ is the starting ID.

Given two languages $L_1, L_2 \subseteq \Sigma^*$, *a polynomial-time reduction from L_1 to L_2* is a polynomial-time computable function $f\colon \Sigma^* \to \Sigma^*$ so that for all $u \in \Sigma^*$,

$$u \in L_1 \quad \text{iff} \quad f(u) \in L_2.$$

The notation $L_1 \leq_P L_2$ is often used to denote the fact that there is polynomial-time reduction from L_1 to L_2. Sometimes, the notation $L_1 \leq_m^P L_2$ is used to stress that this is a many-to-one reduction (that is, f is not necessarily injective). This type of reduction is also known as a *Karp reduction*.

A polynomial reduction $f\colon \Sigma^* \to \Sigma^*$ from a language L_1 to a language L_2 is a method that converts in polynomial time every string $u \in \Sigma^*$ (viewed as an instance of a problem A encoded by language L_1) to a string $f(u) \in \Sigma^*$ (viewed as an instance of a problem B encoded by language L_2) in such way that membership in L_1, that is $u \in L_1$, is equivalent to membership in L_2, that is $f(u) \in L_2$.

As a consequence, if we have a procedure to decide membership in L_2 (to solve every instance of problem B), then we have a procedure for solving membership in L_1 (to solve every instance of problem A), since given any $u \in L_1$, we can first apply f to u to produce $f(u)$, and then apply our procedure to decide whether $f(u) \in L_2$; the defining property of f says that this is equivalent to deciding whether $u \in L_1$. Furthermore, if the procedure for deciding membership in L_2 runs deterministically in polynomial time, since f runs deterministically in polynomial time, so does the procedure for deciding membership in L_1, and similarly if the procedure for deciding membership in L_2 runs non deterministically in polynomial time.

For the above reason, we see that membership in L_2 can be considered at least as hard as membership in L_1, since any method for deciding membership in L_2 yields a method for deciding membership in L_1. Thus, if we view L_1 an encoding a problem A and L_2 as encoding a problem B, then B is at least as hard as A.

The following version of Proposition 10.16 for polynomial-time reducibility is easy to prove.

Proposition 13.4. *Let A, B, C be subsets of $\mathbb{N}$ (or Σ^*). The following properties hold:*

(1) If $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$.

(2) If $A \leq_P B$ then $\overline{A} \leq_P \overline{B}$.

(3) If $A \leq_P B$ and $B \in \mathcal{NP}$, then $A \in \mathcal{NP}$.

(4) If $A \leq_P B$ and $A \notin \mathcal{NP}$, then $B \notin \mathcal{NP}$.

(5) If $A \leq_P B$ and $B \in \mathcal{P}$, then $A \in \mathcal{P}$.

(6) If $A \leq_P B$ and $A \notin \mathcal{P}$, then $B \notin \mathcal{P}$.

Intuitively, we see that if L_1 is a hard problem and L_1 can be reduced to L_2 in polynomial time, then L_2 is also a hard problem.

For example, one can construct a polynomial reduction from the Hamiltonian cycle problem to the satisfiability problem SAT. Given a directed graph $G = (V, E)$ with n nodes, say $V = \{1, \ldots, n\}$, we need to construct in polynomial time a set $F = \tau(G)$ of clauses such that G has a Hamiltonian cycle iff $\tau(G)$ is satisfiable. We need to describe a permutation of the nodes that forms a Hamiltonian cycle. For this we introduce n^2 boolean variables x_{ij}, with the intended interpretation that *x_{ij} is true iff node i is the jth node in a Hamiltonian cycle.*

To express that at least one node must appear as the jth node in a Hamiltonian cycle, we have the n clauses

$$(x_{1j} \vee x_{2j} \vee \cdots \vee x_{nj}), \quad 1 \leq j \leq n. \tag{1}$$

The conjunction of these clauses is satisfied iff for every $j = 1, \ldots, n$ there is some node i which is the jth node in the cycle.

To express that only one node appears in the cycle, we have the clauses

$$(\overline{x_{ij}} \vee \overline{x_{kj}}), \quad 1 \leq i, j, k \leq n,\ i \neq k. \tag{2}$$

Since $(\overline{x_{ij}} \vee \overline{x_{kj}})$ is equivalent to $\overline{(x_{ij} \wedge x_{kj})}$, each such clause asserts that no two distinct nodes may appear as the jth node in the cycle. Let S_1 be the set of all clauses of type (1) or (2).

The conjunction of the clauses in S_1 assert that exactly one node appear at the jth node in the Hamiltonian cycle. We still need to assert that each node i appears exactly once in the cycle. For this, we have the clauses

$$(x_{i1} \vee x_{i2} \vee \cdots \vee x_{in}), \quad 1 \leq i \leq n, \tag{3}$$

and

$$(\overline{x_{ij}} \vee \overline{x_{ik}}), \quad 1 \leq i, j, k \leq n,\ j \neq k. \tag{4}$$

Let S_2 be the set of all clauses of type (3) or (4).

The conjunction of the clauses in $S_1 \cup S_2$ asserts that the x_{ij} represents a bijection of $\{1, 2, \ldots, n\}$, in the sense that for any truth assigment v satisfying all these clauses, $i \mapsto j$ iff $v(x_{ij}) = \mathbf{T}$ defines a bijection of $\{1, 2, \ldots, n\}$.

It remains to assert that this permutation of the nodes is a Hamiltonian cycle, which means that if x_{ij} and x_{kj+1} are both true then there there must be an edge (i, k). By contrapositive, this equivalent to saying that if (i, k) is *not* an edge of G, then $\overline{(x_{ij} \wedge x_{kj+1})}$ is true, which as a clause is equivalent to $(\overline{x_{ij}} \vee \overline{x_{kj+1}})$.

Therefore, for all (i, k) such that $(i, k) \notin E$ (with $i, k \in \{1, 2, \ldots, n\}$), we have the clauses

$$(\overline{x_{ij}} \vee \overline{x_{k\, j+1\ (\mathrm{mod}\ n)}}), \quad j = 1, \ldots, n. \tag{5}$$

Let S_3 be the set of clauses of type (5). The conjunction of all the clauses in $S_1 \cup S_2 \cup S_3$ is the boolean formula $F = \tau(G)$.

We leave it as an exercise to prove that G has a Hamiltonian cycle iff $F = \tau(G)$ is satisfiable.

It is also possible to construct a reduction of the satisfiability problem to the Hamiltonian cycle problem but this is harder. It is easier to construct this reduction in two steps by introducing an intermediate problem, the exact cover problem, and to provide a polynomial reduction from the satisfiability problem to the exact cover problem, and a polynomial reduction from the exact cover problem to the Hamiltonian cycle problem. These reductions are carried out in Section 14.2.

The above construction of a set $F = \tau(G)$ of clauses from a graph G asserting that G has a Hamiltonian cycle iff F is satisfiable illustrates the expressive power of propositional logic.

Remarkably, *every* language in $\mathcal{NP}$ can be reduced to SAT. Thus, SAT is a hardest problem in $\mathcal{NP}$ (Since it is in $\mathcal{NP}$).

Definition 13.6. A language L is *$\mathcal{NP}$-hard* if there is a polynomial reduction from every language $L_1 \in \mathcal{NP}$ to L. A language L is *$\mathcal{NP}$-complete* if $L \in \mathcal{NP}$ and L is $\mathcal{NP}$-hard.

Thus, an $\mathcal{NP}$-hard language is as hard to decide as any language in $\mathcal{NP}$.

Remark: There are $\mathcal{NP}$-hard languages that do not belong to $\mathcal{NP}$. Such problems are really hard. Two standard examples are K_0 and K, which encode the halting problem. Since K_0 and K are not computable, they can't be in $\mathcal{NP}$. Furthermore, since every language L in $\mathcal{NP}$ is accepted nondeterminsticaly in polynomial time $p(X)$, for some polynomial $p(X)$, for every input w we can try all computations of length at most $p(|w|)$ (there can be exponentially many, but only a finite number), so every language in $\mathcal{NP}$ is computable. Finally, it is shown in Theorem 10.17 that K_0 and K are complete with respect to many-one reducibility, so in particular they are $\mathcal{NP}$-hard. An example of a computable $\mathcal{NP}$-hard language not in $\mathcal{NP}$ will be described after Theorem 13.6.

The importance of $\mathcal{NP}$-complete problems stems from the following theorem which follows immediately from Proposition 13.4.

Theorem 13.5. *Let L be an $\mathcal{NP}$-complete language. Then, $\mathcal{P} = \mathcal{NP}$ iff $L \in \mathcal{P}$.*

There are analogies between $\mathcal{P}$ and the class of computable sets, and $\mathcal{NP}$ and the class of listable sets, but there are also important differences. One major difference is that the family of computable sets is properly contained in the family of listable sets, but it is an open problem whether $\mathcal{P}$ is properly contained in $\mathcal{NP}$. We also know that a set L is computable iff both L and $\overline{L}$ are listable, but it is also an open problem whether if both $L \in \mathcal{NP}$ and $\overline{L} \in \mathcal{NP}$, then $L \in \mathcal{P}$. This suggests defining

$$\text{co}\mathcal{NP} = \{\overline{L} \mid L \in \mathcal{NP}\},$$

that is, co$\mathcal{NP}$ consists of all complements of languages in $\mathcal{NP}$. Since $\mathcal{P} \subseteq \mathcal{NP}$ and $\mathcal{P}$ is closed under complementation,

$$\mathcal{P} \subseteq \text{co}\mathcal{NP},$$

and thus

$$\mathcal{P} \subseteq \mathcal{NP} \cap \text{co}\mathcal{NP},$$

but nobody knows whether the inclusion is proper. There are problems in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ not known to be in $\mathcal{P}$; see Section 14.3. It is unknown whether $\mathcal{NP}$ is closed under complementation, that is, nobody knows whether $\mathcal{NP} = \text{co}\mathcal{NP}$. This is considered unlikely. We will come back to co$\mathcal{NP}$ in Section 14.3.

Next, we prove a famous theorem of Steve Cook and Leonid Levin (proved independently): SAT is $\mathcal{NP}$-complete.

13.7 The Cook–Levin Theorem: SAT is $\mathcal{NP}$-Complete

Instead of showing directly that SAT is $\mathcal{NP}$-complete, which is rather complicated, we proceed in two steps, as suggested by Lewis and Papadimitriou.

(1) First, we define a tiling problem adapted from H. Wang (1961) by Harry Lewis, and we prove that it is $\mathcal{NP}$-complete.

(2) We show that the tiling problem can be reduced to SAT.

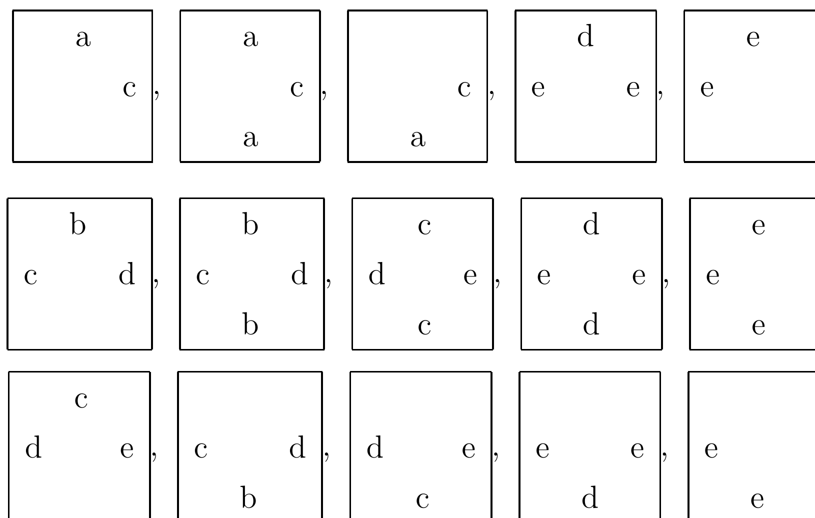
We are given a finite set $\mathcal{T} = \{t_1, \ldots, t_p\}$ of *tile patterns*, for short, *tiles*. Copies of these tile patterns may be used to tile a rectangle of predetermined size $2s \times s$ ($s > 1$). However, there are constraints on the way that these tiles may be adjacent horizontally and vertically.

The *horizontal constraints* are given by a relation $H \subseteq \mathcal{T} \times \mathcal{T}$, and the *vertical constraints* are given by a relation $V \subseteq \mathcal{T} \times \mathcal{T}$.

Thus, a *tiling system* is a triple $T = (\mathcal{T}, V, H)$ with V and H as above.

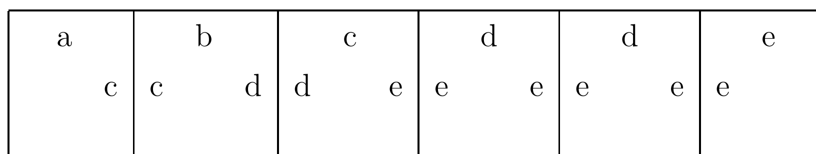
The bottom row of the rectangle of tiles is specified before the tiling process begins.

For example, consider the following tile patterns:



The horizontal and the vertical constraints are that the letters on adjacent edges match (blank edges do not match).

For $s = 3$, given the bottom row



we have the tiling shown below:

<table>
<tr><td>c
a</td><td>c d
b</td><td>d e
c</td><td>e e
d</td><td>e e
d</td><td>e
e</td></tr>
<tr><td>a
c
a</td><td>b
c d
b</td><td>c
d e
c</td><td>d
e e
d</td><td>d
e e
d</td><td>e
e
e</td></tr>
<tr><td>a
c</td><td>b
c d</td><td>c
d e</td><td>d
e e</td><td>d
e e</td><td>e
e</td></tr>
</table>

Formally, the problem is then as follows:

The Bounded Tiling Problem

Given any tiling system $(\mathcal{T}, V, H)$, any integer $s > 1$, and any initial row of tiles σ_0 (of length $2s$)

$$\sigma_0 \colon \{1, 2, \ldots, s, s+1, \ldots, 2s\} \to \mathcal{T},$$

find a $2s \times s$-tiling σ extending σ_0, i.e., a function

$$\sigma \colon \{1, 2, \ldots, s, s+1, \ldots, 2s\} \times \{1, \ldots, s\} \to \mathcal{T}$$

so that

(1) $\sigma(m, 1) = \sigma_0(m)$, for all m with $1 \leq m \leq 2s$.

(2) $(\sigma(m, n), \sigma(m+1, n)) \in H$, for all m with $1 \leq m \leq 2s - 1$, and all n, with $1 \leq n \leq s$.

(3) $(\sigma(m, n), \sigma(m, n+1)) \in V$, for all m with $1 \leq m \leq 2s$, and all n, with $1 \leq n \leq s - 1$.

Formally, an *instance of the tiling problem* is a triple $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$, where $(\mathcal{T}, V, H)$ is a tiling system, $\widehat{s}$ is the string representation of the number $s \geq 2$, in binary and σ_0 is an initial row of tiles (the bottom row).

For example, if $s = 1025$ (as a decimal number), then its binary representation is $\widehat{s} = 10000000001$. The length of $\widehat{s}$ is $\log_2 s + 1$.

Recall that the input must be a string. This is why the number s is represented by a string in binary.

If we only included a *single* tile σ_0 in position $(s+1, 1)$, then the length of the input $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$ would be $\log_2 s + C + 2$ for some constant C corresponding to the length of the string encoding $(\mathcal{T}, V, H)$.

However, the rectangular grid has size $2s^2$, which is *exponential* in the length $\log_2 s+C+2$ of the input $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$. Thus, it is impossible to check in polynomial time that a proposed solution is a tiling.

However, if we include in the input the bottom row σ_0 of length $2s$, then the size of the grid is indeed polynomial in the size of the input.

Theorem 13.6. *The tiling problem defined earlier is $\mathcal{NP}$-complete.*

Proof. Let $L \subseteq \Sigma^*$ be any language in $\mathcal{NP}$ and let u be any string in Σ^*. Assume that L is accepted in polynomial time bounded by $p(|u|)$.

We show how to construct an instance of the tiling problem, $((\mathcal{T}, V, H)_L, \widehat{s}, \sigma_0)$, where $s = p(|u|) + 2$, and where the bottom row encodes the starting ID, so that $u \in L$ iff the tiling problem $((\mathcal{T}, V, H)_L, \widehat{s}, \sigma_0)$ has a solution.

First, note that the problem is indeed in $\mathcal{NP}$, since we have to guess a rectangle of size $2s^2$, and that checking that a tiling is legal can indeed be done in $O(s^2)$, where s is *bounded by the the size of the input* $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$, since the input contains the bottom row of $2s$ symbols (this is the reason for including the bottom row of $2s$ tiles in the input!).

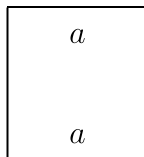
The idea behind the definition of the tiles is that, in a solution of the tiling problem, the labels on the horizontal edges between two adjacent rows represent a legal ID, $upav$.

In a given row, the labels on vertical edges of adjacent tiles keep track of the change of state and direction.

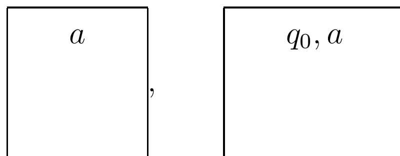
Let Γ be the tape alphabet of the TM, M. As before, we assume that M signals that it accepts u by halting with the output 1 (true).

From M, we create the following tiles:

(1) For every $a \in \Gamma$, tiles

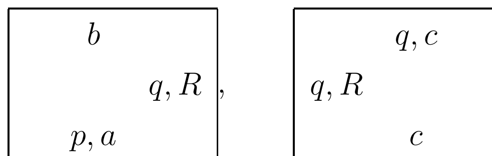


(2) For every $a \in \Gamma$, the bottom row uses tiles

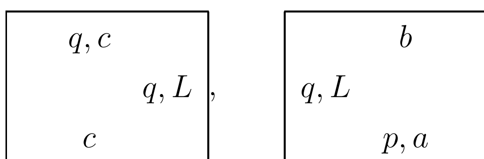


where q_0 is the start state.

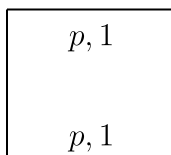
(3) For every instruction $(p, a, b, R, q) \in \delta$, for every $c \in \Gamma$, tiles



(4) For every instruction $(p, a, b, L, q) \in \delta$, for every $c \in \Gamma$, tiles



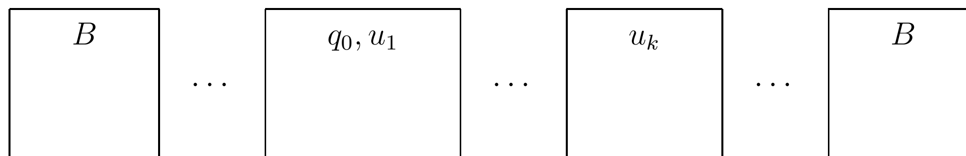
(5) For every halting state, p, tiles



The purpose of tiles of type (5) is to fill the $2s \times s$ rectangle iff M accepts u. Since $s = p(|u|) + 2$ and the machine runs for at most $p(|u|)$ steps, the $2s \times s$ rectangle can be tiled iff $u \in L$.

The vertical and the horizontal constraints are that adjacent edges have the same label (or no label).

If $u = u_1 \cdots u_k$, the initial bottom row σ_0, of length $2s$, is:



where the tile labeled q_0, u_1 is in position $s + 1$.

The example below illustrates the construction:

B B	$\cdots$	B f, R q, c	$f, 1$ f, R 1	$\cdots$	B B
B B	$\cdots$	q, c q, L c	1 q, L p, a	$\cdots$	B B
B B	$\cdots$	c p, R r, b	p, a p, R a	$\cdots$	B B

We claim that $u = u_1 \cdots u_k$ is accepted by M iff the tiling problem just constructed has a solution.

The upper horizontal edge of the first (bottom) row of tiles represents the starting configuation $B^s q_0 u B^{s-|u|}$. By induction, we see that after i ($i \leq p(|u|) = s - 2$) steps the upper horizontal edge of the $(i+1)$th row of tiles represents the current ID *upav* reached by the Turing machine. Since the machine runs for at most $p(|u|)$ steps and since $s = p(|u|) + 2$, when the computation stops, at most the lowest $p(|u|) + 1 = s - 1$ rows of the the $2s \times s$ rectangle have been tiled. Assume the machine M stops after $r \leq s - 2$ steps. Then the lowest $r + 1$ rows have been tiled, and since no further instruction can be executed (since the machine entered a halting state), the remaining $s - r - 1$ rows can be filled iff tiles of type (5) can be used iff the machine stopped in an ID containing a pair $p\,1$ where p is a halting state. Therefore, the machine M accepts u iff the $2s \times s$ rectangle can be tiled. $\square$

Remarks.

(1) The problem becomes harder if we only specify a *single* tile σ_0 as input, instead of a row of length $2s$. If s is specified in binary (or any other base, but not in tally notation), then the $2s^2$ grid has size exponential in the length $\log_2 s + C + 2$ of the input $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$, and this tiling problem is actually $\mathcal{NEXP}$-complete! The class $\mathcal{NEXP}$ is the family of languages that can be accepted by a nondeterministic Turing machine that runs in time bounded by $2^{p(|x|)}$, for every x, where p is a polynomial; see the remark after Definition 14.4. By the time hierarchy theorem (Cook, Seiferas, Fischer, Meyer, Zak), it is known that $\mathcal{NP}$ is properly contained in $\mathcal{NEXP}$; see Papadimitriou [14] (Chapters 7 and 20) and Arora and Barak [2] (Chapter 3, Section 3.2). Then the tiling problem with a single tile as input is a computable $\mathcal{NP}$-hard problem not in $\mathcal{NP}$.

(2) If we relax the finiteness condition and require that the entire upper half-plane be tiled, i.e., for every $s > 1$, there is a solution to the $2s \times s$-tiling problem, then the problem is undecidable.

In 1972, Richard Karp published a list of 21 $\mathcal{NP}$-complete problems.

We finally prove the Cook-Levin theorem.

Theorem 13.7. *(Cook, 1971, Levin, 1973) The satisfiability problem* SAT *is $\mathcal{NP}$-complete.*

Proof. We reduce the tiling problem to SAT. Given a tiling problem, $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$, we introduce boolean variables

$$x_{mnt},$$

for all m with $1 \leq m \leq 2s$, all n with $1 \leq n \leq s$, and all tiles $t \in \mathcal{T}$.

The intuition is that $x_{mnt} = \mathbf{T}$ iff tile t occurs in some tiling σ so that $\sigma(m, n) = t$.

We define the following clauses:

(1) For all m, n in the correct range, as above,

$$(x_{mnt_1} \vee x_{mnt_2} \vee \cdots \vee x_{mnt_p}),$$

for all p tiles in $\mathcal{T}$.

This clause states that every position in σ is tiled.

(2) For any two distinct tiles $t \neq t' \in \mathcal{T}$, for all m, n in the correct range, as above,

$$(\overline{x}_{mnt} \vee \overline{x}_{mnt'}).$$

This clause states that a position may not be occupied by more than one tile.

(3) For every pair of tiles $(t, t') \in \mathcal{T} \times \mathcal{T} - H$, for all m with $1 \leq m \leq 2s - 1$, and all n, with $1 \leq n \leq s$,

$$(\overline{x}_{mnt} \vee \overline{x}_{m+1\,nt'}).$$

This clause enforces the horizontal adjacency constraints.

(4) For every pair of tiles $(t, t') \in \mathcal{T} \times \mathcal{T} - V$, for all m with $1 \leq m \leq 2s$, and all n, with $1 \leq n \leq s - 1$,

$$(\overline{x}_{mnt} \vee \overline{x}_{m\,n+1\,t'}).$$

This clause enforces the vertical adjacency constraints.

(5) For all m with $1 \leq m \leq 2s$,

$$(x_{m1\sigma_0(m)}).$$

This clause states that the bottom row is correctly tiled with σ_0.

It is easily checked that the tiling problem has a solution iff the conjunction of the clauses just defined is satisfiable. Thus, SAT is $\mathcal{NP}$-complete. $\square$

We sharpen Theorem 13.7 to prove that 3-SAT is also $\mathcal{NP}$-complete. This is the satisfiability problem for clauses containing at most three literals.

We know that we can't go further and retain
$\mathcal{NP}$-completeteness, since 2-SAT is in $\mathcal{P}$.

Theorem 13.8. *(Cook, 1971) The satisfiability problem* 3-SAT *is $\mathcal{NP}$-complete.*

Proof. We have to break "long clauses"

$$C = (L_1 \vee \cdots \vee L_k),$$

i.e., clauses containing $k \geq 4$ literals, into clauses with at most three literals, in such a way that satisfiability is preserved.

For example, consider the following clause with $k = 6$ literals:

$$C = (L1 \vee L2 \vee L3 \vee L4 \vee L5 \vee L6).$$

We create 3 new boolean variables y_1, y_2, y_3, and the 4 clauses

$$(L_1 \vee L_2 \vee y_1),\ (\overline{y_1} \vee L_3 \vee y_2),\ (\overline{y_2} \vee L_4 \vee y_3),\ (\overline{y}_3 \vee L_5 \vee L_6).$$

Let C' be the conjunction of these clauses. We claim that C is satisfiable iff C' is.

Assume that C' is satisfiable but C is not. If so, in any truth assigment v, $v(L_i) = \mathbf{F}$, for $i = 1, 2, \ldots, 6$. To satisfy the first clause, we must have $v(y_1) = \mathbf{T}$., Then to satisfy the second clause, we must have $v(y_2) = \mathbf{T}$, and similarly satisfy the third clause, we must have $v(y_3) = \mathbf{T}$. However, since $v(L_5) = \mathbf{F}$ and $v(L_6) = \mathbf{F}$, the only way to satisfy the fourth clause is to have $v(y_3) = \mathbf{F}$, contradicting that $v(y_3) = \mathbf{T}$. Thus, C is indeed satisfiable.

Let us now assume that C is satisfiable. This means that there is a smallest index i such that L_i is satisfied.

Say $i = 1$, so $v(L_1) = \mathbf{T}$. Then if we let $v(y_1) = v(y_2) = v(y_3) = \mathbf{F}$, we see that C' is satisfied.

Say $i = 2$, so $v(L_1) = \mathbf{F}$ and $v(L_2) = \mathbf{T}$. Again if we let $v(y_1) = v(y_2) = v(y_3) = \mathbf{F}$, we see that C' is satisfied.

Say $i = 3$, so $v(L_1) = \mathbf{F}$, $v(L_2) = \mathbf{F}$, and $v(L_3) = \mathbf{T}$. If we let $v(y_1) = \mathbf{T}$ and $v(y_2) = v(y_3) = \mathbf{F}$, we see that C' is satisfied.

Say $i = 4$, so $v(L_1) = \mathbf{F}$, $v(L_2) = \mathbf{F}$, $v(L_3) = \mathbf{F}$, and $v(L_4) = \mathbf{T}$. If we let $v(y_1) = \mathbf{T}$, $v(y_2) = \mathbf{T}$ and $v(y_3) = \mathbf{F}$, we see that C' is satisfied.

Say $i = 5$, so $v(L_1) = \mathbf{F}$, $v(L_2) = \mathbf{F}$, $v(L_3) = \mathbf{F}$, $v(L_4) = \mathbf{F}$, and $v(L_5) = \mathbf{T}$. If we let $v(y_1) = \mathbf{T}$, $v(y_2) = \mathbf{T}$ and $v(y_3) = \mathbf{T}$, we see that C' is satisfied.

Say $i = 6$, so $v(L_1) = \mathbf{F}$, $v(L_2) = \mathbf{F}$, $v(L_3) = \mathbf{F}$, $v(L_4) = \mathbf{F}$, $v(L_5) = \mathbf{F}$, and $v(L_6) = \mathbf{T}$. Again, if we let $v(y_1) = \mathbf{T}$, $v(y_2) = \mathbf{T}$ and $v(y_3) = \mathbf{T}$, we see that C' is satisfied.

Therefore if C is satisfied, then C' is satisfied in all cases.

In general, for every long clause, create $k-3$ new boolean variables $y_1, \ldots y_{k-3}$, and the $k-2$ clauses

$$\begin{aligned}&(L_1 \vee L_2 \vee y_1), (\overline{y_1} \vee L_3 \vee y_2), (\overline{y_2} \vee L_4 \vee y_3), \cdots,\\ &(\overline{y}_{k-4} \vee L_{k-2} \vee y_{k-3}), (\overline{y}_{k-3} \vee L_{k-1} \vee L_k).\end{aligned}$$

Let C' be the conjunction of these clauses. We claim that C is satisfiable iff C' is.

Assume that C' is satisfiable, but that C is not. Then, for every truth assignment v, we have $v(L_i) = \mathbf{F}$, for $i = 1, \ldots, k$.

However, C' is satisfied by some v, and the only way this can happen is that $v(y_1) = \mathbf{T}$, to satisfy the first clause. Then, $v(\overline{y_1}) = \mathbf{F}$, and we must have $v(y_2) = \mathbf{T}$, to satisfy the second clause.

By induction, we must have $v(y_{k-3}) = \mathbf{T}$, to satisfy the next to the last clause. However, the last clause is now false, a contradiction.

Thus, if C' is satisfiable, then so is C.

Conversely, assume that C is satisfiable. If so, there is some truth assignment, v, so that $v(C) = \mathbf{T}$, and thus, there is a smallest index i, with $1 \leq i \leq k$, so that $v(L_i) = \mathbf{T}$ (and so, $v(L_j) = \mathbf{F}$ for all $j < i$).

Let v' be the assignment extending v defined so that

$$v'(y_j) = \mathbf{F} \quad \text{if} \quad \max\{1, i-1\} \leq j \leq k-3,$$

and $v'(y_j) = \mathbf{T}$, otherwise.

It is easily checked that $v'(C') = \mathbf{T}$. ∎

Another version of 3-SAT can be considered, in which every clause has exactly three literals. We will call this the problem *exact* 3-SAT.

Theorem 13.9. *(Cook, 1971) The satisfiability problem for exact* 3-SAT *is $\mathcal{NP}$-complete.*

Proof. A clause of the form (L) is satisfiable iff the following four clauses are satisfiable:

$$(L \vee u \vee v), (L \vee \overline{u} \vee v), (L \vee u \vee \overline{v}), (L \vee \overline{u} \vee \overline{v}).$$

A clause of the form $(L_1 \vee L_2)$ is satisfiable iff the following two clauses are satisfiable:

$$(L_1 \vee L_2 \vee u), (L_1 \vee L_2 \vee \overline{u}).$$

Thus, we have a reduction of 3-SAT to exact 3-SAT. $\square$

We now make some remarks on the conversion of propositions to CNF.

Recall that the set of propositions (over the connectives $\vee$, $\wedge$, and $\neg$) is defined inductively as follows:

(1) Every propositional letter, $x \in \mathbf{PV}$, is a proposition (an *atomic* proposition).

(2) If A is a proposition, then $\neg A$ is a proposition.

(3) If A and B are propositions, then $(A \vee B)$ is a proposition.

(4) If A and B are propositions, then $(A \wedge B)$ is a proposition.

Two propositions A and B are *equivalent*, denoted $A \equiv B$, if

$$v \models A \quad \text{iff} \quad v \models B$$

for all truth assignments, v.

It is easy to show that $A \equiv B$ iff the proposition

$$(\neg A \vee B) \wedge (\neg B \vee A)$$

is valid.

Every proposition, A, is equivalent to a proposition, A', in CNF.

There are several ways of proving this fact. One method is algebraic, and consists in using the algebraic laws of boolean algebra.

First, one may convert a proposition to *negation normal form*, or *nnf*. A proposition is in nnf if occurrences of $\neg$ only appear in front of propositional variables, but not in front of compound propositions.

Any proposition can be converted to an equivalent one in nnf by using the de Morgan laws:

$$\begin{aligned}
\neg(A \vee B) &\equiv (\neg A \wedge \neg B)\\
\neg(A \wedge B) &\equiv (\neg A \vee \neg B)\\
\neg\neg A &\equiv A.
\end{aligned}$$

Then, a proposition in nnf can be converted to CNF, but the question of uniqueness of the CNF is a bit tricky.

For example, the proposition

$$A = (u \land (x \lor y)) \lor (\neg u \land (x \lor y))$$

has

$$\begin{aligned} A_1 &= (u \lor x \lor y) \land (\neg u \lor x \lor y) \\ A_2 &= (u \lor \neg u) \land (x \lor y) \\ A_3 &= x \lor y, \end{aligned}$$

as equivalent propositions in CNF!

We can get a *unique* CNF equivalent to a given proposition if we do the following:

(1) Let $\mathrm{Var}(A) = \{x_1, \ldots, x_m\}$ be the set of variables occurring in A.

(2) Define a *maxterm w.r.t.* $\mathrm{Var}(A)$ as any disjunction of m pairwise distinct literals formed from $\mathrm{Var}(A)$, and not containing both some variable x_i and its negation $\neg x_i$.

(3) Then, it can be shown that for any proposition A that is not a tautology, there is a *unique* proposition in CNF *equivalent* to A, whose clauses consist of maxterms formed from $\mathrm{Var}(A)$.

The above definition can yield strange results. For instance, the CNF of any unsatisfiable proposition with m distinct variables is the conjunction of all of its 2^m maxterms!

The above notion does not cope well with minimality.

For example, according to the above, the CNF of

$$A = (u \land (x \lor y)) \lor (\neg u \land (x \lor y))$$

should be

$$A_1 = (u \lor x \lor y) \land (\neg u \lor x \lor y).$$

There are also propositions such that any equivalent proposition in CNF has size exponential in terms of the original proposition.

Here is such an example:

$$A = (x_1 \land x_2) \lor (x_3 \land x_4) \lor \cdots \lor (x_{2n-1} \land x_{2n}).$$

Observe that it is in DNF.

We will prove a little later that any CNF for A contains 2^n occurrences of variables.

A nice method to convert a proposition in nnf to CNF is to construct a tree whose nodes are labeled with sets of propositions using the following (Gentzen-style) rules :

$$\frac{P, \Delta \qquad Q, \Delta}{(P \wedge Q), \Delta}$$

and

$$\frac{P, Q, \Delta}{(P \vee Q), \Delta}$$

where Δ stands for any set of propositions (even empty), and the comma stands for union. Thus, it is assumed that $(P \wedge Q) \notin \Delta$ in the first case, and that $(P \vee Q) \notin \Delta$ in the second case.

Since we interpret a set, Γ, of propositions as a disjunction, a valuation, v, satisfies Γ iff it satisfies *some* proposition in Γ.

Observe that a valuation v satisfies the conclusion of a rule iff it satisfies both premises in the first case, and the single premise in the second case.

Using these rules, we can build a finite tree whose leaves are labeled with sets of literals.

By the above observation, a valuation v satisfies the proposition labeling the root of the tree iff it satisfies all the propositions labeling the leaves of the tree.

But then, a CNF for the original proposition A (in nnf, at the root of the tree) is the conjunction of the clauses appearing as the leaves of the tree.

We may exclude the clauses that are tautologies, and we may discover in the process that A is a tautology (when all leaves are tautologies).

Going back to our "bad" proposition, A, by induction, we see that any tree for A has 2^n leaves.

However, it should be noted that for any proposition, A, we can construct in polynomial time a formula, A', in CNF, so that A is satisfiable iff A' is satisfiable, by creating *new* variables.

We proceed recursively. The trick is that we replace

$$(C_1 \wedge \cdots \wedge C_m) \vee (D_1 \wedge \cdots \wedge D_n)$$

by

$$(C_1 \vee y) \wedge \cdots \wedge (C_m \vee y) \wedge (D_1 \vee \overline{y}) \wedge \cdots \wedge (D_n \vee \overline{y}),$$

where the C_i's and the D_j's are clauses, and y is a new variable.

It can be shown that the number of new variables required is at most quadratic in the size of A.

Warning: In general, the proposition A' is *not* equivalent to the proposition A.

Rules for dealing for $\neg$ can also be created. In this case, we work with pairs of sets of propositions,

$$\Gamma \to \Delta,$$

where, the propositions in Γ are interpreted conjunctively, and the propositions in Δ are interpreted disjunctively.

We obtain a sound and complete proof system for propositional logic (a "Gentzen-style" proof system, see Gallier's *Logic for Computer Science*).

Chapter 14

Some $\mathcal{NP}$-Complete Problems

14.1 Statements of the Problems

In this chapter we will show that certain classical algorithmic problems are $\mathcal{NP}$-complete. This chapter is heavily inspired by Lewis and Papadimitriou's excellent treatment [11]. In order to study the complexity of these problems in terms of resource (time or space) bounded Turing machines (or RAM programs), it is crucial to be able to encode instances of a problem P as strings in a language L_P. Then an instance of a problem P is solvable iff the corresponding string belongs to the language L_P. This implies that our problems must have a yes–no answer, which is not always the usual formulation of optimization problems where what is required is to find some *optimal* solution, that is, a solution minimizing or maximizing so objective (cost) function F. For example the standard formulation of the traveling salesman problem asks for a tour (of the cities) of minimal cost.

Fortunately, there is a trick to reformulate an optimization problem as a yes–no answer problem, which is to explicitly incorporate a *budget* (or *cost*) term B into the problem, and instead of asking whether some objective function F has a minimum or a maximum w, we ask whether there is a solution w such that $F(w) \leq B$ in the case of a minimum solution, or $F(w) \geq B$ in the case of a maximum solution.

If we are looking for a minimum of F, we try to guess the minimum value B of F and then we solve the problem of finding w such that $F(w) \leq B$. If our guess for B is too small, then we fail. In this case, we try again with a larger value of B. Otherwise, if B was not too small we find some w such that $F(w) \leq B$, but w may not correspond to a minimum of F, so we try again with a smaller value of B, and so on. This yields an approximation method to find a minimum of F.

Similarly, if we are looking for a maximum of F, we try to guess the maximum value B of F and then we solve the problem of finding w such that $F(w) \geq B$. If our guess for B is too large, then we fail. In this case, we try again with a smaller value of B. Otherwise, if B was not too large we find some w such that $F(w) \geq B$, but w may not correspond to a maximum of F, so we try again with a greater value of B, and so on. This yields an

approximation method to find a maximum of F.

We will see several examples of this technique in Problems 5–8 listed below.

The problems that will consider are

(1) Exact Cover

(2) Hamiltonian Cycle for directed graphs

(3) Hamiltonian Cycle for undirected graphs

(4) The Traveling Salesman Problem

(5) Independent Set

(6) Clique

(7) Node Cover

(8) Knapsack, also called subset sum

(9) Inequivalence of $*$-free Regular Expressions

(10) The 0-1-integer programming problem

We begin by describing each of these problems.

(1) **Exact Cover**

We are given a finite nonempty set $U = \{u_1, \ldots, u_n\}$ (the universe), and a family $\mathcal{F} = \{S_1, \ldots, S_m\}$ of $m \geq 1$ nonempty subsets of U. The question is whether there is an *exact cover*, that is, a subfamily $\mathcal{C} \subseteq \mathcal{F}$ of subsets in $\mathcal{F}$ such that the sets in $\mathcal{C}$ are disjoint and their union is equal to U.

For example, let $U = \{u_1, u_2, u_3, u_4, u_5, u_6\}$, and let $\mathcal{F}$ be the family

$$\mathcal{F} = \{\{u_1, u_3\}, \{u_2, u_3, u_6\}, \{u_1, u_5\}, \{u_2, u_3, u_4\}, \{u_5, u_6\}, \{u_2, u_4\}\}.$$

The subfamily

$$\mathcal{C} = \{\{u_1, u_3\}, \{u_5, u_6\}, \{u_2, u_4\}\}$$

is an exact cover.

It is easy to see that **Exact Cover** is in $\mathcal{NP}$. To prove that it is $\mathcal{NP}$-complete, we will reduce the **Satisfiability Problem** to it. This means that we provide a method running in polynomial time that converts every instance of the **Satisfiability Problem** to an instance of **Exact Cover**, such that the first problem has a solution iff the converted problem has a solution.

(2) **Hamiltonian Cycle (for Directed Graphs)**

Recall that a *directed graph* G is a pair $G = (V, E)$, where $E \subseteq V \times V$. Elements of V are called *nodes* (or *vertices*). A pair $(u, v) \in E$ is called an *edge* of G. We will restrict ourselves to *simple graphs*, that is, graphs without edges of the form (u, u); equivalently, $G = (V, E)$ is a simple graph if whenever $(u, v) \in E$, then $u \neq v$.

Given any two nodes $u, v \in V$, a *path from u to v* is any sequence of $n+1$ edges $(n \geq 0)$

$$(u, v_1), (v_1, v_2), \ldots, (v_n, v).$$

(If $n = 0$, a path from u to v is simply a single edge, (u, v).)

A directed graph G is *strongly connected* if for every pair $(u, v) \in V \times V$, there is a path from u to v. A *closed path, or cycle*, is a path from some node u to itself. We will restrict out attention to finite graphs, i.e. graphs (V, E) where V is a finite set.

Definition 14.1. Given a directed graph G, a *Hamiltonian cycle* is a cycle that passes through all the nodes exactly once (note, some edges may not be traversed at all).

Hamiltonian Cycle Problem (for Directed Graphs): Given a directed graph G, is there an Hamiltonian cycle in G?

Is there is a Hamiltonian cycle in the directed graph D shown in Figure 14.1?

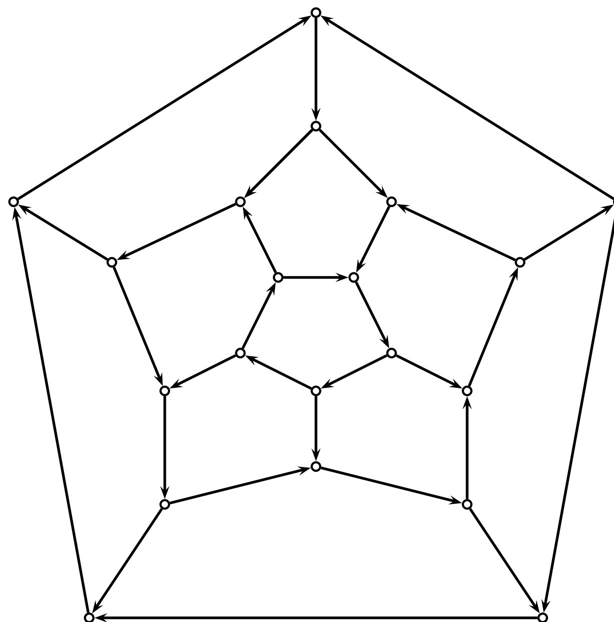


Figure 14.1: A tour "around the world."

Finding a Hamiltonian cycle in this graph does not appear to be so easy! A solution is shown in Figure 14.2 below.

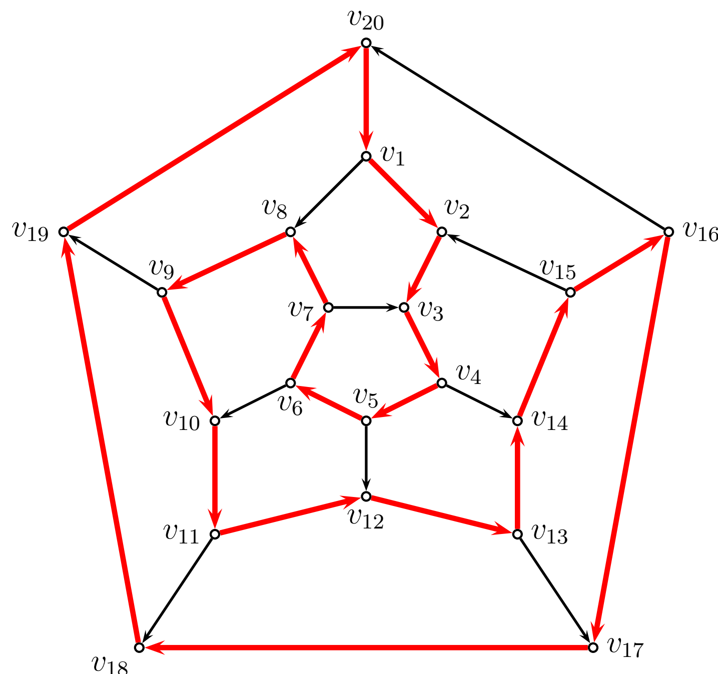


Figure 14.2: A Hamiltonian cycle in D.

It is easy to see that **Hamiltonian Cycle (for Directed Graphs)** is in $\mathcal{NP}$. To prove that it is $\mathcal{NP}$-complete, we will reduce **Exact Cover** to it. This means that we provide a method running in polynomial time that converts every instance of **Exact Cover** to an instance of **Hamiltonian Cycle (for Directed Graphs)** such that the first problem has a solution iff the converted problem has a solution. This is perphaps the hardest reduction.

(3) **Hamiltonian Cycle (for Undirected Graphs)**

Recall that an *undirected graph* G is a pair $G = (V, E)$, where E is a set of subsets $\{u, v\}$ of V consisting of exactly two distinct elements. Elements of V are called *nodes* (or *vertices*). A pair $\{u, v\} \in E$ is called an *edge* of G.

Given any two nodes $u, v \in V$, a *path from* u *to* v is any sequence of n nodes ($n \geq 2$)

$$u = u_1, u_2, \ldots, u_n = v$$

such that $\{u_i, u_{i+1}\} \in E$ for $i = 1, \ldots, n-1$. (If $n = 2$, a path from u to v is simply a single edge, $\{u, v\}$.)

An undirected graph G is *connected* if for every pair $(u, v) \in V \times V$, there is a path from u to v. A *closed path, or cycle*, is a path from some node u to itself.

Definition 14.2. Given an undirected graph G, a *Hamiltonian cycle* is a cycle that passes through all the nodes exactly once (note, some edges may not be traversed at all).

Hamiltonian Cycle Problem (for Undirected Graphs): Given an undirected graph G, is there an Hamiltonian cycle in G?

An instance of this problem is obtained by changing every directed edge in the directed graph of Figure 14.1 to an undirected edge. The directed Hamiltonian cycle given in Figure 14.2 is also an undirected Hamiltonian cycle of the undirected graph of Figure 14.3.

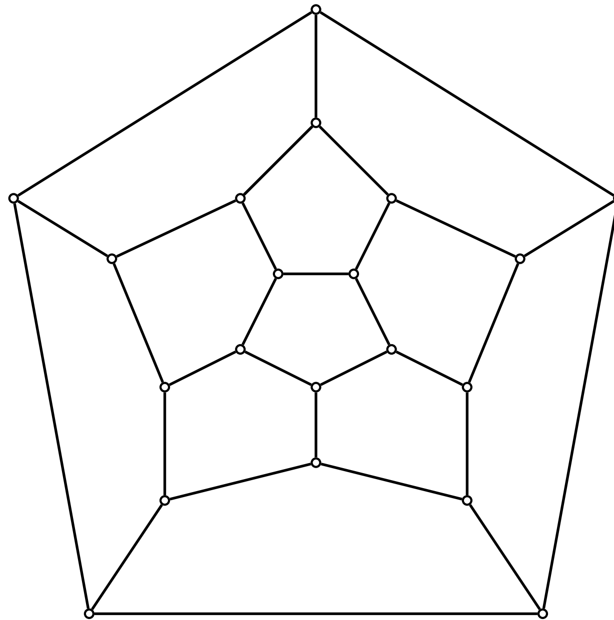


Figure 14.3: A tour "around the world," undirected version.

We see immediately that **Hamiltonian Cycle (for Undirected Graphs)** is in $\mathcal{NP}$. To prove that it is $\mathcal{NP}$-complete, we will reduce **Hamiltonian Cycle (for Directed Graphs)** to it. This means that we provide a method running in polynomial time that converts every instance of **Hamiltonian Cycle (for Directed Graphs)** to an instance of **Hamiltonian Cycle (for Undirected Graphs)** such that the first problem has a solution iff the converted problem has a solution. This is an easy reduction.

(4) **Traveling Salesman Problem**

We are given a set $\{c_1, c_2, \ldots, c_n\}$ of $n \geq 2$ cities, and an $n \times n$ matrix $D = (d_{ij})$ of nonnegative integers, where d_{ij} is the *distance* (or *cost*) of traveling from city c_i to city c_j. We assume that $d_{ii} = 0$ and $d_{ij} = d_{ji}$ for all i, j, so that the matrix D is symmetric and has zero diagonal.

Traveling Salesman Problem: Given some $n \times n$ matrix $D = (d_{ij})$ as above and some integer $B \geq 0$ (the *budget* of the traveling salesman), find a permutation π of $\{1, 2, \ldots, n\}$ such that

$$c(\pi) = d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + \cdots + d_{\pi(n-1)\pi(n)} + d_{\pi(n)\pi(1)} \leq B.$$

The quantity $c(\pi)$ is the *cost* of the trip specified by π. The Traveling Salesman Problem has been stated in terms of a budget so that it has a yes or no answer, which allows us to convert it into a language. A minimal solution corresponds to the smallest feasible value of B.

Example 14.1. Consider the 4×4 symmetric matrix given by

$$D = \begin{pmatrix} 0 & 2 & 1 & 1 \\ 2 & 0 & 1 & 1 \\ 1 & 1 & 0 & 3 \\ 1 & 1 & 3 & 0 \end{pmatrix},$$

and the budget $B = 4$. The tour specified by the permutation

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 2 & 3 \end{pmatrix}$$

has cost 4, since

$$\begin{aligned} c(\pi) &= d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + d_{\pi(3)\pi(4)} + d_{\pi(4)\pi(1)} \\ &= d_{14} + d_{42} + d_{23} + d_{31} \\ &= 1 + 1 + 1 + 1 = 4. \end{aligned}$$

The cities in this tour are traversed in the order

$$(1, 4, 2, 3, 1).$$

It is clear that the **Traveling Salesman Problem** is in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce the **Hamiltonian Cycle Problem (Undirected Graphs)** to it. This means that we provide a method running in polynomial time that converts every instance of **Hamiltonian Cycle Problem (Undirected Graphs)** to an instance of the **Traveling Salesman Problem** such that the first problem has a solution iff the converted problem has a solution.

(5) **Independent Set**

The problem is this: Given an undirected graph $G = (V, E)$ and an integer $K \geq 2$, is there a set C of nodes with $|C| \geq K$ such that for all $v_i, v_j \in C$, there is *no* edge $\{v_i, v_j\} \in E$?

A maximal independent set with 3 nodes is shown in Figure 14.4. A maximal solution

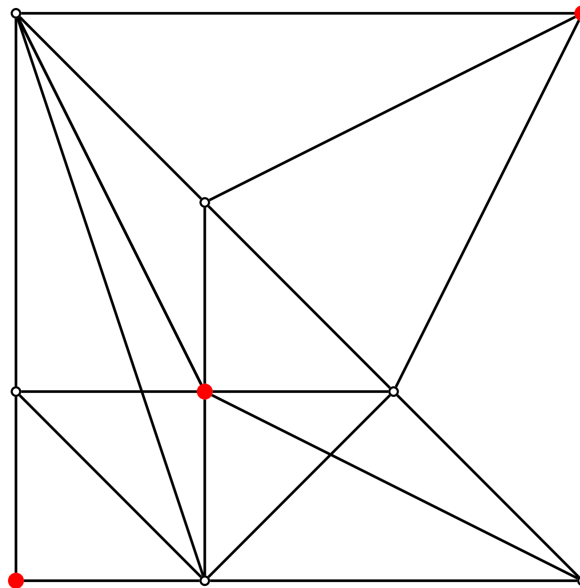


Figure 14.4: A maximal Independent Set in a graph

corresponds to the largest feasible value of K. The problem **Independent Set** is obviously in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce **Exact 3-Satisfiability** to it. This means that we provide a method running in polynomial time that converts every instance of **Exact 3-Satisfiability** to an instance of **Independent Set** such that the first problem has a solution iff the converted problem has a solution.

(6) **Clique**

The problem is this: Given an undirected graph $G = (V, E)$ and an integer $K \geq 2$, is there a set C of nodes with $|C| \geq K$ such that for all $v_i, v_j \in C$, there is *some* edge $\{v_i, v_j\} \in E$? Equivalently, does G contain a complete subgraph with at least K nodes?

A maximal clique with 4 nodes is shown in Figure 14.5. A maximal solution corresponds to the largest feasible value of K. The problem **Clique** is obviously in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce **Independent Set** to it. This means that we provide a method running in polynomial time that converts every instance of **Independent**

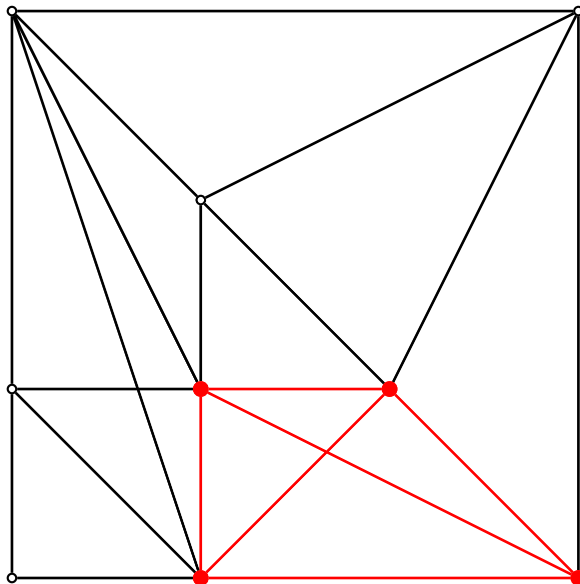


Figure 14.5: A maximal Clique in a graph

Set to an instance of **Clique** such that the first problem has a solution iff the converted problem has a solution.

(7) **Node Cover**

The problem is this: Given an undirected graph $G = (V, E)$ and an integer $B \geq 2$, is there a set C of nodes with $|C| \leq B$ such that C covers all edges in G, which means that for every edge $\{v_i, v_j\} \in E$, either $v_i \in C$ or $v_j \in C$?

A minimal node cover with 6 nodes is shown in Figure 14.6. A minimal solution corresponds to the smallest feasible value of B. The problem **Node Cover** is obviously in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce **Independent Set** to it. This means that we provide a method running in polynomial time that converts every instance of **Independent Set** to an instance of **Node Cover** such that the first problem has a solution iff the converted problem has a solution.

The Node Cover problem has the following interesting interpretation: think of the nodes of the graph as rooms of a museum (or art gallery *etc.*), and each edge as a straight corridor that joins two rooms. Then Node Cover may be useful in assigning as few as possible guards to the rooms, so that all corridors can be seen by a guard.

(8) **Knapsack (also called Subset sum)**

The problem is this: Given a finite nonempty set $S = \{a_1, a_2, \ldots, a_n\}$ of nonnegative integers, and some integer $K \geq 0$, all represented in binary, is there a nonempty subset

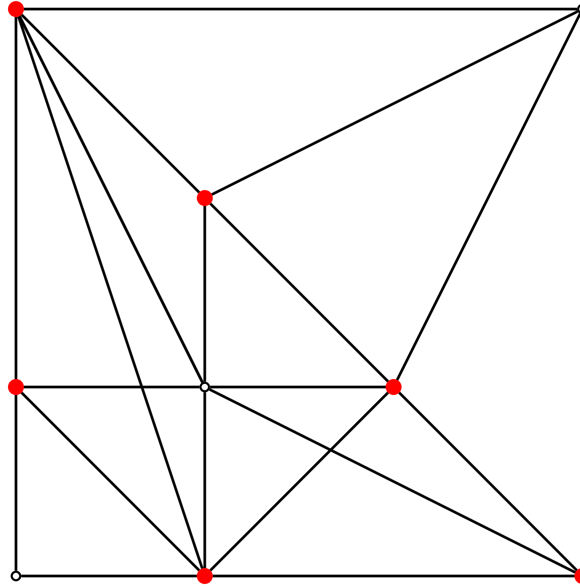


Figure 14.6: A minimal Node Cover in a graph

$I \subseteq \{1, 2, \ldots, n\}$ such that

$$\sum_{i \in I} a_i = K?$$

A "concrete" realization of this problem is that of a hiker who is trying to fill her/his backpack to its maximum capacity with items of varying weights or values.

It is easy to see that the **Knapsack** Problem is in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce **Exact Cover** to it. This means that we provide a method running in polynomial time that converts every instance of **Exact Cover** to an instance of **Knapsack** Problem such that the first problem has a solution iff the converted problem has a solution.

Remark: The **0-1 Knapsack Problem** is defined as the following problem. Given a set of n items, numbered from 1 to n, each with a weight $w_i \in \mathbb{N}$ and a value $v_i \in \mathbb{N}$, given a maximum capacity $W \in \mathbb{N}$ and a budget $B \in \mathbb{N}$, is there a set of n variables $x_1, \ldots, x_n$ with $x_i \in \{0, 1\}$ such that

$$\sum_{i=1}^{n} x_i v_i \geq B,$$

$$\sum_{i=1}^{n} x_i w_i \leq W.$$

Informally, the problem is to pick items to include in the knapsack so that the sum of the values exceeds a given minimum B (the goal is to maximize this sum), and the sum of the weights is less than or equal to the capacity W of the knapsack. A maximal solution corresponds to the largest feasible value of B.

The **Knapsack** Problem as we defined it (which is how Lewis and Papadimitriou define it) is the special case where $v_i = w_i = 1$ for $i = 1, \ldots, n$ and $W = B$. For this reason, it is also called the **Subset Sum** Problem. Clearly, the **Knapsack** (**Subset Sum**) Problem reduces to the **0-1 Knapsack** Problem, and thus the **0-1 Knapsack** Problem is also NP-complete.

(9) **Inequivalence of $*$-free Regular Expressions**

Recall that the problem of deciding the equivalence $R_1 \cong R_2$ of two regular expressions R_1 and R_2 is the problem of deciding whether R_1 and R_2 define the same language, that is, $\mathcal{L}[R_1] = \mathcal{L}[R_2]$. Is this problem in $\mathcal{NP}$?

In order to show that the equivalence problem for regular expressions is in $\mathcal{NP}$ we would have to be able to somehow check in polynomial time that two expressions define the same language, but this is still an open problem.

What might be easier is to decide whether two regular expressions R_1 and R_2 are *inequivalent*. For this, we just have to find a string w such that either $w \in \mathcal{L}[R_1] - \mathcal{L}[R_2]$ or $w \in \mathcal{L}[R_2] - \mathcal{L}[R_1]$. The problem is that if we can guess such a string w, we still have to check in polynomial time that $w \in (\mathcal{L}[R_1] - \mathcal{L}[R_2]) \cup (\mathcal{L}[R_2] - \mathcal{L}[R_1])$, and this implies that there is a bound on the length of w which is polynomial in the sizes of R_1 and R_2. Again, this is an open problem.

To obtain a problem in $\mathcal{NP}$ we have to consider a restricted type of regular expressions, and it turns out that $*$-free regular expressions are the right candidate. A *$*$-free regular expression* is a regular expression which is built up from the atomic expressions using only $+$ and $\cdot$, but not $*$. For example,

$$R = ((a + b)aa(a + b) + aba(a + b)b)$$

is such an expression.

It is easy to see that if R is a $*$-free regular expression, then for every string $w \in \mathcal{L}[R]$ we have $|w| \leq |R|$. In particular, $\mathcal{L}[R]$ is finite. The above observation shows that if R_1 and R_2 are $*$-free and if there is a string $w \in (\mathcal{L}[R_1] - \mathcal{L}[R_2]) \cup (\mathcal{L}[R_2] - \mathcal{L}[R_1])$, then $|w| \leq |R_1| + |R_2|$, so we can indeed check this in polynomial time. It follows that the inequivalence problem for $*$ -free regular expressions is in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce the **Satisfiability Problem** to it. This means that we provide a method running in polynomial time that converts every instance of **Satisfiability**

Problem to an instance of **Inequivalence of Regular Expressions** such that the first problem has a solution iff the converted problem has a solution.

Observe that both problems of **Inequivalence of Regular Expressions** and **Equivalence of Regular Expressions** are as hard as **Inequivalence of $*$-free Regular Expressions**, since if we could solve the first two problems in polynomial time, then we we could solve **Inequivalence of $*$-free Regular Expressions** in polynomial time, but since this problem is $\mathcal{NP}$-complete, we would have $\mathcal{P} = \mathcal{NP}$. This is very unlikely, so the complexity of **Equivalence of Regular Expressions** remains open.

(10) 0-1 **integer programming problem**

Let A be any $p \times q$ matrix with integer coefficients and let $b \in \mathbb{Z}^p$ be any vector with integer coefficients. The 0-1 **integer programming problem** is to find whether a system of p linear equations in q variables

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1q}x_q &= b_1 \\ \vdots \qquad\quad & \quad\vdots \\ a_{i1}x_1 + \cdots + a_{iq}x_q &= b_i \\ \vdots \qquad\quad & \quad\vdots \\ a_{p1}x_1 + \cdots + a_{pq}x_q &= b_p \end{aligned}$$

with $a_{ij}, b_i \in \mathbb{Z}$ has any solution $x \in \{0,1\}^q$, that is, with $x_i \in \{0,1\}$. In matrix form, if we let

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1q} \\ \vdots & \ddots & \vdots \\ a_{p1} & \cdots & a_{pq} \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ \vdots \\ b_p \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ \vdots \\ x_q \end{pmatrix},$$

then we write the above system as

$$Ax = b.$$

It is immediate that 0-1 **integer programming problem** is in $\mathcal{NP}$. To prove that it is $\mathcal{NP}$-complete we reduce the **bounded tiling** problem to it. This means that we provide a method running in polynomial time that converts every instance of the **bounded tiling** problem to an instance of the 0-1 **integer programming problem** such that the first problem has a solution iff the converted problem has a solution.

14.2 Proofs of $\mathcal{NP}$-Completeness

(1) **Exact Cover**

To prove that **Exact Cover** is $\mathcal{NP}$-complete, we reduce the **Satisfiability Problem** to it:

Satisfiability Problem $\leq_P$ **Exact Cover**

Given a set $F = \{C_1, \ldots, C_\ell\}$ of ℓ clauses constructed from n propositional variables $x_1, \ldots, x_n$, we must construct in polynomial time an instance $\tau(F) = (U, \mathcal{F})$ of **Exact Cover** such that F is satisfiable iff $\tau(F)$ has a solution.

Example 14.2. If

$$F = \{C_1 = (x_1 \vee \overline{x_2}),\ C_2 = (\overline{x_1} \vee x_2 \vee x_3),\ C_3 = (x_2),\ C_4 = (\overline{x_2} \vee \overline{x_3})\},$$

then the universe U is given by

$$U = \{x_1, x_2, x_3, C_1, C_2, C_3, C_4, p_{11}, p_{12}, p_{21}, p_{22}, p_{23}, p_{31}, p_{41}, p_{42}\},$$

and the family $\mathcal{F}$ consists of the subsets

$$\begin{aligned}
&\{p_{11}\}, \{p_{12}\}, \{p_{21}\}, \{p_{22}\}, \{p_{23}\}, \{p_{31}\}, \{p_{41}\}, \{p_{42}\} \\
&T_{1,\mathbf{F}} = \{x_1, p_{11}\} \\
&T_{1,\mathbf{T}} = \{x_1, p_{21}\} \\
&T_{2,\mathbf{F}} = \{x_2, p_{22}, p_{31}\} \\
&T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\} \\
&T_{3,\mathbf{F}} = \{x_3, p_{23}\} \\
&T_{3,\mathbf{T}} = \{x_3, p_{42}\} \\
&\{C_1, p_{11}\}, \{C_1, p_{12}\}, \{C_2, p_{21}\}, \{C_2, p_{22}\}, \{C_2, p_{23}\}, \\
&\{C_3, p_{31}\}, \{C_4, p_{41}\}, \{C_4, p_{42}\}.
\end{aligned}$$

It is easy to check that the set $\mathcal{C}$ consisting of the following subsets is an exact cover:

$$\begin{aligned}
&T_{1,\mathbf{T}} = \{x_1, p_{21}\}, T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\}, T_{3,\mathbf{F}} = \{x_3, p_{23}\}, \\
&\{C_1, p_{11}\}, \{C_2, p_{22}\}, \{C_3, p_{31}\}, \{C_4, p_{42}\}.
\end{aligned}$$

The general method to construct $(U, \mathcal{F})$ from $F = \{C_1, \ldots, C_\ell\}$ proceeds as follows. Say

$$C_j = (L_{j1} \vee \cdots \vee L_{jm_j})$$

is the jth clause in F, where L_{jk} denotes the kth literal in C_j and $m_j \geq 1$. The universe of $\tau(F)$ is the set

$$U = \{x_i \mid 1 \leq i \leq n\} \cup \{C_j \mid 1 \leq j \leq \ell\} \cup \{p_{jk} \mid 1 \leq j \leq \ell,\ 1 \leq k \leq m_j\}$$

where in the third set p_{jk} corresponds to the kth literal in C_j.

The following subsets are included in $\mathcal{F}$:

(a) There is a set $\{p_{jk}\}$ for every p_{jk}.

(b) For every boolean variable x_i, the following two sets are in $\mathcal{F}$:

$$T_{i,\mathbf{T}} = \{x_i\} \cup \{p_{jk} \mid L_{jk} = \overline{x_i}\}$$

which contains x_i and all negative occurrences of x_i, and

$$T_{i,\mathbf{F}} = \{x_i\} \cup \{p_{jk} \mid L_{jk} = x_i\}$$

which contains x_i and all its positive occurrences. Note carefully that $T_{i,\mathbf{T}}$ involves negative occurrences of x_i whereas $T_{i,\mathbf{F}}$ involves positive occurrences of x_i.

(c) For every clause C_j, the m_j sets $\{C_j, p_{jk}\}$ are in $\mathcal{F}$.

It remains to prove that F is satisfiable iff $\tau(F)$ has a solution. We claim that if v is a truth assignement that satisfies F, then we can make an exact cover $\mathcal{C}$ as follows:

For each x_i, we put the subset $T_{i,\mathbf{T}}$ in $\mathcal{C}$ iff $v(x_i) = \mathbf{T}$, else we we put the subset $T_{i,\mathbf{F}}$ in $\mathcal{C}$ iff $v(x_i) = \mathbf{F}$. Also, for every clause C_j, we put some subset $\{C_j, p_{jk}\}$ in $\mathcal{C}$ for a literal L_{jk} which is made true by v. By construction of $T_{i,\mathbf{T}}$ and $T_{i,\mathbf{F}}$, this p_{jk} is not in any set in $\mathcal{C}$ selected so far. Since by hypothesis F is satisfiable, such a literal exists for every clause. Having covered all x_i and C_j, we put a set $\{p_{jk}\}$ in $\mathcal{C}$ for every remaining p_{jk} which has not yet been covered by the sets already in $\mathcal{C}$.

Going back to Example 14.2, the truth assigment $v(x_1) = \mathbf{T}, v(x_2) = \mathbf{T}, v(x_3) = \mathbf{F}$ satisfies F, so we put

$$\begin{aligned} &T_{1,\mathbf{T}} = \{x_1, p_{21}\}, T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\}, T_{3,\mathbf{F}} = \{x_3, p_{23}\}, \\ &\{C_1, p_{11}\}, \{C_2, p_{22}\}, \{C_3, p_{31}\}, \{C_4, p_{42}\} \end{aligned}$$

in $\mathcal{C}$.

We leave as an exercise to check that the above procedure works.

Conversely, if $\mathcal{C}$ is an exact cover of $\tau(F)$, we define a truth assigment as follows:

For every x_i, if $T_{i,\mathbf{T}}$ is in $\mathcal{C}$, then we set $v(x_i) = \mathbf{T}$, else if $T_{i,\mathbf{F}}$ is in $\mathcal{C}$, then we set $v(x_i) = \mathbf{F}$. We leave it as an exercise to check that this procedure works.

Example 14.3. Given the exact cover

$$T_{1,\mathbf{T}} = \{x_1, p_{21}\}, T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\}, T_{3,\mathbf{F}} = \{x_3, p_{23}\},$$
$$\{C_1, p_{11}\}, \{C_2, p_{22}\}, \{C_3, p_{31}\}, \{C_4, p_{42}\},$$

we get the satisfying assigment $v(x_1) = \mathbf{T}, v(x_2) = \mathbf{T}, v(x_3) = \mathbf{F}$.

If we now consider the proposition is CNF given by

$$F_2 = \{C_1 = (x_1 \vee \overline{x_2}),\ C_2 = (\overline{x_1} \vee x_2 \vee x_3),\ C_3 = (x_2),\ C_4 = (\overline{x_2} \vee \overline{x_3} \vee x_4)\}$$

where we have added the boolean variable x_4 to clause C_4, then U also contains x_4 and p_{43} so we need to add the following subsets to $\mathcal{F}$:

$$T_{4,\mathbf{F}} = \{x_4, p_{43}\},\ T_{4,\mathbf{T}} = \{x_4\},\ \{C_4, p_{43}\},\ \{p_{43}\}.$$

The truth assigment $v(x_1) = \mathbf{T}, v(x_2) = \mathbf{T}, v(x_3) = \mathbf{F}, v(x_4) = \mathbf{T}$ satisfies F_2, so an exact cover $\mathcal{C}$ is

$$T_{1,\mathbf{T}} = \{x_1, p_{21}\}, T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\}, T_{3,\mathbf{F}} = \{x_3, p_{23}\}, T_{4,\mathbf{T}} = \{x_4\},$$
$$\{C_1, p_{11}\}, \{C_2, p_{22}\}, \{C_3, p_{31}\}, \{C_4, p_{42}\}, \{p_{43}\}.$$

Observe that this time, because the truth assignment v makes both literals corresponding to p_{42} and p_{43} true and since we picked p_{42} to form the subset $\{C_4, p_{42}\}$, we need to add the singleton $\{p_{43}\}$ to $\mathcal{C}$ to cover all elements of U.

(2) **Hamiltonian Cycle (for Directed Graphs)**

To prove that **Hamiltonian Cycle (for Directed Graphs)** is $\mathcal{NP}$-complete, we will reduce **Exact Cover** to it:

Exact Cover $\leq_P$ **Hamiltonian Cycle (for Directed Graphs)**

We need to find an algorithm working in polynomial time that converts an instance $(U, \mathcal{F})$ of **Exact Cover** to a directed graph $G = \tau(U, \mathcal{F})$ such that G has a Hamiltonian cycle iff $(U, \mathcal{F})$ has an exact cover.

The construction of the graph G uses a trick involving a small subgraph *Gad* with 7 (distinct) nodes known as a *gadget* shown in Figure 14.7.

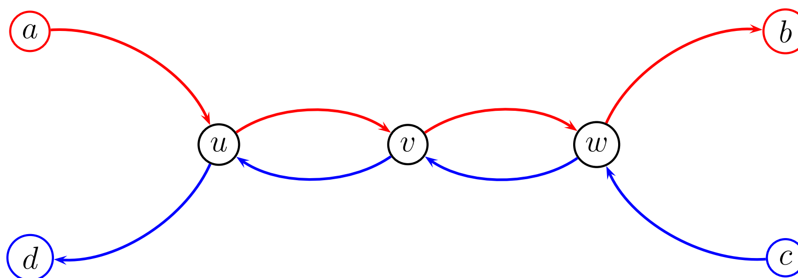


Figure 14.7: A gadget *Gad*

The crucial property of the graph *Gad* is that if *Gad* is a subgraph of a bigger graph G in such a way that no edge of G is incident to any of the nodes u, v, w unless it is one of the eight edges of *Gad* incident to the nodes u, v, w, then for any Hamiltonian cycle in G, either the path $(a, u), (u, v), (v, w), (w, b)$ is traversed or the path $(c, w), (w, v), (v, u), (u, d)$ is traversed, but not both.

The reader should convince herself/himself that indeed, any Hamiltonian cycle that does not traverse either the subpath $(a, u), (u, v), (v, w), (w, b)$ from a to b or the subpath $(c, w), (w, v)$, $(v, u), (u, d)$ from c to d will not traverse one of the nodes u, v, w. Also, the fact that node v is traversed exactly once forces only one of the two paths to be traversed but not both. The reader should also convince herself/himself that a smaller graph does not guarantee the desired property.

It is convenient to use the simplified notation with a special type of edge labeled with the exclusive or sign $\oplus$ between the "edges" between a and b and between d and c, as shown in Figure 14.8.

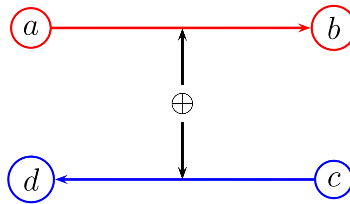


Figure 14.8: A shorthand notation for a gadget

Whenever such a figure occurs, the actual graph is obtained by substituting a copy of the graph *Gad* (the four nodes a, b, c, d must be distinct). This abbreviating device can be extended to the situation where we build gadgets between a given pair (a, b) and several other pairs $(c_1, d_1), \ldots, (c_m, d_m)$, all nodes beeing distinct, as illustrated in Figure 14.9.

Either all three edges $(c_1, d_1), (c_2, d_2), (c_3, d_3)$ are traversed or the edge (a, b) is traversed, and these possibilities are mutually exclusive.

The graph $G = \tau(U, \mathcal{F})$ where $U = \{u_1, \ldots, u_n\}$ (with $n \geq 1$) and $\mathcal{F} = \{S_1, \ldots, S_m\}$ (with $m \geq 1$) is constructed as follows:

The graph G has $m + n + 2$ nodes $\{u_0, u_1, \ldots, u_n, S_0, S_1, \ldots, S_m\}$. Note that we have added two extra nodes u_0 and S_0. For $i = 1, \ldots, m$, there are *two* edges $(S_{i-1}, S_i)_1$ and $(S_{i-1}, S_i)_2$ from S_{i-1} to S_i. For $j = 1, \ldots, n$, from u_{j-1} to u_j, there are as many edges as there are sets $S_i \in \mathcal{F}$ containing the element u_j. We can think of each edge between u_{j-1} and u_j as an occurrence of u_j in a uniquely determined set $S_i \in \mathcal{F}$; we

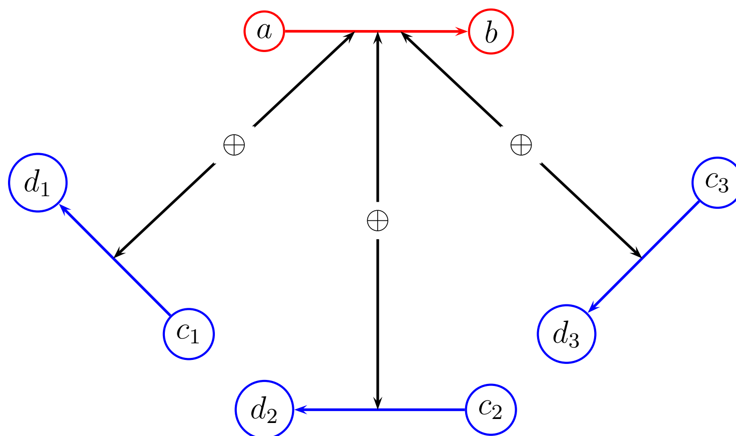


Figure 14.9: A shorthand notation for several gadgets

denote this edge by $(u_{j-1}, u_j)_i$. We also have an edge from u_n to S_0 and an edge from S_m to u_0, thus "closing the cycle."

What we have constructed so far is not a legal graph since it may have many parallel edges, but are going to turn it into a legal graph by pairing edges between the u_j's and edges between the S_i's. Indeed, since each edge $(u_{j-1}, u_j)_i$ between u_{j-1} and u_j corresponds to an occurrence of u_j in some uniquely determined set $S_i \in \mathcal{F}$ (that is, $u_j \in S_i$), we put an exclusive-or edge between the edge $(u_{j-1}, u_j)_i$ and the edge $(S_{i-1}, S_i)_2$ between S_{i-1} and S_i, which we call the *long edge*. The other edge $(S_{i-1}, S_i)_1$ between S_{i-1} and S_i (not paired with any other edge) is called the *short edge*. Effectively, we put a copy of the gadget graph *Gad* with $a = u_{j-1}, b = u_j, c = S_{i-1}, d = S_i$ for any pair (u_j, S_i) such that $u_j \in S_i$. The resulting object is indeed a directed graph with no parallel edges.

Example 14.4. The above construction is illustrated in Figure 14.10 for the instance of the exact cover problem given by

$$U = \{u_1, u_2, u_3, u_4\},\ \mathcal{F} = \{S_1 = \{u_3, u_4\},\ S_2 = \{u_2, u_3, u_4\},\ S_3 = \{u_1, u_2\}\}.$$

It remains to prove that $(U, \mathcal{F})$ has an exact cover iff the graph $G = \tau(U, \mathcal{F})$ has a Hamiltonian cycle. First, assume that G has a Hamiltonian cycle. If so, for every j some unique "edge" $(u_{j-1}, u_j)_i$ is traversed once (since every u_j is traversed once), and by the exclusive-or nature of the gadget graphs, the corresponding long edge $(S_{i-1}, S_i)_2$ can't be traversed, which means that the short edge $(S_{i-1}, S_i)_1$ is traversed. Consequently, if $\mathcal{C}$ consists of those subsets S_i such that the short edge $(S_{i-1}, S_i)_1$ is traversed, then $\mathcal{C}$ consists of pairwise disjoint subsets whose union is U, namely $\mathcal{C}$ is an exact cover.

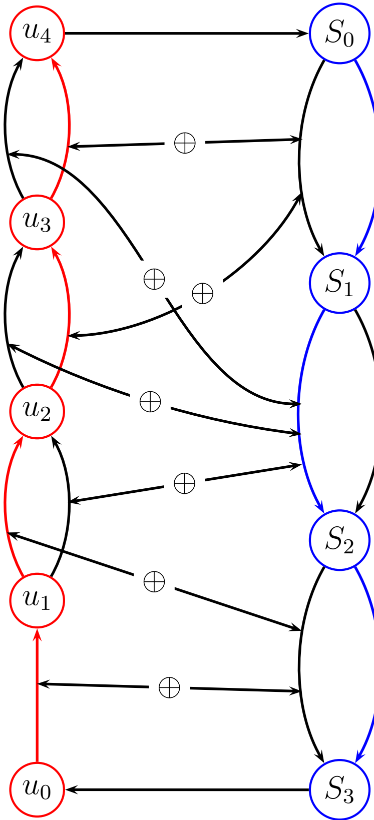


Figure 14.10: The directed graph constructed from the data $(U, \mathcal{F})$ of Example 14.4

In our example, there is a Hamiltonian where the blue edges are traversed between the S_i nodes, and the red edges are traversed between the u_j nodes, namely

$$\text{short } (S_0, S_1),\ \text{long } (S_1, S_2),\ \text{short } (S_2, S_3),\ (S_3, u_0),$$
$$(u_0, u_1)_3,\ (u_1, u_2)_3,\ (u_2, u_3)_1,\ (u_3, u_4)_1,\ (u_4, S_0).$$

The subsets corresponding to the short (S_{i-1}, S_i) edges are S_1 and S_3, and indeed $\mathcal{C} = \{S_1, S_3\}$ is an exact cover.

Note that the exclusive-or property of the gadgets implies the following: since the edge $(u_0, u_1)_3$ must be chosen to obtain a Hamiltonian, the long edge (S_2, S_3) can't be chosen, so the edge $(u_1, u_2)_3$ must be chosen, but then the edge $(u_1, u_2)_2$ is not chosen so the long edge (S_1, S_2) must be chosen, so the edges $(u_2, u_3)_2$ and $(u_3, u_4)_2$ can't be chosen, and thus edges $(u_2, u_3)_1$ and $(u_3, u_4)_1$ must be chosen.

Conversely, if $\mathcal{C}$ is an exact cover for $(U, \mathcal{F})$, then consider the path in G obtained by traversing each short edge $(S_{i-1}, S_i)_1$ for which $S_i \in \mathcal{C}$, each edge $(u_{j-1}, u_j)_i$ such that $u_j \in S_i$, which means that this edge is connected by a $\oplus$-sign to the long edge $(S_{i-1}, S_i)_2$

(by construction, for each u_j there is a unique such S_i), and the edges (u_n, S_0) and (S_m, u_0), then we obtain a Hamiltonian cycle.

In our example, the exact cover $\mathcal{C} = \{S_1, S_3\}$ yields the Hamiltonian

$$\text{short } (S_0, S_1),\ \text{long } (S_1, S_2),\ \text{short } (S_2, S_3),\ (S_3, u_0),$$
$$(u_0, u_1)_3,\ (u_1, u_2)_3,\ (u_2, u_3)_1,\ (u_3, u_4)_1,\ (u_4, S_0)$$

that we encountered earlier.

(3) **Hamiltonian Cycle (for Undirected Graphs)**

To show that **Hamiltonian Cycle (for Undirected Graphs)** is $\mathcal{NP}$-complete we reduce **Hamiltonian Cycle (for Directed Graphs)** to it:

Hamiltonian Cycle (for Directed Graphs) $\leq_P$ **Hamiltonian Cycle (for Undirected Graphs)**

Given any directed graph $G = (V, E)$ we need to construct in polynomial time an undirected graph $\tau(G) = G' = (V', E')$ such that G has a (directed) Hamiltonian cycle iff G' has a (undirected) Hamiltonian cycle. This is easy. We make three distinct copies v_0, v_1, v_2 of every node $v \in V$ which we put in V', and for every edge $(u, v) \in E$ we create five edges $\{u_0, u_1\}, \{u_1, u_2\}, \{u_2, v_0\}, \{v_0, v_1\}, \{v_1, v_2\}$ which we put in E', as illustrated in the diagram shown in Figure 14.11.

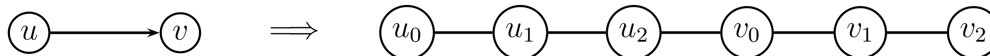


Figure 14.11: Conversion of a directed graph into an undirected graph

The crucial point about the graph G' is that although there may be several edges adjacent to a node u_0 or a node u_2, the only way to reach u_1 from u_0 is through the edge $\{u_0, u_1\}$ and the only way to reach u_1 from u_2 is through the edge $\{u_1, u_2\}$.

Suppose there is a Hamiltonian cycle in G'. If this cycle arrives at a node u_0 from the node u_1, then by the above remark, the previous node in the cycle must be u_2. Then, the predecessor of u_2 in the cycle must be a node v_0 such that there is an edge $\{u_2, v_0\}$ in G' arising from an edge (u, v) in G. The nodes in the cycle in G' are traversed in the order (v_0, u_2, u_1, u_0) where v_0 and u_2 are traversed in the opposite order in which they occur as the endpoints of the edge (u, v) in G. If so, consider the reverse of our Hamiltonian cycle in G', which is also a Hamiltonian cycle since G' is unoriented. In this cycle, we go from u_0 to u_1, then to u_2, and finally to v_0. In G, we traverse the edge from u to v. In order for the cycle in G' to be Hamiltonian, we must continue by visiting v_1 and v_2, since otherwise v_1 is never traversed. Now, the next node w_0 in the Hamiltonian cycle in G' corresponds to an edge (v, w) in G, and by repeating our

reasoning we see that our Hamiltonian cycle in G' determines a Hamiltonian cycle in G. We leave it as an easy exercise to check that a Hamiltonian cycle in G yields a Hamiltonian cycle in G'.

(4) **Traveling Salesman Problem**

To show that the **Traveling Salesman Problem** is $\mathcal{NP}$-complete, we reduce the **Hamiltonian Cycle Problem (Undirected Graphs)** to it:

Hamiltonian Cycle Problem (Undirected Graphs) $\leq_P$ **Traveling Salesman Problem**

This is a fairly easy reduction.

Given an undirected graph $G = (V, E)$, we construct an instance $\tau(G) = (D, B)$ of the traveling salesman problem so that G has a Hamiltonian cycle iff the traveling salesman problem has a solution. If we let $n = |V|$, we have n cities and the matrix $D = (d_{ij})$ is defined as follows:

$$d_{ij} = \begin{cases} 0 & \text{if } i = j \\ 1 & \text{if } \{v_i, v_j\} \in E \\ 2 & \text{otherwise.} \end{cases}$$

We also set the budget B as $B = n$.

Any tour of the cities has cost equal to n plus the number of pairs (v_i, v_j) such that $i \neq j$ and $\{v_i, v_j\}$ is *not* an edge of G. It follows that a tour of cost n exists iff there are no pairs (v_i, v_j) of the second kind iff the tour is a Hamiltonian cycle.

The reduction from **Hamiltonian Cycle Problem (Undirected Graphs)** to the **Traveling Salesman Problem** is quite simple, but a direct reduction of say **Satisfiability** to the **Traveling Salesman Problem** is hard. By breaking this reduction into several steps made it simpler to achieve.

(5) **Independent Set**

To show that **Independent Set** is $\mathcal{NP}$-complete, we reduce **Exact 3-Satisfiability** to it:

Exact 3-Satisfiability $\leq_P$ **Independent Set**

Recall that in **Exact 3-Satisfiability** every clause C_i has exactly three literals L_{i1}, L_{i2}, L_{i3}.

Given a set $F = \{C_1, \ldots, C_m\}$ of $m \geq 2$ such clauses, we construct in polynomial time an undirected graph $G = (V, E)$ such that F is satisfiable iff G has an independent set C with at least $K = m$ nodes.

For every i $(1 \leq i \leq m)$, we have three nodes c_{i1}, c_{i2}, c_{i3} corresponding to the three literals L_{i1}, L_{i2}, L_{i3} in clause C_i, so there are $3m$ nodes in V. The "core" of G consists of m triangles, one for each set $\{c_{i1}, c_{i2}, c_{i3}\}$. We also have an edge $\{c_{ik}, c_{j\ell}\}$ iff L_{ik} and $L_{j\ell}$ are complementary literals.

Example 14.5. Let F be the set of clauses

$$F = \{C_1 = (x_1 \vee \overline{x_2} \vee x_3),\ C_2 = (\overline{x_1} \vee \overline{x_2} \vee x_3),\ C_3 = (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}),\ C_4 = (x_1 \vee x_2 \vee x_3)\}.$$

The graph G associated with F is shown in Figure 14.12.

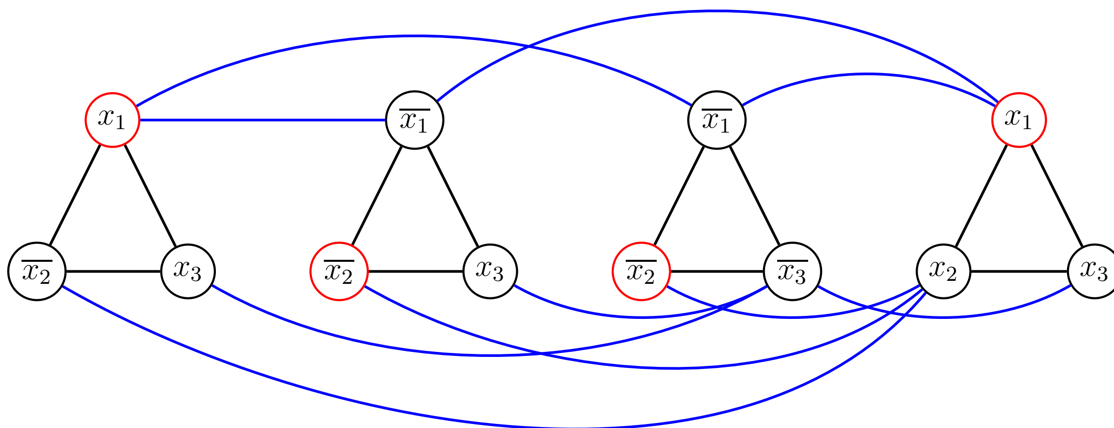


Figure 14.12: The graph constructed from the clauses of Example 14.5

It remains to show that the construction works. Since any three nodes in a triangle are connected, an independent set C can have at most one node per triangle and thus has at most m nodes. Since the budget is $K = m$, we may assume that there is an independent set with m nodes. Define a (partial) truth assignment by

$$v(x_i) = \begin{cases} \mathbf{T} & \text{if } L_{jk} = x_i \text{ and } c_{jk} \in C \\ \mathbf{F} & \text{if } L_{jk} = \overline{x_i} \text{ and } c_{jk} \in C. \end{cases}$$

Since the non-triangle edges in G link nodes corresponding to complementary literals and nodes in C are not connected, our truth assigment does not assign clashing truth values to the variables x_i. Not all variables may receive a truth value, in which case we assign an arbitrary truth value to the unassigned variables. This yields a satisfying assignment for F.

In Example 14.5, the set $C = \{c_{11}, c_{22}, c_{32}, c_{41}\}$ corresponding to the nodes shown in red in Figure 14.12 form an independent set, and they induce the partial truth assignment $v(x_1) = \mathbf{T}, v(x_2) = \mathbf{F}$. The variable x_3 can be assigned an arbitrary value, say $v(x_3) = \mathbf{F}$, and v is indeed a satisfying truth assignment for F.

Conversely, if v is a truth assignment for F, then we obtain an independent set C of size m by picking for each clause C_i a node c_{ik} corresponding to a literal L_{ik} whose value under v is $\mathbf{T}$.

(6) **Clique**

To show that **Clique** is $\mathcal{NP}$-complete, we reduce **Independent Set** to it:

Independent Set $\leq_P$ **Clique**

The key the reduction is the notion of the complement of an undirected graph $G = (V, E)$. The *complement* $G^c = (V, E^c)$ of the graph $G = (V, E)$ is the graph with the same set of nodes V as G but there is an edge $\{u, v\}$ (with $u \neq v$) in E^c iff $\{u, v\} \notin E$. Then, it is not hard to check that there is a bijection between maximum independent sets in G and maximum cliques in G^c. The reduction consists in constructing from a graph G its complement G^c, and then G has an independent set iff G^c has a clique.

This construction is illustrated in Figure 14.13, where a maximum independent set in the graph G is shown in blue and a maximum clique in the graph G^c is shown in red.

Figure 14.13: A graph (left) and its complement (right)

(7) **Node Cover**

To show that **Node Cover** is $\mathcal{NP}$-complete, we reduce **Independent Set** to it:

Independent Set $\leq_P$ **Node Cover**

This time the crucial observation is that if N is an independent set in G, then the complement $C = V - N$ of N in V is a node cover in G. Thus there is an independent set of size at least K iff there is a node cover of size at most $n - K$ where $n = |V|$ is the number of nodes in V. The reduction leaves the graph unchanged and replaces K by $n - K$. An example is shown in Figure 14.14 where an independent set is shown in blue and a node cover is shown in red.

Figure 14.14: An inpendent set (left) and a node cover (right)

(8) **Knapsack (also called Subset sum)**

To show that **Knapsack** is $\mathcal{NP}$-complete, we reduce **Exact Cover** to it:

Exact Cover $\leq_P$ Knapsack

Given an instance $(U, \mathcal{F})$ of set cover with $U = \{u_1, \ldots, u_n\}$ and $\mathcal{F} = \{S_1, \ldots, S_m\}$, a family of subsets of U, we need to produce in polynomial time an instance $\tau(U, \mathcal{F})$ of the knapsack problem consisting of k nonnegative integers $a_1, \ldots, a_k$ and another integer $K > 0$ such that there is a subset $I \subseteq \{1, \ldots, k\}$ such that $\sum_{i \in I} a_i = K$ iff there is an exact cover of U using subsets in $\mathcal{F}$.

The trick here is the relationship between *set union* and *integer addition*.

Example 14.6. Consider the exact cover problem given by $U = \{u_1, u_2, u_3, u_4\}$ and

$$\mathcal{F} = \{S_1 = \{u_3, u_4\},\ S_2 = \{u_2, u_3, u_4\},\ S_3 = \{u_1, u_2\}\}.$$

We can represent each subset S_j by a binary string a_j of length 4, where the ith bit from the left is 1 iff $u_i \in S_j$, and 0 otherwise. In our example

$$\begin{aligned} a_1 &= 0011 \\ a_2 &= 0111 \\ a_3 &= 1100. \end{aligned}$$

Then, the trick is that some family $\mathcal{C}$ of subsets S_j is an exact cover if the sum of the corresponding numbers a_j adds up to $1111 = 2^4 - 1 = K$. For example,

$$\mathcal{C} = \{S_1 = \{u_3, u_4\},\ S_3 = \{u_1, u_2\}\}$$

is an exact cover and

$$a_1 + a_3 = 0011 + 1100 = 1111.$$

Unfortunately, there is a problem with this encoding which has to do with the fact that addition may involve carry. For example, assuming four subsets and the universe $U = \{u_1, \ldots, u_6\}$,

$$11 + 13 + 15 + 24 = 63,$$

in binary

$$001011 + 001101 + 001111 + 011000 = 111111,$$

but if we convert these binary strings to the corresponding subsets we get the subsets

$$\begin{aligned} S_1 &= \{u_3, u_5, u_6\} \\ S_2 &= \{u_3, u_4, u_6\} \\ S_3 &= \{u_3, u_4, u_5, u_6\} \\ S_4 &= \{u_2, u_3\}, \end{aligned}$$

which are not disjoint and do not cover U.

The fix is surprisingly simple: use base m (where m is the number of subsets in $\mathcal{F}$) instead of base 2.

Example 14.7. Consider the exact cover problem given by $U = \{u_1, u_2, u_3, u_4, u_5, u_6\}$ and $\mathcal{F}$ given by

$$\begin{aligned} S_1 &= \{u_3, u_5, u_6\} \\ S_2 &= \{u_3, u_4, u_6\} \\ S_3 &= \{u_3, u_4, u_5, u_6\} \\ S_4 &= \{u_2, u_3\}, \\ S_5 &= \{u_1, u_2, u_4\}. \end{aligned}$$

In base $m = 5$, the numbers corresponding to $S_1, \ldots, S_5$ are

$$\begin{aligned} a_1 &= 001011 \\ a_2 &= 001101 \\ a_3 &= 001111 \\ a_4 &= 011000 \\ a_5 &= 110100. \end{aligned}$$

This time,

$$a_1 + a_2 + a_3 + a_4 = 001011 + 001101 + 001111 + 011000 = 014223 \neq 111111,$$

so $\{S_1, S_2, S_3, S_4\}$ is not a solution. However

$$a_1 + a_5 = 001011 + 110100 = 111111,$$

and $\mathcal{C} = \{S_1, S_5\}$ is an exact cover.

Thus, given an instance $(U, \mathcal{F})$ of **Exact Cover** where $U = \{u_1, \ldots, u_n\}$ and $\mathcal{F} = \{S_1, \ldots, S_m\}$ the reduction to **Knapsack** consists in forming the m numbers $a_1, \ldots, a_m$ (each of n bits) encoding the subsets S_j, namely $a_{ji} = 1$ iff $u_i \in S_j$, else 0, and to let $K = 1 + m^2 + \cdots + m^{n-1}$, which is represented in base m by the string $\underbrace{11 \cdots 11}_{n}$. In testing whether $\sum_{i \in I} a_i = K$ for some subset $I \subseteq \{1, \ldots, m\}$, we use arithmetic in base m.

If a candidate solution $\mathcal{C}$ involves at most $m - 1$ subsets, then since the corresponding numbers are added in base m, a carry can never happen. If the candidate solution involves all m subsets, then $a_1 + \cdots + a_m = K$ iff $\mathcal{F}$ is a partition of U, since otherwise some bit in the result of adding up these m numbers in base m is not equal to 1, even if a carry occurs.

(9) **Inequivalence of $*$-free Regular Expressions**

To show that **Inequivalence of $*$-free Regular Expressions** is $\mathcal{NP}$-complete, we reduce the **Satisfiability Problem** to it:

Satisfiability Problem $\leq_P$ Inequivalence of $*$-free Regular Expressions

We already argued that **Inequivalence of $*$-free Regular Expressions** is in $\mathcal{NP}$ because if R is a $*$-free regular expression, then for every string $w \in \mathcal{L}[R]$ we have $|w| \leq |R|$. The above observation shows that if R_1 and R_2 are $*$-free and if there is a string $w \in (\mathcal{L}[R_1] - \mathcal{L}[R_2]) \cup (\mathcal{L}[R_2] - \mathcal{L}[R_1])$, then $|w| \leq |R_1| + |R_2|$, so we can indeed check this in polynomial time. It follows that the inequivalence problem for $*$ -free regular expressions is in $\mathcal{NP}$.

We reduce the **Satisfiability Problem** to the **Inequivalence of $*$-free Regular Expressions** as follows. For any set of clauses $P = C_1 \wedge \cdots \wedge C_p$, if the propositional variables occurring in P are $x_1, \ldots, x_n$, we produce two $*$-free regular expressions R, S over $\Sigma = \{0, 1\}$, such that P is satisfiable iff $L_R \neq L_S$. The expression S is actually

$$S = \underbrace{(0 + 1)(0 + 1) \cdots (0 + 1)}_{n}.$$

The expression R is of the form

$$R = R_1 + \cdots + R_p,$$

where R_i is constructed from the clause C_i in such a way that L_{R_i} corresponds precisely to the set of truth assignments that falsify C_i; see below.

Given any clause C_i, let R_i be the $*$-free regular expression defined such that, if x_j and $\overline{x}_j$ both belong to C_i (for some j), then $R_i = \emptyset$, else

$$R_i = R_i^1 \cdot R_i^2 \cdots R_i^n,$$

where R_i^j is defined by

$$R_i^j = \begin{cases} 0 & \text{if } x_j \text{ is a literal of } C_i \\ 1 & \text{if } \overline{x}_j \text{ is a literal of } C_i \\ (0+1) & \text{if } x_j \text{ does not occur in } C_i. \end{cases}$$

Clearly, all truth assignments that falsify C_i must assign **F** to x_j if $x_j \in C_i$ or assign **T** to x_j if $\overline{x}_j \in C_i$. Therefore, L_{R_i} corresponds to the set of truth assignments that falsify C_i (where 1 stands for **T** and 0 stands for **F**) and thus, if we let

$$R = R_1 + \cdots + R_p,$$

then L_R corresponds to the set of truth assignments that falsify $P = C_1 \wedge \cdots \wedge C_p$. Since $L_S = \{0,1\}^n$ (all binary strings of length n), we conclude that $L_R \neq L_S$ iff P is satisfiable. Therefore, we have reduced the **Satisfiability Problem** to our problem and the reduction clearly runs in polynomial time. This proves that the problem of deciding whether $L_R \neq L_S$, for any two $*$-free regular expressions R and S is $\mathcal{NP}$-complete.

(10) 0-1 **integer programming problem**

It is easy to check that the problem is in $\mathcal{NP}$.

To prove that the is $\mathcal{NP}$-complete we reduce the **bounded-tiling problem** to it:

bounded-tiling problem $\leq_P$ 0-1 **integer programming problem**

Given a tiling problem, $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$, we create a 0-1-valued variable x_{mnt}, such that $x_{mnt} = 1$ iff tile t occurs in position (m,n) in some tiling. Write equations or inequalities expressing that a tiling exists and then use "slack variables" to convert inequalities to equations. For example, to express the fact that every position is tiled by a single tile, use the equation

$$\sum_{t \in \mathcal{T}} x_{mnt} = 1,$$

for all m, n with $1 \leq m \leq 2s$ and $1 \leq n \leq s$. We leave the rest as as exercise.

14.3 Succinct Certificates, co$\mathcal{NP}$, and $\mathcal{EXP}$

All the problems considered in Section 14.1 share a common feature, which is that for each problem, a solution is produced nondeterministically (an exact cover, a directed Hamiltonian cycle, a tour of cities, an independent set, a node cover, a clique *etc.*), and then this candidate solution is checked deterministically and in polynomial time. The candidate solution is a string called a *certificate* (or *witness*).

It turns out that membership on $\mathcal{NP}$ can be defined in terms of certificates. To be a certificate, a string must satisfy two conditions:

1. It must be *polynomially succinct*, which means that its length is at most a polynomial in the length of the input.

2. It must be *checkable* in polynomial time.

All "yes" inputs to a problem in $\mathcal{NP}$ must have at least one certificate, while all "no" inputs must have none.

The notion of certificate can be formalized using the notion of a polynomially balanced language.

Definition 14.3. Let Σ be an alphabet, and let ";" be a symbol not in Σ. A language $L' \subseteq \Sigma^*; \Sigma^*$ is said to be *polynomially balanced* if there exists a polynomial $p(X)$ such that for all $x, y \in \Sigma^*$, if $x; y \in L'$ then $|y| \leq p(|x|)$.

Suppose L' is a polynomially balanced language and that $L' \in \mathcal{P}$. Then we can consider the language

$$L = \{x \in \Sigma^* \mid (\exists y \in \Sigma^*)(x; y \in L')\}.$$

The intuition is that for each $x \in L$, the set

$$\{y \in \Sigma^* \mid x; y \in L'\}$$

is the set of certificates of x. For every $x \in L$, a Turing machine can nondeterministically guess one of its certificates y, and then use the deterministic Turing machine for L' to check in polynomial time that $x; y \in L'$. Note that, by definition, strings not in L have no certificate. It follows that $L \in \mathcal{NP}$.

Conversely, if $L \in \mathcal{NP}$ and the alphabet Σ has at least two symbols, we can encode the paths in the computation tree for every input $x \in L$, and we obtain a polynomially balanced language $L' \subseteq \Sigma^*; \Sigma^*$ in $\mathcal{P}$ such that

$$L = \{x \in \Sigma^* \mid (\exists y \in \Sigma^*)(x; y \in L')\}.$$

The details of this construction are left as an exercise. In summary, we obtain the following theorem.

Theorem 14.1. *Let $L \subseteq \Sigma^*$ be a language over an alphabet Σ with at least two symbols, and let ";" be a symbol not in Σ. Then $L \in \mathcal{NP}$ iff there is a polynomially balanced language $L' \subseteq \Sigma^*; \Sigma^*$ such that $L' \in \mathcal{P}$ and*

$$L = \{x \in \Sigma^* \mid (\exists y \in \Sigma^*)(x; y \in L')\}.$$

A striking illustration of the notion of succint certificate is illustrated by the set of *composite* integers, namely those natural numbers $n \in \mathbb{N}$ that can be written as the product pq of two numbers $p, q \geq 2$ with $p, q \in \mathbb{N}$. For example, the number

$$4,294,967,297$$

is a composite!

This is far from obvious, but if an oracle gives us the certificate $\{6,700,417,\ 641\}$, it is easy to carry out in polynomial time the multiplication of these two numbers and check that it is equal to $4,294,967,297$. Finding a certificate is usually (very) hard, but checking that it works is easy. This is the point of certificates.

We conclude this section with a brief discussion of the complexity classes co$\mathcal{NP}$ and $\mathcal{EXP}$.

By definition,

$$\text{co}\mathcal{NP} = \{\overline{L} \mid L \in \mathcal{NP}\},$$

that is, co$\mathcal{NP}$ consists of all complements of languages in $\mathcal{NP}$. Since $\mathcal{P} \subseteq \mathcal{NP}$ and $\mathcal{P}$ is closed under complementation,

$$\mathcal{P} \subseteq \text{co}\mathcal{NP},$$

but nobody knows whether $\mathcal{NP}$ is closed under complementation, that is, nobody knows whether $\mathcal{NP} = \text{co}\mathcal{NP}$.

What can be shown is that if $\mathcal{NP} \neq \text{co}\mathcal{NP}$ then $\mathcal{P} \neq \mathcal{NP}$. However it is possible that $\mathcal{P} \neq \mathcal{NP}$ and yet $\mathcal{NP} = \text{co}\mathcal{NP}$, although this is considered unlikely.

Of course, $\mathcal{P} \subseteq \mathcal{NP} \cap \text{co}\mathcal{NP}$. There are problems in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ not known to be in $\mathcal{P}$. One of the most famous in the following problem:

Integer factorization problem:

Given an integer $N \geq 3$, and another integer M (a budget) such that $1 < M < N$, does N have a factor d with $1 < d \leq M$?

That **Integer factorization** is in $\mathcal{NP}$ is clear. To show that **Integer factorization** is in co$\mathcal{NP}$, we can guess a factorization of N into distinct factors all greater than M, check that they are prime using the results of Chapter 15 showing that testing primality is in $\mathcal{NP}$ (even in $\mathcal{P}$, but that's much harder to prove), and then check that the product of these factors is N.

It is widely believed that **Integer factorization** does not belong to $\mathcal{P}$, which is the technical justification for saying that this problem is hard. Most cryptographic algorithms rely on this unproven fact. If **Integer factorization** was either $\mathcal{NP}$-complete or co$\mathcal{NP}$-complete, then we would have $\mathcal{NP} = \text{co}\mathcal{NP}$, which is considered very unlikely.

Remark: If $\sqrt{N} \leq M < N$, the above problem is equivalent to asking whether N is prime.

A natural instance of a problem in co$\mathcal{NP}$ is the *unsatisfiability problem* for propositions UNSAT $= \neg$SAT, namely deciding that a proposition P has no satisfying assignmnent.

A proposition P (in CNF) is *falsifiable* if there is some truth assigment v such that $\widehat{v}(P) = \mathbf{F}$. It is obvious that the set of falsifiable propositions is in $\mathcal{NP}$. Since a proposition

P is valid iff P is not falsifiable, the *validity (or tautology) problem* TAUT for propositions is in co$\mathcal{NP}$. In fact, TAUT is co$\mathcal{NP}$-complete; see Papadimitriou [14].

This is easy to prove. Since SAT is $\mathcal{NP}$-complete, for every language $L \in \mathcal{NP}$, there is a polynomial-time computable function $f \colon \Sigma^* \to \Sigma^*$ such that $x \in L$ iff $f(x) \in$ SAT. Then $x \notin L$ iff $f(x) \notin$ SAT, that is, $x \in \overline{L}$ iff $f(x) \in \neg$SAT, which means that every language $\overline{L} \in$ co$\mathcal{NP}$ is polynomial-time reducible to $\neg$SAT = UNSAT. But TAUT = $\{\neg P \mid P \in \text{UNSAT}\}$, so we have the polynomial-time computable function g given by $g(x) = \neg f(x)$ which gives us the reduction $x \in \overline{L}$ iff $g(x) \in$ TAUT, which shows that TAUT is co$\mathcal{NP}$-complete.

Despite the fact that this problem has been extensively studied, not much is known about its exact complexity.

The reasoning used to show that TAUT is co$\mathcal{NP}$-complete can also be used to show the following interesting result.

Proposition 14.2. *If a language L is $\mathcal{NP}$-complete, then its complement $\overline{L}$ is* co$\mathcal{NP}$*-complete.*

Proof. By definition, since $L \in \mathcal{NP}$, we have $\overline{L} \in$ co$\mathcal{NP}$. Since L is $\mathcal{NP}$-complete, for every language $L_2 \in \mathcal{NP}$, there is a polynomial-time computable function $f \colon \Sigma^* \to \Sigma^*$ such that $x \in L_2$ iff $f(x) \in L$. Then $x \notin L_2$ iff $f(x) \notin L$, that is, $x \in \overline{L_2}$ iff $f(x) \in \overline{L}$, which means that $\overline{L}$ is co$\mathcal{NP}$-hard as well, thus co$\mathcal{NP}$-complete. ☐

The class $\mathcal{EXP}$ is defined as follows.

Definition 14.4. A deterministic Turing machine M is said to be *exponentially bounded* if there is a polynomial $p(X)$ such that for every input $x \in \Sigma^*$, there is no ID ID_n such that

$$ID_0 \vdash ID_1 \vdash^* ID_{n-1} \vdash ID_n, \quad \text{with} \quad n > 2^{p(|x|)}.$$

The class $\mathcal{EXP}$ is the class of all languages that are accepted by some exponentially bounded deterministic Turing machine.

Remark: We can also define the class $\mathcal{NEXP}$ as in Definition 14.4, except that we allow nondeterministic Turing machines.

One of the interesting features of $\mathcal{EXP}$ is that it contains $\mathcal{NP}$.

Theorem 14.3. *We have the inclusion $\mathcal{NP} \subseteq \mathcal{EXP}$.*

Sketch of proof. Let M be some nondeterministic Turing machine accepting L in polynomial time bounded by $p(X)$. We can construct a deterministic Turing machine M' that operates as follows: for every input x, M' simulates M on all computations of length 1, then on all possible computations of length 2, and so on, up to all possible computations of length $p(|x|)+1$. At this point, either an accepting computation has been discovered or all computations have halted rejecting. We claim that M' operates in time bounded by $2^{q(|x|)}$ for some

poynomial $q(X)$. First, let r be the degree of nondeterminism of M, that is, the maximum number of triples (b, m, q) such that a quintuple (p, q, b, m, q) is an instructions of M. Then to simulate a computation of M of length ℓ, M' needs $\mathcal{O}(\ell)$ steps—to copy the input, to produce a string c in $\{1, \ldots, r\}^\ell$, and so simulate M according to the choices specified by c. It follows that M' can carry out the simulation of M on an input x in

$$\sum_{\ell=1}^{p(|x|)+1} r^\ell \leq (r+1)^{p(|x|)+1}$$

steps. Including the $\mathcal{O}(\ell)$ extra steps for each ℓ, we obtain the bound $(r+2)^{p(|x|)+1}$. Then, we can pick a constant k such that $2^k > r+2$, and with $q(X) = k(p(X)+1)$, we see that M' operates in time bounded by $2^{q(|x|)}$. ☐

It is also immediate to see that $\mathcal{EXP}$ is closed under complementation. Furthermore the strict inclusion $\mathcal{P} \subset \mathcal{EXP}$ holds.

Theorem 14.4. *We have the strict inclusion $\mathcal{P} \subset \mathcal{EXP}$.*

Sketch of proof. We use a diagonalization argument to produce a language E such that $E \notin \mathcal{P}$, yet $E \in \mathcal{EXP}$. We need to code a Turing machine as a string, but this can certainly be done using the techniques of Chapter 9. Let $\#(M)$ be the code of Turing machine M. Define E as

$$E = \{\#(M)x \mid M \text{ accepts input } x \text{ after at most } 2^{|x|} \text{ steps}\}.$$

We claim that $E \notin \mathcal{P}$. We proceed by contradiction. If $E \in \mathcal{P}$, then so is the language E_1 given by

$$E_1 = \{\#(M) \mid M \text{ accepts } \#(M) \text{ after at most } 2^{|\#(M)|} \text{ steps}\}.$$

Since $\mathcal{P}$ is closed under complementation, we also have $\overline{E_1} \in \mathcal{P}$. Let M^* be a deterministic Turing machine accepting $\overline{E_1}$ in time $p(X)$, for some polynomial $p(X)$. Since $p(X)$ is a polynomial, there is some n_0 such that $p(n) \leq 2^n$ for all all $n \geq n_0$. We may also assume that $|\#(M^*)| \geq n_0$, since if not we can add n_0 "dead states" to M^*.

Now, what happens if we run M^* on its own code $\#(M^*)$?

It is easy to see that we get a contradiction, namely M^* accepts $\#(M^*)$ iff M^* rejects $\#(M^*)$. We leave this verification as an exercise.

In conclusion, $\overline{E_1} \notin \mathcal{P}$, which in turn implies that $E \notin \mathcal{P}$.

It remains to prove that $E \in \mathcal{EXP}$. This is because we can construct a Turing machine that can in exponential time simulate any Turing machine M on input x for $2^{|x|}$ steps. ☐

In summary, we have the chain of inclusions

$$\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{EXP},$$

where the left inclusion and the right inclusion are both open problems, but we know that at least one of these two inclusions is strict.

We also have the inclusions

$$\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{EXP} \subseteq \mathcal{NEXP}.$$

Nobody knows whether $\mathcal{EXP} = \mathcal{NEXP}$, but it can be shown that if $\mathcal{EXP} \neq \mathcal{NEXP}$, then $\mathcal{P} \neq \mathcal{NP}$; see Papadimitriou [14].

Chapter 15

Primality Testing is in $\mathcal{NP}$

15.1 Prime Numbers and Composite Numbers

Prime numbers have fascinated mathematicians and more generally curious minds for thousands of years. What is a prime number? Well, $2, 3, 5, 7, 11, 13, \ldots, 9973$ are prime numbers.

Definition 15.1. A positive integer p is *prime* if $p \geq 2$ and if p is only divisible by 1 and p. Equivalently, p is prime if and only if p is a positive integer $p \geq 2$ that is not divisible by any integer m such that $2 \leq m < p$. A positive integer $n \geq 2$ which is not prime is called *composite*.

Observe that the number 1 is considered neither a prime nor a composite. For example, $6 = 2 \cdot 3$ is composite. Is $3\,215\,031\,751$ composite? Yes, because

$$3\,215\,031\,751 = 151 \cdot 751 \cdot 28351.$$

Even though the definition of primality is very simple, the structure of the set of prime numbers is highly nontrivial. The prime numbers are the basic building blocks of the natural numbers because of the following theorem bearing the impressive name of *fundamental theorem of arithmetic*.

Theorem 15.1. *Every natural number $n \geq 2$ has a unique factorization*

$$n = p_1^{i_1} p_2^{i_2} \cdots p_k^{i_k},$$

where the exponents $i_1, \ldots, i_k$ are positive integers and $p_1 < p_2 < \cdots < p_k$ are primes.

Every book on number theory has a proof of Theorem 15.1. The proof is not difficult and uses induction. It has two parts. The first part shows the existence of a factorization. The second part shows its uniqueness. For example, see Apostol [1] (Chapter 1, Theorem 1.10).

How many prime numbers are there? Many! In fact, infinitely many.

Theorem 15.2. *The set of prime numbers is infinite.*

Proof. The following proof attributed to Hermite only use the fact that every integer greater than 1 has some prime divisor. We prove that for every natural number $n \geq 2$, there is some prime $p > n$. Consider $N = n! + 1$. The number N must be divisible by some prime p ($p = N$ is possible). Any prime p dividing N is distinct from $2, 3, \ldots, n$, since otherwise p would divide $N - n! = 1$, a contradiction. $\square$

The problem of determining whether a given integer is prime is one of the better known and most easily understood problems of pure mathematics. This problem has caught the interest of mathematicians again and again for centuries. However, it was not until the 20th century that questions about primality testing and factoring were recognized as problems of practical importance, and a central part of applied mathematics. The advent of cryptographic systems that use large primes, such as RSA, was the main driving force for the development of fast and reliable methods for primality testing. Indeed, in order to create RSA keys, one needs to produce large prime numbers.

15.2 Methods for Primality Testing

The general strategy to test whether an integer $n > 2$ is prime or composite is to choose some property, say A, implied by primality, and to search for a counterexample a to this property for the number n, namely some a for which property A fails. We look for properties for which checking that a candidate a is indeed a countexample can be done quickly.

Is simple property that is the basis of several primality testing algorithms is the *Fermat test*, namely

$$a^{n-1} \equiv 1 \pmod{n},$$

which means that $a^{n-1} - 1$ is divisible by n (see Definition 15.2 for the meaning of the notation $a \equiv b \pmod{n}$). If n is prime, and if $\gcd(a, n) = 1$, then the above test is indeed satisfied; this is Fermat's little theorem, Theorem 15.7.

Typically, together with the number n being tested for primality, some candidate counterexample a is supplied to an algorithm which runs a test to determine whether a is really a counterexample to property A for n. If the test says that a is a counterexample, also called a *witness*, then we know for sure that n is composite.

For example, using the Fermat test, if $n = 10$ and $a = 3$, we check that

$$3^9 = 19683 = 10 \cdot 1968 + 3,$$

so $3^9 - 1$ is not divisible by 10, which means that

$$a^{n-1} = 3^9 \not\equiv 1 \pmod{10},$$

and the Fermat test fails. This shows that 10 is not prime and that $a = 3$ is a witness of this fact.

If the algorithm reports that a is not a witness to the fact that n is composite, does this imply that n is prime? Unfortunately, no. This is because, there may be some composite number n and some candidate counterexample a for which the test says that a is not a counterexample. Such a number a is called a *liar*.

For example, using the Fermat test for $n = 91 = 7 \cdot 13$ and $a = 3$, we can check that

$$a^{n-1} = 3^{90} \equiv 1 \pmod{91},$$

so the Fermat test succeeds even though 91 is not prime. The number $a = 3$ is a liar.

The other reason is that we haven't tested all the candidate counterexamples a for n. In the case where $n = 91$, it can be shown that $2^{90} - 64$ is divisible by 91, so the Fermat test fails for $a = 2$, which confirms that 91 is not prime, and $a = 2$ is a witness of this fact.

Unfortunately, the Fermat test has the property that it may succeed for all candidate counterexamples, even though n is composite. The number $n = 561 = 3 \cdot 11 \cdot 17$ is such a devious number. It can be shown that for all $a \in \{2, \ldots, 560\}$ such that $\gcd(a, 561) = 1$, we have

$$a^{560} \equiv 1 \pmod{561},$$

so *all* these a are liars.

Such composite numbers for which the Fermat test succeeds for all candidate counterexamples are called *Carmichael numbers*, and unfortunately there are infinitely many of them. Thus the Fermat test is doomed. There are various ways of strengthening the Fermat test, but we will not discuss this here. We refer the interested reader to Crandall and Pomerance [5] and Gallier and Quaintance [8].

The remedy is to make sure that we pick a property A such that if n is composite, then at least some candidate a is not a liar, and to test all potential countexamples a. The difficulty is that trying all candidate countexamples can be too expensive to be practical.

There are two classes of primality testing algorithms:

(1) Algorithms that try all possible countexamples, and for which the test does not lie. These algorithms give a definite answer: n is prime or n is composite. Until 2002, no algorithms running in polynomial time, were known. The situation changed in 2002 when a paper with the title "PRIMES is in **P**," by Agrawal, Kayal and Saxena, appeared on the website of the Indian Institute of Technology at Kanpur, India. In this paper, it was shown that testing for primality has a deterministic (nonrandomized) algorithm that runs in polynomial time.

    We will not discuss algorithms of this type here, and instead refer the reader to Crandall and Pomerance [5] and Ribenboim [17].

(2) Randomized algorithms. To avoid having problems with infinite events, we assume that we are testing numbers in some large finite interval $\mathcal{I}$. Given any positive integer $m \in \mathcal{I}$, some candidate witness a is chosen at random. We have a test which, given m and a potential witness a, determines whether or not a is indeed a witness to the fact that m is composite. Such an algorithm is a *Monte Carlo* algorithm, which means the following:

(1) *If the test is positive, then $m \in \mathcal{I}$ is composite.* In terms of probabilities, this is expressed by saying that the conditional probability that $m \in \mathcal{I}$ is composite given that the test is positive is equal to 1. If we denote the event that some positive integer $m \in \mathcal{I}$ is composite by C, then we can express the above as

$$\mathsf{Pr}(C \mid \text{test is positive}) = 1.$$

(2) *If $m \in \mathcal{I}$ is composite, then the test is positive for at least 50% of the choices for a.* We can express the above as

$$\mathsf{Pr}(\text{test is positive} \mid C) \geq \frac{1}{2}.$$

This gives us a degree of confidence in the test.

The contrapositive of (1) says that if $m \in \mathcal{I}$ is prime, then the test is negative. If we denote by P the event that some positive integer $m \in \mathcal{I}$ is prime, then this is expressed as

$$\mathsf{Pr}(\text{test is negative} \mid P) = 1.$$

If we repeat the test ℓ times by picking independent potential witnesses, then the conditional probability that the test is negative ℓ times given that n is composite, written $\mathsf{Pr}(\text{test is negative } \ell \text{ times} \mid C)$, is given by

$$\begin{aligned}
\mathsf{Pr}(\text{test is negative } \ell \text{ times} \mid C) &= \mathsf{Pr}(\text{test is negative} \mid C)^{\ell} \\
&= (1 - \mathsf{Pr}(\text{test is positive} \mid C))^{\ell} \\
&\leq \left(1 - \frac{1}{2}\right)^{\ell} \\
&= \left(\frac{1}{2}\right)^{\ell},
\end{aligned}$$

where we used Property (2) of a Monte Carlo algorithm that

$$\mathsf{Pr}(\text{test is positive} \mid C) \geq \frac{1}{2}$$

and the independence of the trials. *This confirms that if we run the algorithm ℓ times, then* $\mathsf{Pr}($*test is negative ℓ times* $\mid C)$ *is very small.* In other words, it is very unlikely that the test will lie ℓ times (is negative) given that the number $m \in \mathcal{I}$ is composite.

If the probabilty $\mathsf{Pr}(P)$ of the event P is known, which requires knowledge of the distribution of the primes in the interval $\mathcal{I}$, then the conditional probability

$$\mathsf{Pr}(P \mid \text{test is negative } \ell \text{ times})$$

can be determined using Bayes's rule.

A Monte Carlo algorithm does not give a definite answer. However, if ℓ is large enough (say $\ell = 100$), then the conditional probability that the number n being tested is prime given that the test is negative ℓ times, is very close to 1.

Two of the best known randomized algorithms for primality testing are the *Miller–Rabin test* and the *Solovay–Strassen test*. We will not discuss these methods here, and we refer the reader to Gallier and Quaintance [8].

However, what we will discuss is a nondeterministic algorithm that checks that a number n is prime by guessing a certain kind of tree that we call a Lucas tree (because this algorithm is based on a method due to E. Lucas), and then verifies in polynomial time (in the lentgh $\log_2 n$ of the input given in binary) that this tree constitutes a 'proof" that n is indeed prime. This shows that primality testing is in $\mathcal{NP}$, a fact that is not obvious at all. Of course, this is a much weaker result than the AKS algorithm, but the proof that the AKS works in polynomial time (in $\log_2 n$) is much harder.

The Lucas test, and basically all of the primality-testing algorithms, use modular arithmetic and some elementary facts of number theory such as the Euler-Fermat theorem, so we proceed with a review of these concepts.

15.3 Modular Arithmetic, the Groups $\mathbb{Z}/n\mathbb{Z}$, $(\mathbb{Z}/n\mathbb{Z})^*$

Recall the fundamental notion of congruence modulo n and its notation due to Gauss (circa 1802).

Definition 15.2. For any $a, b \in \mathbb{Z}$, we write $a \equiv b \pmod{m}$ iff $a - b = km$, for some $k \in \mathbb{Z}$ (in other words, $a - b$ is divisible by m), and we say that *a and b are congruent modulo m*.

For example, $37 \equiv 1 \pmod 9$, since $37 - 1 = 36 = 4 \cdot 9$. It can also be shown that $200^{250} \equiv 1 \pmod{251}$, but this is impossible to do by brute force, so we will develop some tools to either avoid such computations, or to make them tractable.

It is easy to check that congruence is an equivalence relation but it also satisfies the following properties.

Proposition 15.3. *For any positive integer m, for all $a_1, a_2, b_1, b_2 \in \mathbb{Z}$, the following properties hold. If $a_1 \equiv b_1 \pmod m$ and $a_2 \equiv b_2 \pmod m$, then*

(1) $a_1 + a_2 \equiv b_1 + b_2 \pmod m$.

(2) $a_1 - a_2 \equiv b_1 - b_2 \pmod{m}$.

(3) $a_1 a_2 \equiv b_1 b_2 \pmod{m}$.

Proof. We only check (3), leaving (1) and (2) as easy exercises. Because $a_1 \equiv b_1 \pmod{m}$ and $a_2 \equiv b_2 \pmod{m}$, we have $a_1 = b_1 + k_1 m$ and $a_2 = b_2 + k_2 m$, for some $k_1, k_2 \in \mathbb{Z}$, so we obtain

$$\begin{aligned} a_1 a_2 - b_1 b_2 &= a_1(a_2 - b_2) + (a_1 - b_1)b_2 \\ &= (a_1 k_2 + k_1 b_2)m. \end{aligned}$$

□

Proposition 15.3 allows us to define addition, subtraction, and multiplication on equivalence classes modulo m.

Definition 15.3. Given any positive integer m, we denote by $\mathbb{Z}/m\mathbb{Z}$ the set of equivalence classes modulo m. If we write $\overline{a}$ for the equivalence class of $a \in \mathbb{Z}$, then we define addition, subtraction, and multiplication on residue classes as follows:

$$\begin{aligned} \overline{a} + \overline{b} &= \overline{a + b} \\ \overline{a} - \overline{b} &= \overline{a - b} \\ \overline{a} \cdot \overline{b} &= \overline{ab}. \end{aligned}$$

The above operations make sense because $\overline{a + b}$ does not depend on the representatives chosen in the equivalence classes $\overline{a}$ and $\overline{b}$, and similarly for $\overline{a - b}$ and $\overline{ab}$. Each equivalence class $\overline{a}$ contains a unique representative from the set of remainders $\{0, 1, \ldots, m-1\}$, modulo m, so the above operations are completely determined by $m \times m$ tables. Using the arithmetic operations of $\mathbb{Z}/m\mathbb{Z}$ is called *modular arithmetic*.

The additions tables of $\mathbb{Z}/n\mathbb{Z}$ for $n = 2, 3, 4, 5, 6, 7$ are shown below.

$n = 2$

+	0	1
0	0	1
1	1	0

$n = 3$

+	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

$n = 4$

+	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

$n = 5$

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

$n = 6$

+	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

$n = 7$

+	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	2	3	4	5	6	0
2	2	3	4	5	6	0	1
3	3	4	5	6	0	1	2
4	4	5	6	0	1	2	3
5	5	6	0	1	2	3	4
6	6	0	1	2	3	4	5

It is easy to check that the addition operation + is commutative (abelian), associative, that 0 is an identity element for +, and that every element a has $-a$ as additive inverse, which means that

$$a + (-a) = (-a) + a = 0.$$

It is easy to check that the multiplication operation $\cdot$ is commutative (abelian), associative, that 1 is an identity element for $\cdot$, and that $\cdot$ is distributive on the left and on the right with respect to addition. We usually suppress the dot and write $\overline{a}\,\overline{b}$ instead of $\overline{a} \cdot \overline{b}$. The multiplication tables of $\mathbb{Z}/n\mathbb{Z}$ for $n = 2, 3, \ldots, 9$ are shown below. Since $0 \cdot m = m \cdot 0 = 0$ for all m, these tables are only given for nonzero arguments.

$n = 2$

·	1
1	1

$n = 3$

·	1	2
1	1	2
2	2	1

$n = 4$

·	**1**	2	**3**
1	**1**	2	**3**
2	2	0	2
3	**3**	2	**1**

$n = 5$

·	1	2	3	4
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

$n = 6$

·	**1**	2	3	4	**5**
1	**1**	2	3	4	**5**
2	2	4	0	2	4
3	3	0	3	0	3
4	4	2	0	4	2
5	**5**	4	3	2	**1**

$n = 7$

·	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	4	6	1	3	5
3	3	6	2	5	1	4
4	4	1	5	2	6	3
5	5	3	1	6	4	2
6	6	5	4	3	2	1

$n = 8$

·	**1**	2	**3**	4	**5**	6	**7**
1	**1**	2	**3**	4	**5**	6	**7**
2	2	4	6	0	2	4	6
3	**3**	6	**1**	4	**7**	2	**5**
4	4	0	4	0	4	0	4
5	**5**	2	**7**	4	**1**	6	**3**
6	6	4	2	0	6	4	2
7	**7**	6	**5**	4	**3**	2	**1**

$n = 9$

·	**1**	**2**	3	**4**	**5**	6	**7**	**8**
1	**1**	**2**	3	**4**	**5**	6	**7**	**8**
2	**2**	**4**	6	**8**	**1**	3	**5**	**7**
3	3	6	0	3	6	0	3	6
4	**4**	**8**	3	**7**	**2**	6	**1**	**5**
5	**5**	**1**	6	**2**	**7**	3	**8**	**4**
6	6	3	0	6	3	0	6	3
7	**7**	**5**	3	**1**	**8**	6	**4**	**2**
8	**8**	**7**	6	**5**	**4**	3	**2**	**1**

Examining the above tables, we observe that for $n = 2, 3, 5, 7$, which are primes, every element has an inverse, which means that for every nonzero element a, there is some (actually, unique) element b such that

$$a \cdot b = b \cdot a = 1.$$

For $n = 2, 3, 5, 7$, we say that $\mathbb{Z}/n\mathbb{Z} - \{0\}$ is an abelian group under multiplication. When n is composite, there exist nonzero elements whose product is zero. For example, when $n = 6$, we have $3 \cdot 2 = 0$, when $n = 8$, we have $4 \cdot 4 = 0$, when $n = 9$, we have $6 \cdot 6 = 0$.

For $n = 4, 6, 8, 9$, the elements a that have an inverse are precisely those that are relatively prime to the modulus n (that is, $\gcd(a, n) = 1$).

These observations hold in general. Recall the Bezout theorem: two nonzero integers $m, n \in \mathbb{Z}$ are relatively prime ($\gcd(m, n) = 1$) iff there are integers $a, b \in \mathbb{Z}$ such that

$$am + bn = 1.$$

Proposition 15.4. *Given any integer $n \geq 1$, for any $a \in \mathbb{Z}$, the residue class $\overline{a} \in \mathbb{Z}/n\mathbb{Z}$ is invertible with respect to multiplication iff* $\gcd(a, n) = 1$.

Proof. If $\overline{a}$ has inverse $\overline{b}$ in $\mathbb{Z}/n\mathbb{Z}$, then $\overline{a}\,\overline{b} = 1$, which means that

$$ab \equiv 1 \pmod{n},$$

that is $ab = 1 + nk$ for some $k \in \mathbb{Z}$, which is the Bezout identity

$$ab - nk = 1$$

and implies that $\gcd(a, n) = 1$. Conversely, if $\gcd(a, n) = 1$, then by Bezout's identity there exist $u, v \in \mathbb{Z}$ such that

$$au + nv = 1,$$

so $au = 1 - nv$, that is,

$$au \equiv 1 \pmod{n},$$

which means that $\overline{a}\,\overline{u} = 1$, so $\overline{a}$ is invertible in $\mathbb{Z}/n\mathbb{Z}$. ☐

We have alluded to the notion of a group. Here is the formal definition.

Definition 15.4. A *group* is a set G equipped with a binary operation $\cdot\colon G \times G \to G$ that associates an element $a \cdot b \in G$ to every pair of elements $a, b \in G$, and having the following properties: $\cdot$ is associative, has an identity element $e \in G$, and every element in G is invertible (w.r.t. $\cdot$). More explicitly, this means that the following equations hold for all $a, b, c \in G$:

(G1) $a \cdot (b \cdot c) = (a \cdot b) \cdot c.$ (associativity);

(G2) $a \cdot e = e \cdot a = a.$ (identity);

(G3) For every $a \in G$, there is some $a^{-1} \in G$ such that $a \cdot a^{-1} = a^{-1} \cdot a = e.$ (inverse).

A group G is *abelian* (or *commutative*) if

$$a \cdot b = b \cdot a \quad \text{for all } a, b \in G.$$

It is easy to show that the element e satisfying property (G2) is unique, and for any $a \in G$, the element $a^{-1} \in G$ satisfying $a \cdot a^{-1} = a^{-1} \cdot a = e$ required to exist by (G3) is actually unique. This element is called *the* inverse of a.

The set of integers $\mathbb{Z}$ with the addition operation is an abelian group with identity element 0. The set $\mathbb{Z}/n\mathbb{Z}$ of residues modulo m is an abelian group under addition with identity element 0. In general, $\mathbb{Z}/n\mathbb{Z} - \{0\}$ is *not* a group under multiplication, because some nonzero elements may not have an inverse.

The subset of elements, shown in boldface in the multiplication tables, forms an abelian group under multiplication.

Definition 15.5. The group (under multiplication) of invertible elements of the ring $\mathbb{Z}/n\mathbb{Z}$ is denoted by $(\mathbb{Z}/n\mathbb{Z})^*$. Note that this group is abelian and only defined if $n \geq 2$.

The *Euler φ-function* plays an important role in the theory of the groups $(\mathbb{Z}/n\mathbb{Z})^*$.

Definition 15.6. Given any positive integer $n \geq 1$, the *Euler φ-function* (or Euler *totient function*) is defined such that $\varphi(n)$ is the number of integers a, with $1 \leq a \leq n$, which are relatively prime to n; that is, with $\gcd(a, n) = 1$.[1]

If p is prime, then by definition

$$\varphi(p) = p - 1.$$

We leave it as an exercise to show that if p is prime and if $k \geq 1$, then

$$\varphi(p^k) = p^{k-1}(p - 1).$$

It can also be shown that if $\gcd(m, n) = 1$, then

$$\varphi(mn) = \varphi(m)\varphi(n).$$

The above properties yield a method for computing $\varphi(n)$, based on its prime factorization. If $n = p_1^{i_1} \cdots p_k^{i_k}$, then

$$\varphi(n) = p_1^{i_1 - 1} \cdots p_k^{i_k - 1}(p_1 - 1) \cdots (p_k - 1).$$

For example, $\varphi(17) = 16$, $\varphi(49) = 7 \cdot 6 = 42$,

$$\varphi(900) = \varphi(2^2 \cdot 3^2 \cdot 5^2) = 2 \cdot 3 \cdot 5 \cdot 1 \cdot 2 \cdot 4 = 240.$$

Proposition 15.4 shows that $(\mathbb{Z}/n\mathbb{Z})^*$ has $\varphi(n)$ elements. It also shows that $\mathbb{Z}/n\mathbb{Z} - \{0\}$ is a group (under multiplication) iff n is prime.

Definition 15.7. If G is a finite group, the number of elements in G is called the *the order* of G.

Given a group G with identity element e, and any element $g \in G$, we often need to consider the powers of g defined as follows.

Definition 15.8. Given a group G with identity element e, for any nonnegative integer n, it is natural to define the *power* g^n of g as follows:

$$\begin{aligned} g^0 &= e \\ g^{n+1} &= g \cdot g^n. \end{aligned}$$

[1] We allow $a = n$ to accomodate the special case $n = 1$.

Using induction, it is easy to show that

$$g^m g^n = g^{n+m}$$

for all $m, n \in \mathbb{N}$.

Since g has an inverse g^{-1}, we can extend the definition of g^n to negative powers. For $n \in \mathbb{Z}$, with $n < 0$, let

$$g^n = (g^{-1})^{-n}.$$

Then, it is easy to prove that

$$\begin{aligned} g^i \cdot g^j &= g^{i+j} \\ (g^i)^{-1} &= g^{-i} \\ g^i \cdot g^j &= g^j \cdot g^i \end{aligned}$$

for all $i, j \in \mathbb{Z}$.

Given a finite group G of order n, for any element $a \in G$, it is natural to consider the set of powers $\{e, a^1, a^2, \ldots, a^k, \ldots\}$. A crucial fact is that there is a smallest positive $s \in \mathbb{N}$ such that $a^s = e$, and that s divides n.

Proposition 15.5. *Let G be a finite group of order n. For every element $a \in G$, the following facts hold:*

(1) There is a smallest positive integer $s \leq n$ such that $a^s = e$.

(2) The set $\{e, a, \ldots, a^{s-1}\}$ is an abelian group denoted $\langle a \rangle$.

(3) We have $a^n = e$, and the positive integer s divides n, More generally, for any positive integer m, if $a^m = e$, then s divides m.

Proof. (1) Consider the sequence of $n + 1$ elements

$$(e, a^1, a^2, \ldots, a^n).$$

Since G only has n distinct elements, by the pigeonhole principle, there exist i, j such that $0 \leq i < j \leq n$ such that

$$a^i = a^j.$$

By multiplying both sides by $(a^i)^{-1} = a^{-i}$, we get

$$e = a^i (a^i)^{-1} = a^j (a^i)^{-1} = a^j a^{-i} = a^{j-i}.$$

Since $0 \leq i < j \leq n$, we have $0 \leq j - i \leq n$ with $a^{j-i} = e$. Thus there is some s with $0 < s \leq n$ such that $a^s = e$, and thus a smallest such s.

(2) Since $a^s = e$, for any $i, j \in \{0, \ldots, s-1\}$ if we write $i+j = sq+r$ with $0 \leq r \leq s-1$, we have

$$a^i a^j = a^{i+j} = a^{sq+r} = a^{sq} a^r = (a^s)^q a^r = e^q a^r = a^r,$$

so $\langle a \rangle$ is closed under multiplication. We have $e \in \langle a \rangle$ and the inverse of a^i is a^{s-i}, so $\langle a \rangle$ is a group. This group is obviously abelian.

(3) For any element $g \in G$, let $g\langle a \rangle = \{ga^k \mid 0 \leq k \leq s-1\}$. Observe that for any $i \in \mathbb{N}$, we have

$$a^i \langle a \rangle = \langle a \rangle.$$

We claim that for any two elements $g_1, g_2 \in G$, if $g_1\langle a \rangle \cap g_2\langle a \rangle \neq \emptyset$, then $g_1\langle a \rangle = g_2\langle a \rangle$.

Proof of the claim. If $g \in g_1\langle a \rangle \cap g_2\langle a \rangle$, then there exist $i, j \in \{0, \ldots, s-1\}$ such that

$$g_1 a^i = g_2 a^j.$$

Without loss of generality, we may assume that $i \geq j$. By multipliying both sides by $(a^j)^{-1}$, we get

$$g_2 = g_1 a^{i-j}.$$

Consequently

$$g_2\langle a \rangle = g_1 a^{i-j} \langle a \rangle = g_1 \langle a \rangle,$$

as claimed. ☐

It follows that the pairwise disjoint nonempty subsets of the form $g\langle a \rangle$, for $g \in G$, form a partition of G. However, the map φ_g from $\langle a \rangle$ to $g\langle a \rangle$ given by $\varphi_g(a^i) = ga^i$ has for inverse the map $\varphi_{g^{-1}}$, so φ_g is a bijection, and thus the subsets $g\langle a \rangle$ all have the same number of elements, s. Since these subsets form a partition of G, we must have $n = sq$ for some $q \in \mathbb{N}$, which implies that $a^n = e$.

If $g^m = 1$, then writing $m = sq + r$, with $0 \leq r < s$, we get

$$1 = g^m = g^{sq+r} = (g^s)^q \cdot g^r = g^r,$$

so $g^r = 1$ with $0 \leq r < s$, contradicting the minimality of s, so $r = 0$ and s divides m. ☐

Definition 15.9. Given a finite group G of order n, for any $a \in G$, the smallest positive integer $s \leq n$ such that $a^s = e$ in (1) of Proposition 15.5 is called the *order* of a.

For any integer $n \geq 2$, let $(\mathbb{Z}/n\mathbb{Z})^*$ be the group of invertible elements of the ring $\mathbb{Z}/n\mathbb{Z}$. This is a group of order $\varphi(n)$. Then Proposition 15.5 yields the following result.

Theorem 15.6. *(Euler) For any integer $n \geq 2$ and any $a \in \{1, \ldots, n-1\}$ such that* $\gcd(a, n) = 1$, *we have*

$$a^{\varphi(n)} \equiv 1 \pmod{n}.$$

In particular, if n is a prime, then $\varphi(n) = n - 1$, and we get Fermat's little theorem.

Theorem 15.7. *(Fermat's little theorem) For any prime p and any $a \in \{1, \ldots, p-1\}$, we have*

$$a^{p-1} \equiv 1 \pmod{p}.$$

Since 251 is prime, and since $\gcd(200, 252) = 1$, Fermat's little theorem implies our earlier claim that $200^{250} \equiv 1 \pmod{251}$, without making any computations.

Proposition 15.5 suggests considering groups of the form $\langle g \rangle$.

Definition 15.10. A finite group G is *cyclic* iff there is some element $g \in G$ such that $G = \langle g \rangle$. An element $g \in G$ with this property is called a *generator* of G.

Even though, in principle, a finite cyclic group has a very simple structure, finding a generator for a finite cyclic group is generally hard. For example, it turns out that the multiplicative group $(\mathbb{Z}/p\mathbb{Z})^*$ is a cyclic group when p is prime, but no efficient method for finding a generator for $(\mathbb{Z}/p\mathbb{Z})^*$ is known (besides a brute-force search).

Examining the multiplication tables for $(\mathbb{Z}/n\mathbb{Z})^*$ for $n = 3, 4, \ldots, 9$, we can check the following facts:

1. 2 is a generator for $(\mathbb{Z}/3\mathbb{Z})^*$.
2. 3 is a generator for $(\mathbb{Z}/4\mathbb{Z})^*$.
3. 2 is a generator for $(\mathbb{Z}/5\mathbb{Z})^*$.
4. 5 is a generator for $(\mathbb{Z}/6\mathbb{Z})^*$.
5. 3 is a generator for $(\mathbb{Z}/7\mathbb{Z})^*$.
6. Every element of $(\mathbb{Z}/8\mathbb{Z})^*$ satisfies the equation $a^2 = 1 \pmod{8}$, thus $(\mathbb{Z}/8\mathbb{Z})^*$ has no generators.
7. 2 is a generator for $(\mathbb{Z}/9\mathbb{Z})^*$.

More generally, it can be shown that the multiplicative groups $(\mathbb{Z}/p^k\mathbb{Z})^*$ and $(\mathbb{Z}/2p^k\mathbb{Z})^*$ are cyclic groups when p is an odd prime and $k \geq 1$.

Definition 15.11. A generator of the group $(\mathbb{Z}/n\mathbb{Z})^*$ (when there is one), is called a *primitive root modulo n*.

As an exercise, the reader should check that the next value of n for which $(\mathbb{Z}/n\mathbb{Z})^*$ has no generator is $n = 12$.

The following theorem due to Gauss can be shown. For a proof, see Apostol [1] or Gallier and Quaintance [8].

Theorem 15.8. *(Gauss) For every odd prime p, the group $(\mathbb{Z}/p\mathbb{Z})^*$ is cyclic of order $p - 1$. It has $\varphi(p-1)$ generators.*

The generators of $(\mathbb{Z}/p\mathbb{Z})^*$ are the *primitive roots modulo p*.

15.4 The Lucas Theorem; Lucas Trees

In this section we discuss an application of the existence of primitive roots in $(\mathbb{Z}/p\mathbb{Z})^*$ where p is an odd prime, known an the $n-1$ *test*. This test due to E. Lucas determines whether a positive odd integer n is prime or not by examining the prime factors of $n-1$ and checking some congruences.

The $n-1$ test can be described as the construction of a certain kind of tree rooted with n, and it turns out that the number of nodes in this tree is bounded by $2\log_2 n$, and that the number of modular multiplications involved in checking the congruences is bounded by $2\log_2^2 n$.

When we talk about the complexity of algorithms dealing with numbers, we assume that all inputs (to a Turing machine) are strings representing these numbers, typically in base 2. Since the length of the binary representation of a natural number $n \geq 1$ is $\lfloor \log_2 n \rfloor + 1$ (or $\lceil \log_2(n+1) \rceil$, which allows $n = 0$), the complexity of algorithms dealing with (nonzero) numbers m, n, *etc.* is expressed in terms of $\log_2 m, \log_2 n$, *etc.* Recall that for any real number $x \in \mathbb{R}$, the *floor of* x is the greatest integer $\lfloor x \rfloor$ that is less that or equal to x, and the *ceiling of* x is the least integer $\lceil x \rceil$ that is greater that or equal to x.

If we choose to represent numbers in base 10, since for any base b we have $\log_b x = \ln x / \ln b$, we have

$$\log_2 x = \frac{\ln 10}{\ln 2} \log_{10} x.$$

Since $(\ln 10)/(\ln 2) \approx 3.322 \approx 10/3$, we see that the number of decimal digits needed to represent the integer n in base 10 is approximately 30% of the number of bits needed to represent n in base 2.

Since the Lucas test yields a tree such that the number of modular multiplications involved in checking the congruences is bounded by $2\log_2^2 n$, it is not hard to show that testing whether or not a positive integer n is prime, a problem denoted PRIMES, belongs to the complexity class $\mathcal{NP}$. This result was shown by V. Pratt [15] (1975), but Peter Freyd told me that it was "folklore." Since 2002, thanks to the AKS algorithm, we know that PRIMES actually belongs to the class $\mathcal{P}$, but this is a much harder result.

Here is Lehmer's version of the Lucas result, from 1876.

Theorem 15.9. *(Lucas theorem) Let n be a positive integer with $n \geq 2$. Then n is prime iff there is some integer $a \in \{1, 2, \ldots, n-1\}$ such that the following two conditions hold:*

(1) $a^{n-1} \equiv 1 \pmod{n}$.

(2) If $n > 2$, then $a^{(n-1)/q} \not\equiv 1 \pmod{n}$ for all prime divisors q of $n-1$.

Proof. First, assume that Conditions (1) and (2) hold. If $n = 2$, since 2 is prime, we are done. Thus assume that $n \geq 3$, and let r be the order of a. We claim that $r = n-1$. The condition $a^{n-1} \equiv 1 \pmod{n}$ implies that r divides $n-1$. Suppose that $r < n-1$, and let q

be a prime divisor of $(n-1)/r$ (so q divides $n-1$). Since r is the order or a we have $a^r \equiv 1 \pmod{n}$, so we get

$$a^{(n-1)/q} \equiv a^{r(n-1)/(rq)} \equiv (a^r)^{(n-1)/(rq)} \equiv 1^{(n-1)/(rq)} \equiv 1 \pmod{n},$$

contradicting Condition (2). Therefore, $r = n-1$, as claimed.

We now show that n must be prime. Now $a^{n-1} \equiv 1 \pmod{n}$ implies that a and n are relatively prime so by Euler's Theorem (Theorem 15.6),

$$a^{\varphi(n)} \equiv 1 \pmod{n}.$$

Since the order of a is $n-1$, we have $n-1 \leq \varphi(n)$. If $n \geq 3$ is not prime, then n has some prime divisor p, but n and p are integers in $\{1, 2, \ldots, n\}$ that are not relatively prime to n, so by definition of $\varphi(n)$, we have $\varphi(n) \leq n-2$, contradicting the fact that $n-1 \leq \varphi(n)$. Therefore, n must be prime.

Conversely, assume that n is prime. If $n = 2$, then we set $a = 1$. Otherwise, pick a to be any primitive root modulo p. $\square$

Clearly, if $n > 2$ then we may assume that $a \geq 2$. The main difficulty with the $n-1$ test is not so much guessing the primitive root a, but finding a *complete prime factorization* of $n-1$. However, as a nondeterministic algorithm, the $n-1$ test yields a "proof" that a number n is indeed prime which can be represented as a tree, and the number of operations needed to check the required conditions (the congruences) is bounded by $c\log_2^2 n$ for some positive constant c, and this implies that testing primality is in $\mathcal{NP}$.

Before explaining the details of this method, we sharpen slightly Lucas theorem to deal only with odd prime divisors.

Theorem 15.10. *Let n be a positive odd integer with $n \geq 3$. Then n is prime iff there is some integer $a \in \{2, \ldots, n-1\}$ (a guess for a primitive root modulo n) such that the following two conditions hold:*

(1b) $a^{(n-1)/2} \equiv -1 \pmod{n}$.

(2b) If $n-1$ is not a power of 2, then $a^{(n-1)/2q} \not\equiv -1 \pmod{n}$ for all odd prime divisors q of $n-1$.

Proof. Assume that Conditions (1b) and (2b) of Theorem 15.10 hold. Then we claim that Conditions (1) and (2) of Theorem 15.9 hold. By squaring the congruence $a^{(n-1)/2} \equiv -1 \pmod{n}$, we get $a^{n-1} \equiv 1 \pmod{n}$, which is Condition (1) of Theorem 15.9. Since $a^{(n-1)/2} \equiv -1 \pmod{n}$, Condition (2) of Theorem 15.9 holds for $q = 2$. Next, if q is an odd prime divisor of $n-1$, let $m = a^{(n-1)/2q}$. Condition (1b) means that

$$m^q \equiv a^{(n-1)/2} \equiv -1 \pmod{n}.$$

Now if $m^2 \equiv a^{(n-1)/q} \equiv 1 \pmod{n}$, since q is an odd prime, we can write $q = 2k+1$ for some $k \geq 1$, and then

$$m^q \equiv m^{2k+1} \equiv (m^2)^k m \equiv 1^k m \equiv m \pmod{n},$$

and since $m^q \equiv -1 \pmod{n}$, we get

$$m \equiv -1 \pmod{n}$$

(regardless of whether n is prime or not). Thus we proved that if $m^q \equiv -1 \pmod{n}$ and $m^2 \equiv 1 \pmod{n}$, then $m \equiv -1 \pmod{n}$. By contrapositive, we see that if $m \not\equiv -1 \pmod{n}$, then $m^2 \not\equiv 1 \pmod{n}$ or $m^q \not\equiv -1 \pmod{n}$, but since $m^q \equiv a^{(n-1)/2} \equiv -1 \pmod{n}$ by Condition (1a), we conclude that $m^2 \equiv a^{(n-1)/q} \not\equiv 1 \pmod{n}$, which is Condition (2) of Theorem 15.9. But then, Theorem 15.9 implies that n is prime.

Conversely, assume that n is an odd prime, and let a be any primitive root modulo n. Then by little Fermat we know that

$$a^{n-1} \equiv 1 \pmod{n},$$

so

$$(a^{(n-1)/2} - 1)(a^{(n-1)/2} + 1) \equiv 0 \pmod{n}.$$

Since n is prime, either $a^{(n-1)/2} \equiv 1 \pmod{n}$ or $a^{(n-1)/2} \equiv -1 \pmod{n}$, but since a generates $(\mathbb{Z}/n\mathbb{Z})^*$, it has order $n-1$, so the congruence $a^{(n-1)/2} \equiv 1 \pmod{n}$ is impossible, and Condition (1b) must hold. Similarly, if we had $a^{(n-1)/2q} \equiv -1 \pmod{n}$ for some odd prime divisor q of $n-1$, then by squaring we would have

$$a^{(n-1)/q} \equiv 1 \pmod{n},$$

and a would have order at most $(n-1)/q < n-1$, which is absurd. ☐

If n is an odd prime, we can use Theorem 15.10 to build recursively a tree which is a proof, or certificate, of the fact that n is indeed prime. We first illustrate this process with the prime $n = 1279$.

Example 15.1. If $n = 1279$, then we easily check that $n - 1 = 1278 = 2 \cdot 3^2 \cdot 71$. We build a tree whose root node contains the triple $(1279, ((2,1),(3,2),(71,1)), 3)$, where $a = 3$ is the guess for a primitive root modulo 1279. In this simple example, it is clear that 3 and 71 are prime, but we must supply proofs that these number are prime, so we recursively apply the process to the odd divisors 3 and 71.

Since $3 - 1 = 2^1$ is a power of 2, we create a one-node tree $(3, ((2,1)), 2)$, where $a = 2$ is a guess for a primitive root modulo 3. This is a leaf node.

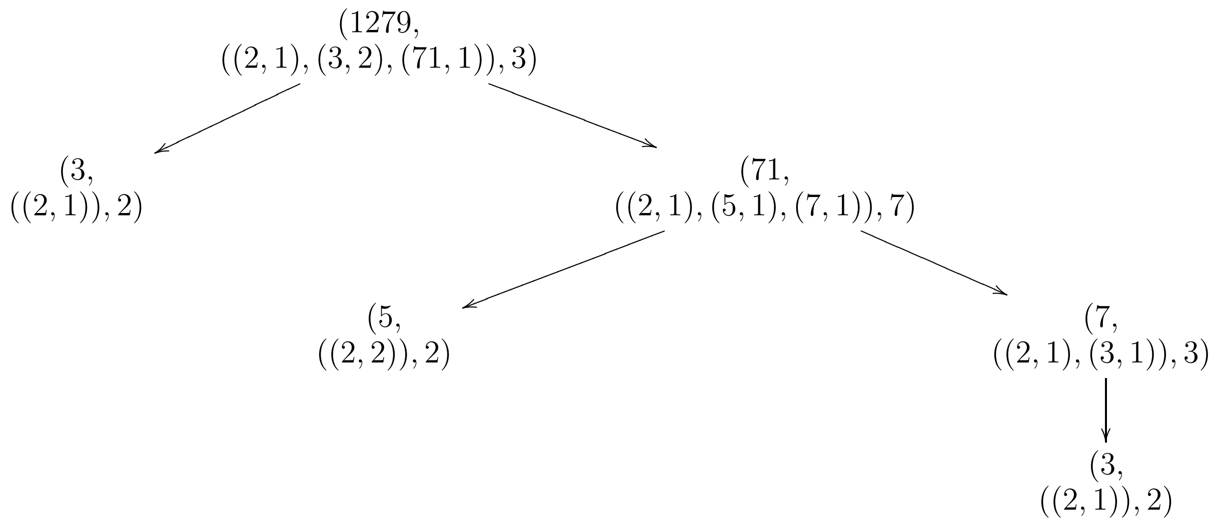
Since $71-1 = 70 = 2\cdot5\cdot7$, we create a tree whose root node is $(71, ((2,1),(5,1),(7,1)), 7)$, where $a = 7$ is the guess for a primitive root modulo 71. Since $5 - 1 = 4 = 2^2$, and

$7-1=6=2\cdot 3$, this node has two successors $(5,((2,2)),2)$ and $(7,((2,1),(3,1)),3)$, where 2 is the guess for a primitive root modulo 5, and 3 is the guess for a primitive root modulo 7.

Since $4=2^2$ is a power of 2, the node $(5,((2,2)),2)$ is a leaf node.

Since $3-1=2^1$, the node $(7,((2,1),(3,1)),3)$ has a single successor, $(3,((2,1)),2)$, where $a=2$ is a guess for a primitive root modulo 3. Since $2=2^1$ is a power of 2, the node $(3,((2,1)),2)$ is a leaf node.

To recap, we obtain the following tree:



We still have to check that the relevant congruences hold at every node. For the root node $(1279,((2,1),(3,2),(71,1)),3)$, we check that

$$3^{1278/2} \equiv 3^{864} \equiv -1 \pmod{1279} \tag{1b}$$

$$3^{1278/(2\cdot 3)} \equiv 3^{213} \equiv 775 \pmod{1279} \tag{2b}$$

$$3^{1278/(2\cdot 71)} \equiv 3^{9} \equiv 498 \pmod{1279}. \tag{2b}$$

Assuming that 3 and 71 are prime, the above congruences check that Conditions (1a) and (2b) are satisfied, and by Theorem 15.10 this proves that 1279 is prime. We still have to certify that 3 and 71 are prime, and we do this recursively.

For the leaf node $(3,((2,1)),2)$, we check that

$$2^{2/2} \equiv -1 \pmod{3}. \tag{1b}$$

For the node $(71,((2,1),(5,1),(7,1)),7)$, we check that

$$7^{70/2} \equiv 7^{35} \equiv -1 \pmod{71} \tag{1b}$$

$$7^{70/(2\cdot 5)} \equiv 7^{7} \equiv 14 \pmod{71} \tag{2b}$$

$$7^{70/(2\cdot 7)} \equiv 7^{5} \equiv 51 \pmod{71}. \tag{2b}$$

Now, we certified that 3 and 71 are prime, assuming that 5 and 7 are prime, which we now establish.

For the leaf node $(5, ((2, 2)), 2)$, we check that

$$2^{4/2} \equiv 2^2 \equiv -1 \pmod{5}. \tag{1b}$$

For the node $(7, ((2, 1), (3, 1)), 3)$, we check that

$$3^{6/2} \equiv 3^3 \equiv -1 \pmod{7} \tag{1b}$$

$$3^{6/(2\cdot 3)} \equiv 3^1 \equiv 3 \pmod{7}. \tag{2b}$$

We have certified that 5 and 7 are prime, given that 3 is prime, which we finally verify.

At last, for the leaf node $(3, ((2, 1)), 2)$, we check that

$$2^{2/2} \equiv -1 \pmod{3}. \tag{1b}$$

The above example suggests the following definition.

Definition 15.12. Given any odd integer $n \geq 3$, a *pre-Lucas tree for* n is defined inductively as follows:

(1) It is a one-node tree labeled with $(n, ((2, i_0)), a)$, such that $n - 1 = 2^{i_0}$, for some $i_0 \geq 1$ and some $a \in \{2, \ldots, n-1\}$.

(2) If $L_1, \ldots, L_k$ are k pre-Lucas (with $k \geq 1$), where the tree L_j is a pre-Lucas tree for some odd integer $q_j \geq 3$, then the tree L whose root is labeled with $(n, ((2, i_0), ((q_1, i_1), \ldots, (q_k, i_k)), a)$ and whose jth subtree is L_j is a *pre-Lucas tree* for n if

$$n - 1 = 2^{i_0} q_1^{i_1} \cdots q_k^{i_k},$$

for some $i_0, i_1, \ldots, i_k \geq 1$, and some $a \in \{2, \ldots, n-1\}$.

Both in (1) and (2), the number a is a guess for a primitive root modulo n.

A pre-Lucas tree for n is a *Lucas tree for* n if the following conditions are satisfied:

(3) If L is a one-node tree labeled with $(n, ((2, i_0)), a)$, then

$$a^{(n-1)/2} \equiv -1 \pmod{n}.$$

(4) If L is a pre-Lucas tree whose root is labeled with $(n, ((2, i_0), ((q_1, i_1), \ldots, (q_k, i_k)), a)$, and whose jth subtree L_j is a pre-Lucas tree for q_j, then L_j is a Lucas tree for q_j for $j = 1, \ldots, k$, and

(a) $a^{(n-1)/2} \equiv -1 \pmod{n}$.

(b) $a^{(n-1)/2q_j} \not\equiv -1 \pmod{n}$ for $j = 1, \ldots, k$.

Since Conditions (3) and (4) of Definition 15.12 are Conditions (1b) and (2b) of Theorem, 15.10, we see that Definition 15.12 has been designed in such a way that Theorem 15.10 yields the following result.

Theorem 15.11. *An odd integer $n \geq 3$ is prime iff it has some Lucas tree.*

The issue is now to see how long it takes to check that a pre-Lucas tree is a Lucas tree. For this, we need a method for computing $x^n \bmod n$ in polynomial time in $\log_2 n$. This is the object of the next section.

15.5 Algorithms for Computing Powers Modulo m

Let us first consider computing the nth power x^n of some positive integer. The idea is to look at the parity of n and to proceed recursively. If n is even, say $n = 2k$, then

$$x^n = x^{2k} = (x^k)^2,$$

so, compute x^k recursively and then square the result. If n is odd, say $n = 2k + 1$, then

$$x^n = x^{2k+1} = (x^k)^2 \cdot x,$$

so, compute x^k recursively, square it, and multiply the result by x.

What this suggests is to write $n \geq 1$ in binary, say

$$n = b_\ell \cdot 2^\ell + b_{\ell-1} \cdot 2^{\ell-1} + \cdots + b_1 \cdot 2^1 + b_0,$$

where $b_i \in \{0, 1\}$ with $b_\ell = 1$ or, if we let $J = \{j \mid b_j = 1\}$, as

$$n = \sum_{j \in J} 2^j.$$

Then we have

$$x^n \equiv x^{\sum_{j \in J} 2^j} = \prod_{j \in J} x^{2^j} \bmod m.$$

This suggests computing the residues r_j such that

$$x^{2^j} \equiv r_j \pmod{m},$$

because then,

$$x^n \equiv \prod_{j \in J} r_j \pmod{m},$$

where we can compute this latter product modulo m two terms at a time.

For example, say we want to compute $999^{179} \bmod 1763$. First, we observe that

$$179 = 2^7 + 2^5 + 2^4 + 2^1 + 1,$$

and we compute the powers modulo 1763:

$$\begin{aligned}
999^{2^1} &\equiv 143 \pmod{1763}\\
999^{2^2} &\equiv 143^2 \equiv 1056 \pmod{1763}\\
999^{2^3} &\equiv 1056^2 \equiv 920 \pmod{1763}\\
999^{2^4} &\equiv 920^2 \equiv 160 \pmod{1763}\\
999^{2^5} &\equiv 160^2 \equiv 918 \pmod{1763}\\
999^{2^6} &\equiv 918^2 \equiv 10 \pmod{1763}\\
999^{2^7} &\equiv 10^2 \equiv 100 \pmod{1763}.
\end{aligned}$$

Consequently,

$$\begin{aligned}
999^{179} &\equiv 999 \cdot 143 \cdot 160 \cdot 918 \cdot 100 \pmod{1763}\\
&\equiv 54 \cdot 160 \cdot 918 \cdot 100 \pmod{1763}\\
&\equiv 1588 \cdot 918 \cdot 100 \pmod{1763}\\
&\equiv 1546 \cdot 100 \pmod{1763}\\
&\equiv 1219 \pmod{1763},
\end{aligned}$$

and we find that

$$999^{179} \equiv 1219 \pmod{1763}.$$

Of course, it would be impossible to exponentiate 999^{179} first and then reduce modulo 1763. As we can see, the number of multiplications needed is bounded by $2\log_2 n$, which is quite good.

The above method can be implemented without actually converting n to base 2. If n is even, say $n = 2k$, then $n/2 = k$ and if n is odd, say $n = 2k + 1$, then $(n-1)/2 = k$, so we have a way of dropping the unit digit in the binary expansion of n and shifting the remaining digits one place to the right without explicitly computing this binary expansion. Here is an algorithm for computing $x^n \bmod m$, with $n \geq 1$, using the *repeated squaring* method.

An Algorithm to Compute $x^n \bmod m$ Using Repeated Squaring

begin
  $u := 1$; $a := x$;

```
    while n > 1 do
      if even(n) then e := 0 else e := 1;
      if e = 1 then u := a · u mod m;
      a := a^2 mod m; n := (n − e)/2
    endwhile;
    u := a · u mod m
  end
```

The final value of u is the result. The reason why the algorithm is correct is that after j rounds through the while loop, $a = x^{2^j} \bmod m$ and

$$u = \prod_{i \in J \mid i < j} x^{2^i} \bmod m,$$

with this product interpreted as 1 when $j = 0$.

Observe that the while loop is only executed $n - 1$ times to avoid squaring once more unnecessarily and the last multiplication $a \cdot u$ is performed outside of the while loop. Also, if we delete the reductions modulo m, the above algorithm is a fast method for computing the nth power of an integer x and the time speed-up of not performing the last squaring step is more significant. We leave the details of the proof that the above algorithm is correct as an exercise.

15.6 PRIMES is in $\mathcal{NP}$

Exponentiation modulo n can performed by repeated squaring, as explained in Section 15.5. In that section, we observed that computing $x^m \bmod n$ requires at most $2\log_2 m$ modular multiplications. Using this fact, we obtain the following result.

Proposition 15.12. *If p is any odd prime, then any pre-Lucas tree L for p has at most $\log_2 p$ nodes, and the number $M(p)$ of modular multiplications required to check that the pre-Lucas tree L is a Lucas tree is less than $2\log_2^2 p$.*

Proof. Let $N(p)$ be the number of nodes in a pre-Lucas tree for p. We proceed by complete induction. If $p = 3$, then $p - 1 = 2^1$, any pre-Lucas tree has a single node, and $1 < \log_2 3$.

Suppose the results holds for any odd prime less than p. If $p - 1 = 2^{i_0}$, then any Lucas tree has a single node, and $1 < \log_2 3 < \log_2 p$. If $p - 1$ has the prime factorization

$$p - 1 = 2^{i_0} q_1^{i_1} \cdots q_k^{i_k},$$

then by the induction hypothesis, each pre-Lucas tree L_j for q_j has less than $\log_2 q_j$ nodes, so

$$N(p) = 1 + \sum_{j=1}^{k} N(q_j) < 1 + \sum_{j=1}^{k} \log_2 q_j = 1 + \log_2(q_1 \cdots q_k) \le 1 + \log_2\left(\frac{p-1}{2}\right) < \log_2 p,$$

establishing the induction hypothesis.

If r is one of the odd primes in the pre-Lucas tree for p, and $r < p$, then there is some other odd prime q in this pre-Lucas tree such that r divides $q-1$ and $q \leq p$. We also have to show that at some point, $a^{(q-1)/2r} \not\equiv -1 \pmod{q}$ for some a, and at another point, that $b^{(r-1)/2} \equiv -1 \pmod{r}$ for some b. Using the fact that the number of modular multiplications required to exponentiate to the power m is at most $2\log_2 m$, we see that the number of multiplications required by the above two exponentiations does not exceed

$$2\log_2\left(\frac{q-1}{2r}\right) + 2\log_2\left(\frac{r-1}{2}\right) < 2\log_2 q - 4 < 2\log_2 p.$$

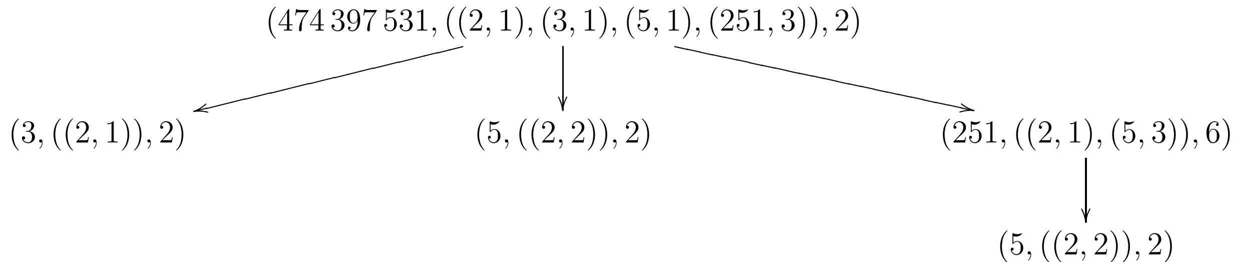
As a consequence, we have

$$M(p) < 2\log_2\left(\frac{p-1}{2}\right) + (N(p)-1)2\log_2 p < 2\log_2 p + (\log_2 p - 1)2\log_2 p = 2\log_2^2 p,$$

as claimed. $\square$

The following impressive example is from Pratt [15].

Example 15.2. Let $n = 474\,397\,531$. It is easy to check that $n - 1 = 474\,397\,531 - 1 = 474\,397\,530 = 2 \cdot 3 \cdot 5 \cdot 251^3$. We claim that the following is a Lucas tree for $n = 474\,397\,531$:



To verify that the above pre-Lucas tree is a Lucas tree, we check that 2 is indeed a primitive root modulo 474 397 531 by computing (using `Mathematica`) that

$$2^{474\,397\,530/2} \equiv 2^{237\,198\,765} \equiv -1 \pmod{474\,397\,531} \tag{1}$$
$$2^{474\,397\,530/(2\cdot 3)} \equiv 2^{79\,066\,255} \equiv 9\,583\,569 \pmod{474\,397\,531} \tag{2}$$
$$2^{474\,397\,530/(2\cdot 5)} \equiv 2^{47\,439\,753} \equiv 91\,151\,207 \pmod{474\,397\,531} \tag{3}$$
$$2^{474\,397\,530/(2\cdot 251)} \equiv 2^{945\,015} \equiv 282\,211\,150 \pmod{474\,397\,531}. \tag{4}$$

The number of modular multiplications is: 27 in (1), 26 in (2), 25 in (3) and 19 in (4).

We have $251 - 1 = 250 = 2 \cdot 5^3$, and we verify that 6 is a primitive root modulo 251 by computing:

$$6^{250/2} \equiv 6^{125} \equiv -1 \pmod{251} \tag{5}$$
$$6^{250/(2\cdot 5)} \equiv 6^{10} \equiv 175 \pmod{251}. \tag{6}$$

The number of modular multiplications is: 6 in (5), and 3 in (6).

We have $5 - 1 = 4 = 2^2$, and 2 is a primitive root modulo 5, since

$$2^{4/2} \equiv 2^2 \equiv -1 \pmod{5}. \tag{7}$$

This takes one multiplication.

We have $3 - 1 = 2 = 2^1$, and 2 is a primitive root modulo 3, since

$$2^{2/2} \equiv 2^1 \equiv -1 \pmod{3}. \tag{8}$$

This takes 0 multiplications.

Therefore, 474 397 531 is prime.

As nice as it is, Proposition 15.12 is deceiving, because *finding* a Lucas tree is hard.

Remark: Pratt [15] presents his method for finding a certificate of primality in terms of a proof system. Although quite elegant, we feel that this method is not as transparent as the method using Lucas trees, which we adapted from Crandall and Pomerance [5]. Pratt's proofs can be represented as trees, as Pratt sketches in Section 3 of his paper. However, Pratt uses the basic version of Lucas' theorem, Theorem 15.9, instead of the improved version, Theorem 15.10, so his proof trees have at least twice as many nodes as ours.

As nice as it is, Proposition 15.12 is deceiving, because *finding* a Lucas tree is hard.

The following nice result was first shown by V. Pratt in 1975 [15].

Theorem 15.13. *The problem* PRIMES *(testing whether an integer is prime) is in* $\mathcal{NP}$.

Proof. Since all even integers besides 2 are composite, we can restrict out attention to odd integers $n \geq 3$. By Theorem 15.11, an odd integer $n \geq 3$ is prime iff it has a Lucas tree. Given any odd integer $n \geq 3$, since all the numbers involved in the definition of a pre-Lucas tree are less than n, there is a finite (very large) number of pre-Lucas trees for n. Given a guess of a Lucas tree for n, checking that this tree is a pre-Lucas tree can be performed in $O(\log_2 n)$, and by Proposition 15.12, checking that it is a Lucas tree can be done in $O(\log_2^2 n)$. Therefore PRIMES is in $\mathcal{NP}$. $\square$

Of course, checking whether a number n is composite is in $\mathcal{NP}$, since it suffices to guess to factors n_1, n_2 and to check that $n = n_1 n_2$, which can be done in polynomial time in $\log_2 n$. Therefore, PRIMES $\in \mathcal{NP} \cap \mathrm{co}\mathcal{NP}$. As we said earlier, this was the situation until the discovery of the AKS algorithm, which places PRIMES in $\mathcal{P}$.

Remark: Altough finding a primitive root modulo p is hard, we know that the number of primitive roots modulo p is $\varphi(\varphi(p))$. If p is large enough, this number is actually quite large. According to Crandal and Pomerance [5] (Chapter 4, Section 4.1.1), if p is a prime and if $p > 200560490131$, then p has more than $p/(2 \ln \ln p)$ primitive roots.

Bibliography

[1] Tom M. Apostol. *Introduction to Analytic Number Theory.* Undergraduate Texts in Mathematics. Springer, first edition, 1976.

[2] Sanjeev Arora and Boaz Barak. *Computational Complexity. A Modern Approach.* Cambridge University Press, first edition, 2009.

[3] Pierre Brémaud. *Markov Chains, Gibbs Fields, Monte Carlo Simulations, and Queues.* TAM, Vol. 31. Springer Verlag, third edition, 2001.

[4] Erhan Cinlar. *Introduction to Stochastic Processes.* Dover, first edition, 2014.

[5] Richard Crandall and Carl Pomerance. *Prime Numbers. A Computational Perspective.* Springer, second edition, 2005.

[6] Martin Davis. Hilbert's tenth problem is unsolvable. *American Mathematical Monthly*, 80(3):233–269, 1973.

[7] Martin Davis, Yuri Matijasevich, and Julia Robinson. Hilbert's tenth problem. diophantine equations: Positive aspects of a negative solution. In *Mathematical Developments Arising from Hilbert Problems*, volume XXVIII, Part 2, pages 323–378. AMS, 1976.

[8] Jean Gallier and Jocelyn Quaintance. Notes on Primality Testing and Public Key Cryptography. Part I: Randomized Algorithms, Miller–Rabin and Solovay–Strassen Tests. Technical report, University of Pennsylvania, Levine Hall, Philadelphia, PA 19104, 2017. pdf file available from http://www.cis.upenn.edu/∼jean/RSA-primality-testing.pdf.

[9] Geoffrey Grimmett and David Stirzaker. *Probability and Random Processes.* Oxford University Press, third edition, 2001.

[10] John G. Kemeny, Snell J. Laurie, and Anthony W. Knapp. *Denumerable Markov Chains.* GTM, Vol. No 40. Springer-Verlag, second edition, 1976.

[11] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, second edition, 1997.

[12] Michael Machtey and Paul Young. *An Introduction to the General Theory of Algorithms.* Elsevier North-Holland, first edition, 1978.

[13] Michael Mitzenmacher and Eli Upfal. *Probability and Computing. Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, first edition, 2005.

[14] Christos H. Papadimitriou. *Computational Complexity*. Pearson, first edition, 1993.

[15] Vaughan R. Pratt. Every prime has a succinct certificate. *SIAM Journal on Computing*, 4(3):214–220, 1975.

[16] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[17] Paulo Ribenboim. *The Little Book of Bigger Primes*. Springer-Verlag, second edition, 2004.

[18] Elaine Rich. *Automata, Computability, and Complexity. Theory and Applications*. Prentice Hall, first edition, 2007.

[19] Mark Stamp. A revealing introduction to hidden markov models. Technical report, San Jose State University, Department of Computer Science, San Jose, California, 2015.