Finite-State Machines

Foundations and Applications to Text Processing and Pattern Recognition

Wojciech Skut, Jakub Piskorski and Jan Daciuk

June 1, 2005

Authors:

Wojciech Skut (wojciech@google.com) Google Inc.
Jakub Piskorski (piskorsk@dfki.de) DFKI GmbH
Jan Daciuk (jandac@eti.pg.gda.pl) Gdańsk University of Technology

Contents

Preface

Following decades of intense research, finite-state machines have established themselves as a formal framework for dealing with a broad range of linguistic phenomena as well as text and speech processing tasks. To name only a few: tokenization, morphology and lexica are very often implemented as finite-state transducers. Regular expressions, familiar from Perl and Unix commands such as `grep` or `sed` and equivalent to finite automata, are a convenient pattern matching tool. Finite-state machines have also been applied to parsing, spelling correction and many other tasks.

Being one of the most thoroughly investigated areas of automata theory, finite-state machines are handled in depth in many brilliant computer science textbooks. So the question arises: why write yet another book on this topic? The answer lies in the specifics of the application. Natural Language Processing (NLP) and Speech Processing often require a perspective different from that of compiler design or Unix tools.

Still, there exist a number of excellent books and tutorials addressing the NLP applications of finite-state machines. However, most of them are closely tied to a specific formalism, such as the XFST toolkit (Karttunen and Beesley 2003) or the INTEX system (Silberztein 1999a). A very good synopsis of finite-state algorithms is provided by Roche and Schabes (1997), but it is limited to the 65 pages of the introductory section. As a result, the relevant knowledge is spread across a vast spectrum of textbooks, journal articles and conference papers: often excellent, but using different notational conventions and requiring different levels of mathematical or computer science background from the reader. All this makes us conclude that there is a need for a textbook explaining the basic finite-state concepts from an NLP-oriented point of view.

The primary aim of this book is to convey the basic intuitions behind finite-state machines and show the readers a number of generic use cases for finite-state methods. Starting from a simple pattern recognition task at the beginning of chapter 1, we introduce different variants of finite-state machines and algorithms for manipulating them. Chapter 2 introduces regular expressions as a convenient formal notation equivalent to finite-state machines. A number of generic formalization techniques are explained in chapter 3. Finally, chapter 4 discusses implementation issues.

We have attempted to keep the mathematical side of this book as simple as possible. Everyone with an undergraduate-level knowledge of set theory and function notation should be able to understand the mathematical underpinning of the concepts and algorithms presented in our book. No previous knowledge of automata theory is required, although some acquaintance with it will certainly be helpful, as will familiarity with basic NLP and Unix/Perl regular expressions.

Of course, it is difficult to cover *all* finite-state NLP applications in a single book. Conscious of this restriction, we have decided to limit the scope of the present edition, published as a reader for a course at the 2005 European Summer School on Logic, Language and Information (ESSLLI), to finite-state automata and their applications, thus leaving out such important concepts as weighted automata and transducers. We envisage extending our book to these topics at a later time.

This book could not have been possible without the support of several people and institutions. We would like to thank Matthew Aylett and Paul Taylor for their valuable comments. Also, we acknowledge the support we received from Rhetorical Systems Ltd in Edinburgh, the German Research Centre for Artificial Intelligence in Saarbrücken and the Gdańsk University of Technology. The organizing committee of the ESSLLI held in August 2005 at the Heriot-Watt University in Edinburgh provided for a publication of our book as a course reader. Special thanks go to Sue Fitt, who kindly agreed to do the proof-reading. All remaining mistakes are, of course, ours.

Chapter 1

Finite-State Automata

This chapter introduces notions that provide the basis for the remaining chapters of our book. After a brief explanation of the basic concepts of formal language theory in section 1.1, a pattern-matching application in section 1.2 serves as an example of how to build a finite-state automaton. Section 1.3 differentiates between deterministic and non-deterministic automata in terms of data structures and computational properties. The remaining sections explain the most common operations on automata: how to turn a non-deterministic into a deterministic one (section 1.4); how to extend an automaton with ϵ-transitions (section 1.5); how to determine the equivalence of two different automata (section 1.6). In section 1.7, we show how to minimize an automaton and list four different minimization algorithms.

1.1 Alphabets and Strings

This book deals with machines that process texts. Texts are composed of words, and words consist of characters. The characters are picked from an *alphabet*: in an English text, we will typically find the lower-case ASCII letters $a \ldots z$, the upper-case ASCII letters $A \ldots Z$, digits, punctuation marks and some special symbols (%, $, &, etc.). In a German or Swedish text, we are also very likely to find diacritics in letters such as ä, ö or ü. A Japanese text may contain Kanji, Katakana and Hiragana characters, but also Latin letters. In general, any finite collection of symbols can be called an alphabet. By convention, the symbol Σ will be used to denote such collections.

Sequences of symbols from an alphabet are called *strings*. Thus, a, ab, $abbbac$ and $gggaz$ are all strings over the ASCII alphabet. The length of a string w, denoted $|w|$, is the number of characters in w. So, $|a| = 1$, $|ab| = 2$ and $|abbbac| = 6$.

The empty sequence of characters is a special case, called the *empty string* and written ϵ. Obviously, $|\epsilon| = 0$.

The set of all strings over an alphabet Σ is denoted Σ^*. Any set of strings is called a *language* (over Σ). Thus, the following sets are called languages over the alphabet $\{a, b\}$:

- $\{a, b, aa\}$ is a finite language consisting of three strings;
- $\{a, aa, aaa, \ldots\}$ is an infinite language that consists of all non-empty sequences of a's;
- Σ^* — the set of all strings over Σ is obviously a language, too;
- $\emptyset$ is the *empty language*, which should not be confused with...
- ... $\{\epsilon\}$, i.e. the language that contains only one word: the empty string.

A number of operations are defined on strings. First, there is concatenation, written $u \cdot v$, or just uv. If $u = a_1 \ldots a_m$ and $v = b_1 \ldots b_n$, then $uv = a_1 \ldots a_m b_1 \ldots b_n$. So, $xxy \cdot zz = xxyzz$. Of course, $u\epsilon = \epsilon u = u$ for any $u \in \Sigma^*$.

If $u = a_1 \dots a_m$ and $v = a_1 \dots a_m \dots a_{m+k}$, then we say that u is a *prefix* of v, written $u <_p v$. Similarly, if $u = a_k \dots a_m$ and $v = a_1 \dots a_m$, then u is a *suffix* of v.

A string u can be reversed, which we denote u^{-1}: $a_1 \dots a_m \longrightarrow a_m \dots a_1$. For example, $(aabac)^{-1} = cabaa$.

1.2 State Diagrams and State Machines

Many text processing applications pre-classify strings into different categories before doing the actual — often category-specific — processing, as in the following examples.

Numbers: The front end of a speech synthesizer needs to distinguish telephone numbers from currency amounts as the two types of expressions are treated differently: *100765* is pronounced *one-oh-oh-seven-six-five* if it is identified as a phone number, and *one hundred thousand seven hundred and sixty-five* if it is part of a currency amount.

Names: A spell checker may distinguish proper names from common words as the former often do not obey the rules of orthography.

URLs and e-mail addresses: Text processors often spot e-mail addresses and URLs in plain text in order to highlight them graphically.

Such classification tasks do not normally present a difficulty to humans as the string categories involved are clearly recognizable by certain distinctive features. For instance, a currency amount is typically preceded by a currency symbol such as $. A proper name is usually a sequence of capitalized words, sometimes preceded by a title such as *Ms*, *Mr*, *Dr*, or *Prof*. The challenge is to make a machine recognize such properties of a string.

As a concrete example, suppose we want to build a device that checks if a given string is a valid e-mail address. We can assume that the format of an e-mail address, e.g. `w.skut@google.com`, is a user name (`w.skut`), followed by `@`, followed by a host name (`google.com`). Both the user name and the host are composed of non-empty sequences of ASCII word characters (`a-z` and `A-Z`, decimal digits, underscores, hyphens). The sequences are separated by dots.

The device should scan the input from left to right, one character at a time, and say "yes" or "no" when it has finished reading. The task itself is quite simple and can be accomplished according to the diagram shown in figure 1.1.

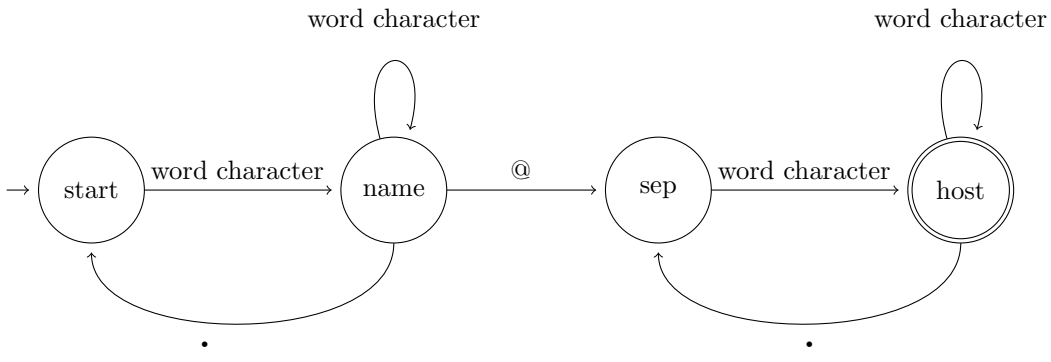


Figure 1.1: A state machine recognizing e-mail addresses.

The circles in the diagram labeled with the symbols `start`, `name`, `sep`, `host` are called *states*. They are connected by arcs labeled with alphabet symbols. The device is initially in the state labeled `start`. Each time it scans a symbol, it follows the arc labeled with the symbol. Unless the arc is a loop, the state of the machine changes.

If the machine encounters a symbol for which no arc exists in the current state, it cannot go any further and we say that the input string has been *rejected*. For example, `peter@paul@gmail` is rejected when the machine arrives at the second `@`.

Likewise, the machine should reject "incomplete" inputs such as `peter` or `sue@`, which are prefixes of valid e-mail addresses. In such cases, the machine ends up in one of the states `start`, `name` or `sep`. The reader may check that, whatever way we enter such a state, the string consumed up to this point is not a legal e-mail address. We call such states *rejecting* because strings that lead to such states are rejected.

On the other hand, if the machine ends up in the state `host`, we can be sure that the string read in is a valid e-mail address. Such states are called *accepting* or *final*. If an input string leads to a final state, we say that the machine *accepts* the string. We adopt the convention of marking final states with double circles. Initial states are marked by short incoming arrows ($\rightarrow$).

The formal construct shown in the diagram is called a *state machine*. The following few points characterize this concept.

States: The state of the machine summarizes the information about the part of the string scanned so far. For instance, `name` means that the string read so far is a potential user name: if we see an `@`, we can proceed to reading the host name. In other words, the concept of a state offers a very simple notion of *memory*.

Alphabet: The machine reads symbols from an *alphabet*. In the case of the above machine, the input alphabet consists of the characters $\{a, \ldots z,\ A, \ldots Z, 1, \ldots 9, \%, \_, @, -, .\}$.

Transitions: The arcs are a graphical representation of *transitions*. A transition captures the notion of a machine jumping from a *source state* to a *target state* on reading an *input symbol*.

Determinism: The machine showed in the above diagram is *deterministic*, i.e. whatever state it is in and whatever symbol it reads, there is at most one arc the machine can follow.

Accordingly, the transitions can be represented by means of a *transition function* δ such that $\delta(q_{source}, a) = q_{target}$ whenever there exists a transition leading from state q_{source} to state q_{target} and consuming symbol a. For example, the arcs in the state diagram shown in figure 1.1 are expressed by the following values of δ:

$$\begin{aligned} \delta(\texttt{start}, a) &= \texttt{name} \\ \delta(\texttt{name}, @) &= \texttt{sep} \\ &\ldots \end{aligned}$$

Function δ maps pairs $\langle q, a\rangle$ of states and symbols to states, which is written formally as $\delta : Q \times \Sigma \rightarrow Q$. Of course, if there is no transition leaving q and labeled a, $\delta(q, a)$ is not defined. In other words, δ is a *partial* function (see also page 8).

In this book, we focus on one particular type of state machines, namely those having a *finite* number of states. Accordingly, they are called *finite-state machines* (FSM). The e-mail address recognizer shown in figure 1.1 clearly belongs to this type. More precisely, it is a *deterministic finite-state automaton* (deterministic FSA, DFSA), which is defined as follows:

Definition 1 *A deterministic finite-state automaton is a quintuple* $A = (\Sigma, Q, q_0, F, \delta)$ *such that*

Σ *is a finite alphabet*

Q *is a finite set of states (the* stateset *of A)*

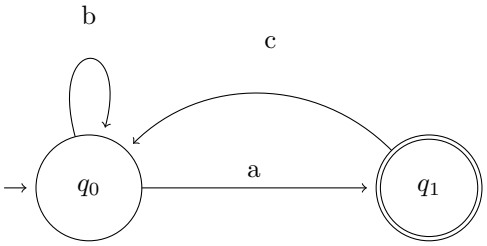
$q_0 \in Q$ *is the* initial *state of A*

$F \subset Q$ *are the* final *(accepting) states of A (we allow more than one final state)*

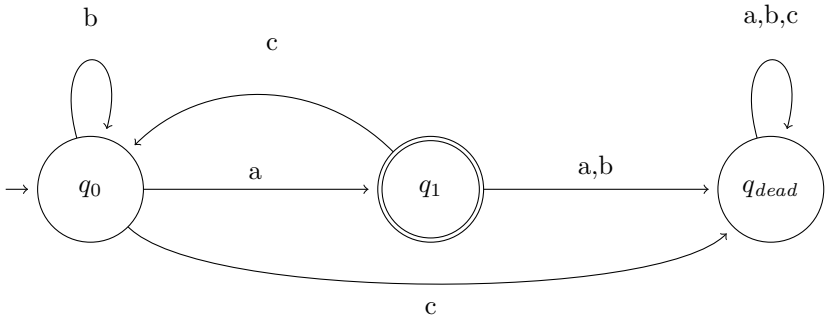
$\delta : Q \times \Sigma \rightarrow Q$ *is the* transition function *of A.*

Complete and Incomplete Automata

In this book, we mostly consider *partial* transition functions $\delta : Q \times \Sigma \to Q$, so that $\delta(q, s)$ may be undefined for certain values of $q \in Q$ and $s \in \Sigma$. Graphically, this corresponds to there being no arc labeled s leaving q, as in the automaton below, where $\delta(q_0, c)$, $\delta(q_1, a)$ and $\delta(q_1, b)$ are not defined. Such automata are called *incomplete*.



An incomplete automaton can be transformed into an equivalent *complete* one, i.e. one with a *total* transition function, defined for each pair $\langle q, s \rangle$. Whenever $\delta(q, s)$ is undefined in the original (incomplete) automaton, we introduce a new transition labeled s and going from q to a new *dead* state (q_{dead}), as shown in the figure below.



The new transition function $\delta_{total}(q, s)$ is thus defined to be equal to q_{dead} for all q and s such that $\delta(q, s)$ is undefined. This includes the case $\delta(q_{dead}, s) = q_{dead}$ for each symbol $s \in \Sigma$.
Informally, we may say that the complete automaton jumps to the dead state whenever the incomplete one would just fail to read the next input symbol. Since $\delta(q_{dead}, s) = q_{dead}$ for all $s \in \Sigma$, the automaton then stays in the dead state until the whole input is consumed. As q_{dead} is a non-final state, the input is thus rejected.
A disadvantage of complete automata is their size: they are larger than equivalent incomplete automata since one has to explicitly represent the dead state and all the transitions that are undefined in an incomplete automaton. In this book, all automata should be assumed to be incomplete unless indicated otherwise.

For example, the machine in figure 1.1 can be viewed as a DFSA $A = (\Sigma, Q, q_0, F, \delta)$ such that: Σ is the 7-bit ASCII character set; $Q = \{\texttt{start}, \texttt{name}, \texttt{sep}, \texttt{host}\}$; $q_0 = \texttt{start}$ is the initial state of A and $F = \{\texttt{host}\}$. The transition function is defined as follows:

$$\delta(\texttt{start}, a) = \texttt{name} \text{ if } a \text{ is a word character, otherwise undefined}$$

$$\delta(\texttt{name}, a) = \begin{cases} \texttt{name} & \text{if } a \text{ is a word character} \\ \texttt{start} & \text{if } a = . \\ \texttt{sep} & \text{if } a = @ \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\delta(\texttt{sep}, a) = \texttt{host} \text{ if } a \text{ is a word character, otherwise undefined}$$

$$\delta(\texttt{host}, a) = \begin{cases} \texttt{host} & \text{if } a \text{ is a word character} \\ \texttt{sep} & \text{if } a = . \\ \text{undefined} & \text{otherwise} \end{cases}$$

Extended Transition Function

The transition function $\delta(q, a)$ tells us to what state the machine jumps from state q when it consumes a single alphabet symbol a. Sometimes it is convenient to check what state is reached when starting in q and reading a *string*, so we could write $\delta(\texttt{start}, address@domain) = \texttt{host}$. For this, we define an *extension* of δ to the domain $Q \times \Sigma^*$:[1]

$$\begin{aligned} \delta(q, \epsilon) &= q \\ \delta(q, ua) &= \begin{cases} \delta(\delta(q, u), a) \text{ if } u \in \Sigma^* \textit{ and both } \delta(q, u), \delta(\delta(q, u), a) \text{ are defined} \\ \text{undefined} \quad \text{otherwise} \end{cases} \end{aligned}$$

This extension allows us to formalize the notion of a DFSA consuming and accepting a string.

Definition 2 *We say that a DFSA* consumes *a string w when $\delta(q_0, w)$ is defined. If, in addition, $\delta(q_0, w) \in F$, we say that the automaton* accepts *w.*

The above definition makes it easy to capture another concept: the language of an automaton.

Definition 3 *Let A be an FSA. The set $\{w \in \Sigma^* : A \text{ accepts } w\}$ of strings accepted by A is called the* language *of A and denoted $\mathcal{L}(A)$.*

The above definition can be generalized to any type of formal machine M as long as the notion of *M accepting a string* can be defined sensibly. Accordingly, the set of all strings accepted by M is then is called its *language* and written $\mathcal{L}(M)$.

Determinism

Determinism is a useful property of automata. Regardless of the current state of the automaton and the next symbol to be scanned, there is always at most one transition that can be taken while reading the symbol (if there is none, we already know that the current input has been rejected). If the automaton has consumed the whole input string ending up in a final state, the string has been accepted.

The acceptance or rejection of any string w can be done in at most $|w|$ steps — one step for each character, no backtracking, as shown in algorithm 1.2.1.

Algorithm 1.2.1: ACCEPT$(A = (\Sigma, Q, q_0, F, \delta), w)$

```
q ← q0
for i ← 1 to |w|
   if δ(q, w[i]) not defined
     return ( false )
    else q ← δ(q, w[i])
if q ∈ F
  return ( true )
 else return ( false )
```

The algorithm takes a DFSA $A = (\Sigma, Q, q_0, F, \delta)$ and a string $w \in \Sigma^*$, and checks whether or not $w \in \mathcal{L}(A)$. The current state of the automaton (q) is initially set to q_0, and then updated to whatever state the transition function points. The main loop (**for** $i = 1$ **to** $|w|$) executes at most $|w|$ times, each time looking up the value of $\delta(q, w[i])$, where i is the i-th character of w. If $\delta(q, w[i])$ is not defined, the algorithm immediately returns *False*. If this does not happen, the loop terminates and the variable q holds the value of $\delta(q_0, w)$, which is then looked up in the set F of final states.

[1] The extension of δ to the domain $Q \times \Sigma^*$ is sometimes denoted δ^* in order to be distingushed from the function δ itself. In order to keep notation as simple as possible, we do not follow this convention in this book. Note that $\delta^*(q, a) = \delta(q, a)$ for all $a \in \Sigma$, so there is no danger of inconsistency.

From a user's prespective, it is very important to know how fast the algorithm is, especially when w is very long. FSAs are often applied to texts in an order of magnitude of gigabytes, so we are primarily interested in seeing how the execution time depends on the length of w.

The total execution time for a string w can be written as a sum of three terms:

$t_{q \leftarrow q_0}$ is the time needed for the assignment $q \leftarrow q_0$;

$t_{\delta(q,a)} \cdot |w|$ is the time needed for looking up a single transition ($t_{\delta(q,a)}$) multiplied by the number of such lookups ($|w|$);

$t_{q \in F}$ is the time needed for checking if state q is final.

Obviously, the values $t_{q \leftarrow q_0}$, $t_{\delta(q,a)}$ and $t_{q \in F}$ are independent of w — they only depend on the size and implementation of the automaton. Thus, it is only the second term ($t_{\delta(q,a)} \cdot |w|$) that grows when w gets longer. The growth of this term is proportional to $|w|$, and the total running time of the algorithm can be expressed as a *linear* function of $|w|$: $t_{total}(w) = \alpha \cdot |w| + \beta$, where α and β are constants independent of w (in our example, $\alpha = t_{\delta(q,a)}$ and $\beta = t_{q \leftarrow q_0} + t_{q \in F}$). We say that the running time of the algorithm is *linear in the length of* w.

For text and speech processing, linear complexity is highly desirable. In the age of the World-Wide Web, we often deal with text collections whose size is measured in gigabytes (1 Gb = 10^9 bytes = 1,000,000,000 bytes) or terabytes (1 Tb = 10^{12} bytes = 1,000,000,000,000 bytes). The processing of one byte with a deterministic automaton usually requires very little time, in the order of magnitude of nanoseconds ($1ns = 10^{-9}s = \frac{1}{1{,}000{,}000{,}000}s$) or at most microseconds ($1\mu s = 10^{-6}s = \frac{1}{1{,}000{,}000}s$). As a result, a linear finite-state algorithm may take a fraction of a second to process a megabyte (1,000,000 bytes) of text, a few minutes for a gigabyte, and a day or two for a terabyte. On the other hand, a *quadratic* ($O(|w|^2)$) algorithm would take days for one megabyte, years for a gigabyte, and millennia for a terabyte of text!

Function Growth and the O-Notation

The value of the constants α and β in a linear function $f(n) = \alpha \cdot n + \beta$ may vary: $f(n) = n + 1000$ is larger for small n than $g(n) = 100 \cdot n + 1$, but g grows much faster than f, so that $g(n) > f(n)$ for $n \geq 11$. What both functions have in common is that for n large enough, their value for $2n$ is approximately twice their value for n. On the other hand, the function $h(n) = 2^n$ does not share this characteristic: already $h(n+1)$ is twice the value of $h(n)$. Intuitively, h belongs to a different "growth class" than f and g.

The "growth class" of a function $f(n)$ is called its *asymptotic complexity* and written $O(f(n))$. It is determined by taking the fastest-growing component of f and ignoring all constant coefficients, which leaves a combination of the functions n^p, p^n and $log\ n$. More formally, the asymptotic complexity of a function $f(n)$ is a function $t(n)$ that forms an upper bound for $f(n)$ for large n, i.e., $f(n) \leq t(n)$ for all $n \geq n_0$ for some $n_0 \geq 0$. The complexity class is denoted using the so called O-notation, i.e., $O(t(n))$ stands for the set of all functions that grow no faster than $t(n)$, disregarding constant factors. For example $O(n + 1000) = O(1000 \cdot n + 1) = O(n)$. Therefore, we say that the asymptotic complexity of both f and g is $O(n)$, while that of h is 2^n.

The most commonly occurring complexity classes are:

Constant $\mathbf{O(1)}$: functions independent of the size of the input;

Logarithmic $\mathbf{O(\log n)}$: very slow growth: $log_2 10 \approx 3.32$, $log_2 100 \approx 6.64$, $log_2 1000 \approx 9.97$

Linear $\mathbf{O(n)}$: growth proportional to the size of input;

Log-linear $\mathbf{O(n \cdot \log n)}$: slower than linear, but better than **quadratic ($\mathbf{O(n^2)}$)**;

Polynomial $\mathbf{O(n^p)}$: for some constant p, includes linear and quadratic;

Exponential $\mathbf{O(p^n)}$: very fast growth; intractable for large values of n.

1.3 Non-Deterministic Automata

Constructing deterministic automata is only easy in the case of very simple structures. Complex ones are much more difficult to design — unless we drop the determinism requirement. For example, consider a machine that recognizes telephone numbers. Such a machine should be able to handle different input formats such as plain local numbers (e.g. **677-54-02-12**), numbers preceded by national area codes (**0131**-667-00-011) and international country codes (**0044**-131-667-00-011, **+44**131-667-00-011).

The basic pattern of such numbers is: an optional international and/or local area code starting with `+`, `0` or `00`, and followed by the local number composed of a sequence of digits and dashes. We assume that the local part of the phone number does not begin with a zero.

The automaton for the local telephone numbers, shown in figure 1.2, is very simple. The arc labels `[0-9]` and `[1-9]` stand for transitions labeled with each of the symbols within the respective range.

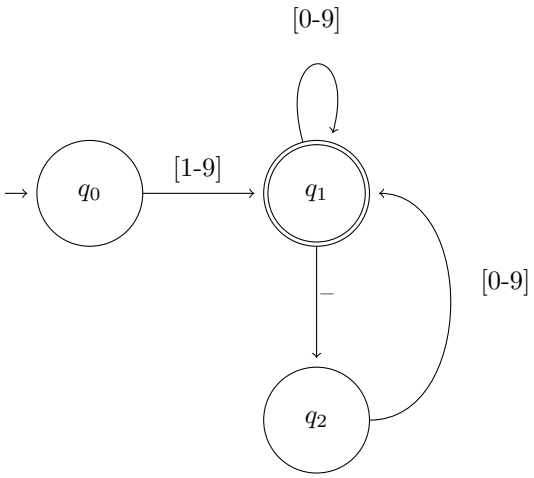


Figure 1.2: FSA recognizing local phone numbers.

The most straightforward way of extending this FSA to national and international dialing codes would be to create a new state diagram by adding an extra initial state q_3, from which the start state of the original FSA could be reached via one of the possible prefixes +, 0, and 00, as shown in figure 1.3. The original start state q_0 should remain an initial state since we want the extended FSA to accept local telephone numbers without a prefix — just as any telephone does.[2]

The arcs of the new state diagram express the intended functionality. However, the machine does not fit the definition of a DFSA. There are two reasons for this non-compliance. Firstly, it has two initial states (q_0 and q_3). Secondly, there are two arcs labeled 0 leaving state q_3. Accordingly, we need to modify the original DFSA definition so that it provides for a) multiple initial states, and b) transition functions that return *sets* of states.

Formally, such a device is called a *non-deterministic finite-state automaton*, and is defined as follows.

Definition 4 *An (ϵ-free) non-deterministic finite-state automaton (NFSA) is a quintuple* $A = (\Sigma, Q, I, F, \Delta)$ *such that*

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

[2]Note that the actual national/international codes (following the prefixes +, 0, and 00) are handled in the original part of the automaton connecting the states q_0, q_1 and q_2. This simplification is discussed again in section 1.5.

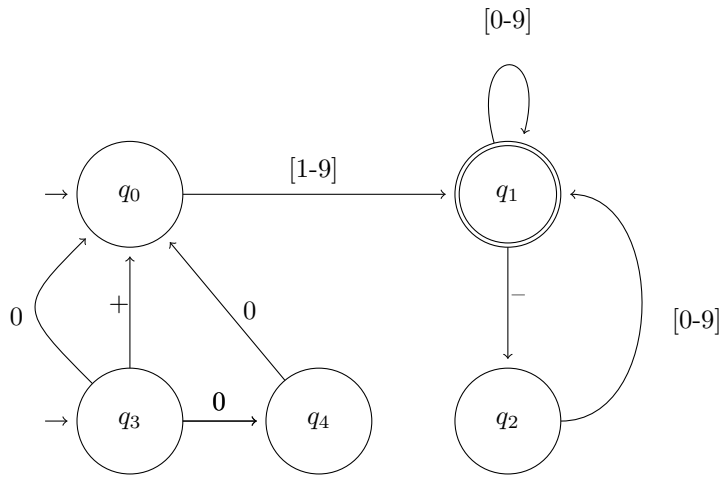


Figure 1.3: NFSA for local and non-local phone numbers.

- $I \subset Q$ *is the set of initial states of* A
- $F \subset Q$ *are the final (accepting) states of* A
- $\Delta : Q \times \Sigma \to 2^Q$ *is the (set-valued) transition function.*

As in a DFSA, the transition function takes a state q and an alphabet symbol a, but now it returns not a single target state, but a *set* of states that can be reached from q via a transition labeled a.[3]

For the automaton shown in figure 1.3, Δ takes the following values:

$$\Delta(q_0, a) = \begin{cases} \{q_1\} & \text{if } a \in \{1, \ldots 9\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\Delta(q_1, a) = \begin{cases} \{q_1\} & \text{if } a \in \{0, \ldots 9\} \\ \{q_2\} & \text{if } a = - \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\Delta(q_2, a) = \begin{cases} \{q_1\} & \text{if } a \in \{0, \ldots 9\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\Delta(q_3, a) = \begin{cases} \{q_0, q_4\} & \text{if } a = 0 \\ \{q_0\} & \text{if } a = + \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\Delta(q_4, a) = \begin{cases} \{q_0\} & \text{if } a = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

The notions of A consuming and accepting a string need to be adapted to the definition of an NFSA. Informally, A accepts a string $w = a_1 \ldots a_t$ if we can get from an initial state to a final state by following transitions labeled with the symbols $a_1 \ldots a_t$. The consecutive transitions form a *path* in A, as stated in the following definition.

Definition 5 *Let* $A = (\Sigma, Q, I, F, \Delta)$ *be an NFSA. Let* $w \in \Sigma^*, w = a_1 \ldots a_t$ *be a string. We say that a sequence* $q_0 \ldots q_t$ *of states is a* path for w *if* $q_i \in \Delta(q_{i-1}, a_i)$ *for* $i = 1 \ldots t$.

[3] Sometimes, it is advantageous to view Δ as a relation over $Q \times \Sigma \times Q$, i.e. a set of triples $\langle q_{source}, a, q_{target} \rangle$ such that $q_{source}, q_{target} \in Q$ and $a \in \Sigma$. The notation $\langle q_{source}, a, q_{target} \rangle \in \Delta$ is then equivalent to $q_{target} \in \Delta(q_{source}, a)$.

Obviously, A consumes a string w if there exists a path $q_0, \ldots, q_{|w|}$ for w such that $q_0 \in I$. A accepts w if there exists a path $q_0, \ldots, q_{|w|}$ for w such that $q_0 \in I$ and $q_{|w|} \in F$.

It is clear from the definitions that a string may be consumed and/or accepted by an NFSA via more than one path $q_0 \ldots q_{|w|}$. As a result, checking the membership of a string in $\mathcal{L}(A)$ is not as easy as for a DFSA.

A possible — although naïve — solution would be to explore one of the possible paths and backtrack if any of the choices turns out to be a failure. For example, if 0131-55-607 is fed to the automaton in figure 1.3, the first choice point is the selection of one of the initial states. If q_0 is chosen, we immediately discover that there are no transitions labeled with the first character of the input string, namely the digit 0, leaving this state. As a result, the other initial state (q_3) must be explored.

From there, the first digit of the input string may lead either to state q_0 or to state q_4. If we go for q_4, we discover a dead-end again: the next character (1) cannot be accepted in this state. In such a case, the device has to backtrack to the previous choice point and explore the other possibility, i.e. following the transition from q_3 to q_0.

This time, we are lucky because the part of the automaton starting at state q_0 is deterministic and the remainder of the string eventually leads to the final state q_1. In general, however, this search technique might be very inefficient. Consider the NFSA shown in figure 1.4.

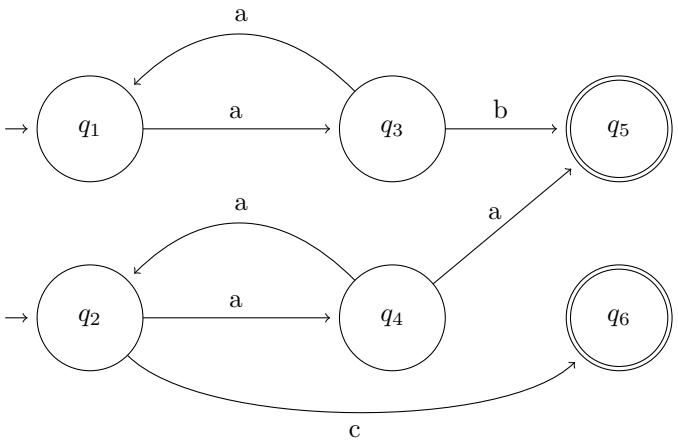


Figure 1.4: An NFSA.

The automaton accepts three classes of strings:

- an odd number of a's followed by a b (along the path $q_1, q_3, q_1, q_3, \ldots, q_3, q_5$)
- an even number of a's followed by a c (along the path $q_2, q_4, q_2, \ldots, q_2, q_6$)
- an even number of a's, no less than 2 (along the path $q_2, q_4, q_2, \ldots, q_4, q_5$).

Now accepting or rejecting a string of the form $a \ldots ac$ may be expensive. If we choose to start in q_1, the automaton will read in the whole sequence of a's alternating between the states q_1 and q_3 only to discover that the last symbol cannot be accepted. Then, we need to backtrack to the beginning, and try the other path, possibly choosing the wrong transition $\langle q_4, a, q_5 \rangle$ for each a while in state q_4.

A better strategy is to keep the set of all states that can be reached in A via the already consumed prefix u of the input string. Suppose we run the NFSA from figure 1.4 on the string $aaab$. For the empty prefix ($u = \epsilon$), the set of reachable states is obviously identical to I. For the first character ($u = a$), we form a new set by taking all states that can be reached from any $q \in I$ by consuming a. We write $\Delta(I, a)$ to denote this set. Figure 1.5 shows the values of $\Delta(I, u)$ for all prefixes u of $aaab$.

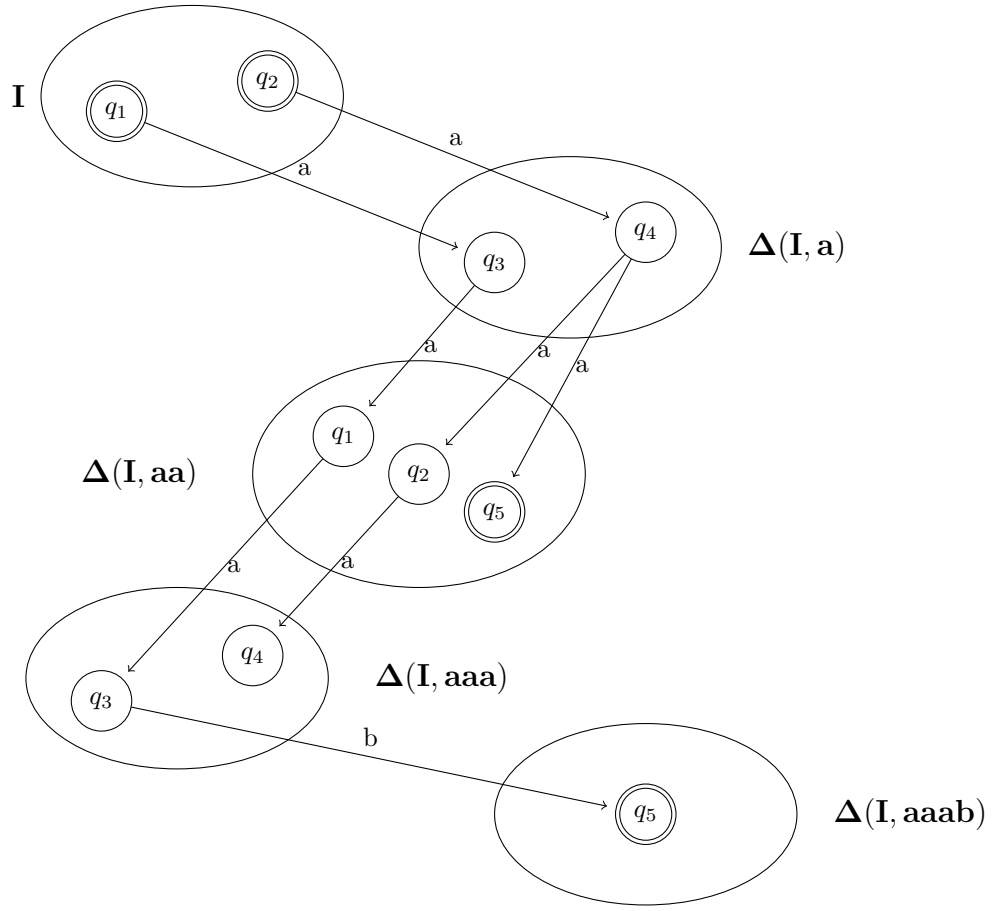


Figure 1.5: The sets $\Delta(I, u)$ of states reachable by the consecutive prefixes u of the string $aaab$.

After consuming the string $aaab$, we end up in the set $\Delta(I, aaab) = \{q_5\}$. Since $q_5 \in F$, it means that there exists a successful path for $aaab$ from I to F in A, hence $aaab \in \mathcal{L}(A)$.

Note that the expression $\Delta(I, a)$ is actually an abuse of notation since the first argument of Δ, according to definition 4, is a *single state* rather than a *set of states*. Therefore, Δ needs to be formally extended to the domain $2^Q \times \Sigma$:[4]

$$\Delta(R, a) \quad = \quad \bigcup_{q \in R} \Delta(q, a) \tag{1.1}$$

Figure 1.6 illustrates this construction idea. In order for expressions such as $\Delta(I, aaa)$ to be well-defined, we must extend the type of the second argument of Δ from single alphabet symbols to strings $u \in \Sigma^*$:

$$\Delta(R, u) \quad = \begin{cases} R & : u = \epsilon \\ \Delta(\Delta(R, v), a) & : u = va, v \in \Sigma^*, a \in \Sigma \end{cases} \tag{1.2}$$

Informally, $\Delta(R, u)$ tells us what states can be reached via string u starting in one of the states in the set $R \subset Q$.

This allows us to formalize the notions of an NFSA A consuming or accepting a string in a way similar to the respective definitions for DFSAs.

[4] If A is a set, then the symbol 2^A denotes the *powerset* of A, i.e. the set of all subsets of A, including the empty set $\emptyset$ and A itself. For example, if $A = \{1, 2\}$, then $2^A = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$.

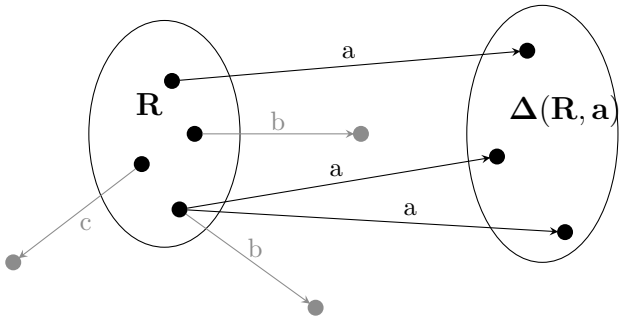


Figure 1.6: Construction of $\Delta(R, a)$ out of a set of states R. $\Delta(R)$ is the set of all states reachable from R via symbol a.

Definition 6 *We say that an NFSA* consumes *a string w when $\Delta(I, w) \neq \emptyset$. If, in addition, $\Delta(I, w) \cap F \neq \emptyset$, we say that the automaton* accepts *w.*

Going back to the search strategy outlined above, it can now be stated that the membership of w in $\mathcal{L}(A)$ can be determined in $|w|$ steps by recursively constructing $\Delta(I, a_1 \dots a_k)$, $k = 0, 1 \dots$ according to formula (1.2), and then checking if $\Delta(I, w) \cap F \neq \emptyset$.

Obviously, this procedure terminates after at most $|w|$ steps, which is an improvement over the naïve backtracking. However, the sets $\Delta(I, a_1 \dots a_k)$ may grow large, $\Delta(I, a_1 \dots a_k) = Q$ in the worst case. Also, there might be as many as $|Q|^2 \cdot |\Sigma|$ transitions leaving such a set. The running time of the algorithm is therefore bounded by $O(|w| \cdot |Q|^2 \cdot |\Sigma|)$. Furthermore, maintaining a set data structure for $\Delta(I, a_1 \dots a_k)$ may also cause some processing overhead. Fortunately, a better solution exists.

1.4 Determinization

Pursuing the idea underlying the construction of the sets $\Delta(I, u)$ a little further, we can arrive at a method of transforming an NFSA into an equivalent DFSA, i.e. a DFSA that accepts exactly the same strings as the original NFSA. The construction of such an equivalent DFSA for an NFSA is called *determinization*.

The trick is to compute the sets $\Delta(I, u)$ off-line and make them states of a DFSA. In this way, each state in the stateset $\hat{Q}$ of the new DFSA $\hat{A} = (\Sigma, \hat{Q}, \hat{q}_0, \hat{F}, \delta)$ is a subset of the original stateset Q.[5] The underlying intuition about these subsets is: if the deterministic automaton has reached a state R by consuming a string w, this means that each $r \in R$ can be reached in the original NFSA A by starting in an initial state and consuming w. This intuition allows us to define the components of $\hat{A}$ as follows.

The initial state. With $w = \epsilon$, we immediately conjecture that $\hat{q}_0 = I$.

The transition function. For any $R \in 2^Q$, the set of states reached from R via a symbol $a \in \Sigma$ can be determined using the recursive formula (1.2). Thus, if we treat subsets of Q as states of a DFSA, the extension of Δ to $2^Q \times \Sigma$ becomes the desired deterministic transition function δ:

$$\delta(R, a) = \Delta(R, a)$$

[5]The determinization method that we are describing is called *subset construction* or *powerset construction* for this reason.

This formula yields $\delta(R, a) = \emptyset$ if no transitions labeled a leave any of the states in R. We interpret this situation as equivalent to $\delta(R, a)$ undefined.

The final states. For a string w to be accepted by A, we must require at least one $r \in R$ to be final, R being the state reached after consuming w. Therefore, the set $\hat{F}$ of final states of $\hat{A}$ is defined as

$$\hat{F} = \{R \in \hat{Q} : R \cap F \neq \emptyset\}$$

The only issue left open is the exact form of $\hat{Q}$, about which it has only been said that it is a subset of 2^Q. Obviously, we can set $\hat{Q} = 2^Q$, and the construction will be correct. However, the size of the DFSA will grow exponentially with the size of the input NFSA, making the determinization of larger automata unfeasible. For example, the determinization of an NFSA with just 33 states would lead to a DFSA with 2^{33} states, which already exceeds the range of integers in most C and C++ compilers. Furthermore, many states in such a fully expanded DFSA may be completely useless. To illustrate this, let us consider the NFSA shown in figure 1.7.

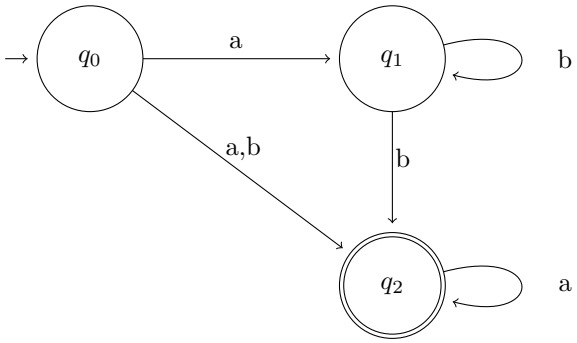


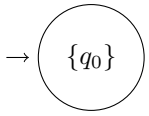
Figure 1.7: An NFSA.

The full powerset expansion of this NFSA yields the structure shown in figure 1.8. The automaton has $2^{|Q|} - 1 = 7$ states.[6] The reader may check that it indeed accepts the same language as the original NFSA. However, it is also easy to see that some of the states are never used. For example, there is no transition entering state $\{q_0, q_1\}$, which means that this state can never be reached from the initial state $\hat{q}_0 = \{q_0\}$.

As a matter of fact, only the states $\{q_0\}$, $\{q_2\}$ and $\{q_1, q_2\}$ can be traversed by a string in Σ^*. Therefore, full powerset expansion of the stateset is not necessarily required in determinization, thus making the construction feasible also for large NFSAs. The algorithm that we present in the remainder of this section takes advantage of this observation and typically constructs only a proper subset of the powerset of Q.

The key observation is that, out of all subsets R of Q, the only ones that matter in the construction are the ones that can be reached by a string $w \in \Sigma^*$ starting in state $\hat{q}_0 = I$. Therefore, we can construct the DFSA by gradually expanding $\hat{Q}$ and δ in the following way.

The first step is to create the initial state $\hat{q}_0$. By definition, $\hat{q}_0 = I = \{q_0\}$.



[6] As explained above, the empty set is ignored.

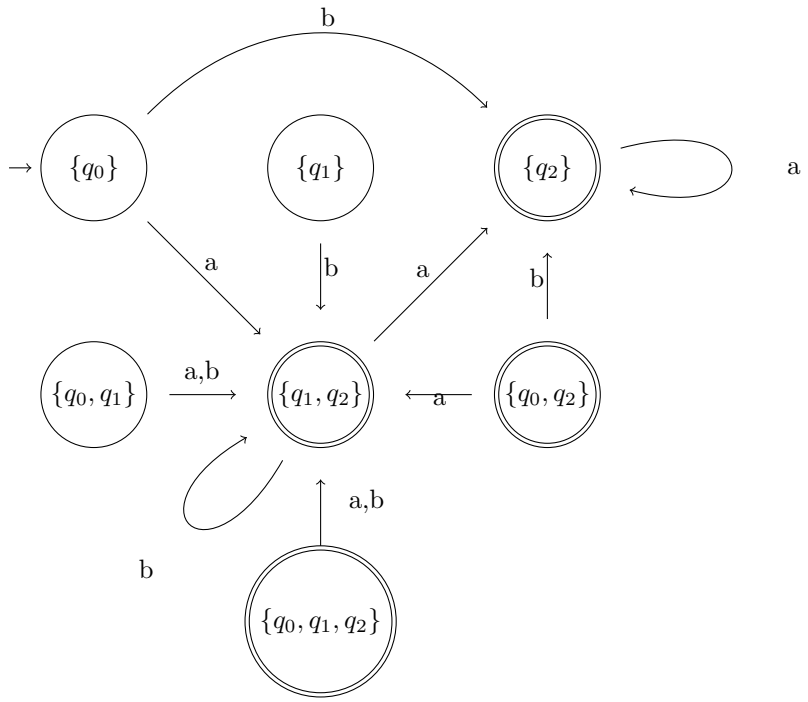
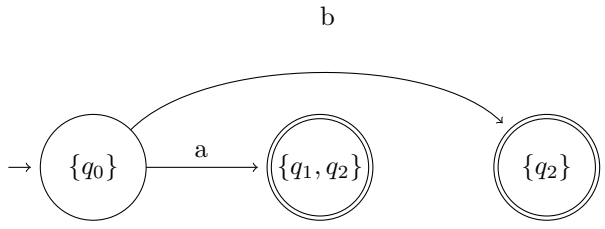


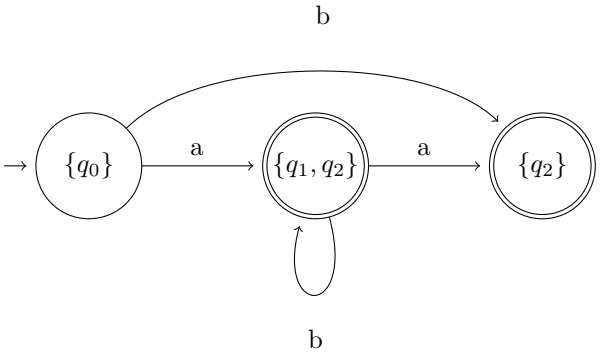
Figure 1.8: Determinization result.

After constructing the initial state $\hat{q}_0$, we construct all states that can be reached from $\hat{q}_0$ by consuming a single symbol s. For $s = a$, the construction method yields $\delta(\hat{q}_0, a) = \{q_1, q_2\}$ because both q_1 and q_2 can be reached from q_0 via a. For $s = b$, we obtain $\delta(\hat{q}_0, b) = \{q_2\}$ because there is only one transition leaving q_0 and labeled b. The intermediate result (the part of the DFSA constructed so far) is shown in the figure below. Note that the new states $\{q_2\}$ and $\{q_1, q_2\}$ are final because $F \cap \{q_2\} = F \cap \{q_1, q_2\} = \{q_2\} \neq \emptyset$.



The above step is the basic building block of the subset construction algorithm. We consider a DFSA state $R \subset Q$ and, for each $s \in \Sigma$, construct $\delta(R, s)$ as the union of all states reachable from R via s. This operation can be called the *expansion* of state R.

In our example, the next states to be *expanded* are those created in the previous step: $\{q_1, q_2\}$, and $\{q_2\}$. For the former, we determine two new transitions: $\delta(\{q_1, q_2\}, a) = \{q_2\}$ and $\delta(\{q_1, q_2\}, b) = \{q_1, q_2\}$ (a loop). Note that both transitions lead to already existing states, so no new ones are constructed. The resulting automaton is shown below.



Having expanded state $\{q_1, q_2\}$, we consider state $\{q_2\}$. This time, there are no transitions labeled b leaving the set $\{q_2\}$, so we leave $\delta(\{q_2\}, b)$ undefined. As for symbol a, the transition $\langle q_2, a, q_2\rangle \in \Delta$ yields the loop $\delta(\{q_2\}, a) = \{q_2\}$. The automaton after adding this loop is shown in figure 1.9.

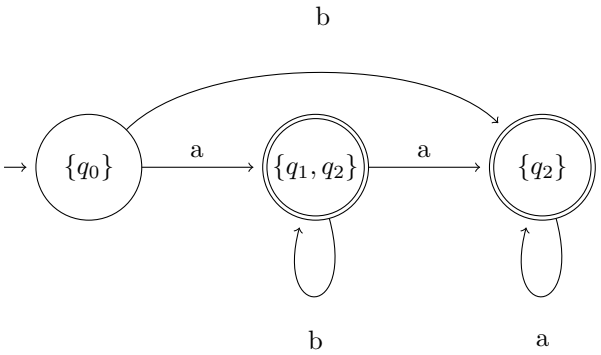


Figure 1.9: DFSA constructed by running the subset construction algorithm on the NFSA shown in figure 1.7.

By now, no new states are left that could be expanded: we have processed all of them. Hence, the DFSA cannot grow any more: its construction has been completed.

A few details are important for an efficient implementation of this algorithm. Firstly, we need to keep track of the states to be expanded. This can be achieved by putting them in a *queue*, i.e. appending them to the end of a list. In each step, the first state in the list is dequeued (removed from the queue), marked as known and processed, and the newly created states are appended to the queue.

The second important implementation issue is that only newly created states should be enqueued. Whenever, during the expansion of a state R, the target state $R' = \delta(R, s)$ of a transition in $\hat{A}$ is determined, we need to check if R' has already been constructed before, in which case it is either already expanded, or awaits expansion in the queue, and need not be re-enqueued. Therefore, we need to maintain a structure representing the part of the DFSA stateset $\hat{Q}$ constructed so far. A good choice is a hash table mapping sets of NFSA states to DFA states: with a good implementation, a set of states can be looked up in such a hash table in almost constant time.

Since each state is placed in the queue exactly once, and the set 2^Q of possible states and the set of transitions are finite, it follows that the queue is empty after at most $|2^Q| = 2^{|Q|}$ steps. Thus, the algorithm always terminates.

The pseudocode of the determinization algorithm is given below. The function DETERMINIZE() takes an NFSA $A = (\Sigma, Q, I, F, \Delta)$ and returns a DFSA $\hat{A} = (\Sigma, \hat{Q}, \hat{q}_0, \hat{F}, \delta)$ such that $\mathcal{L}(\hat{A}) = \mathcal{L}(A)$.

Algorithm 1.4.1: DETERMINIZE$(\Sigma, Q, I, F, \Delta)$

$\hat{q}_0 \leftarrow I$
$\hat{Q} \leftarrow \{\hat{q}_0\}$
ENQUEUE$(Queue, \hat{q}_0)$
while $Queue \neq \emptyset$
  $R \leftarrow$ DEQUEUE$(Queue)$
  for $a \in \Sigma$
    $\delta(R, a) \leftarrow \bigcup_{r \in R} \Delta(r, a)$
    if $\delta(R, a) \notin \hat{Q}$
      $\hat{Q} \leftarrow \hat{Q} \cup \{\delta(R, a)\}$
      ENQUEUE$(Queue, \delta(R, a))$
      if $\delta(R, a) \cap F \neq \emptyset$
        $\hat{F} \leftarrow \hat{F} \cup \{\delta(R, a)\}$
return $(\Sigma, \hat{Q}, \hat{q}_0, \hat{F}, \delta)$

As already mentioned, the maximum number of new states placed in in the queue is $2^{|Q|}$. Accordingly, the worst-case running time of the subset construction algorithm is exponential, i.e. $O(2^{|Q|})$. As a result, operations involving the determinization of an automaton are often the most costly parts of finite-state algorithms.

On the other hand, the actual running time of the algorithm depends on the shape of the NFSA being determinized. As a result, even large NFSAs can often be determinized in a reasonable time.

In practice, subset construction can be made more efficient by employing appropriate data structures, a topic examined in full depth by Leslie (1995). Also, often one does not need to determinize the whole automaton. In such cases, *lazy determinization* applies the subset construction algorithm only to a substructure of the NFSA, which typically results in significant processing time savings (Mohri 1997).

1.5 ϵ-Transitions

In order to model a complex task using finite-state automata, it is common to follow a divide-and-conquer strategy. The task is split into a number of simpler sub-tasks, for each of which an FSA is created. Then, the sub-task FSAs are combined, yielding the desired automaton. The telephone number NFSA presented in section 1.3 can be viewed as a somewhat simplified example of this design: first, a simple DFSA was created for local telephone numbers, and then we extended it with states and transitions that took care of area code prefixes.

However, the extension was done in a rather sloppy way since we just stuck an automaton accepting the prefixes `+`, `0` and `00` to the front of the local telephone number automaton. As a result, the automaton accepts any sequence of numbers starting with `+` provided the next digit is not a zero, irrespective of whether or not the following sequence of digits starts with a valid country code. For example, any string starting with `+214` is accepted although it is not a valid country code.

In order to make the FSA more accurate in a modular way, we can create three automata: one for local phone numbers (LOCALNUMBER), one for national area codes (AREACODE) and one for international country codes (COUNTRYCODE).

In order to combine the three automata into one recognizing different types of telephone numbers, we first create a state diagram describing the intended operational semantics of such a device (figure 1.10). The recognition process starts in an initial state q_0. On reading a 0, the device can move to the AREACODE automaton in order to recognize and consume a national area code.

Alternatively, on seeing either a + or a 00, the device may enter the module COUNTRYCODE, which checks whether or not the remainder of the input string starts with a valid international country code.

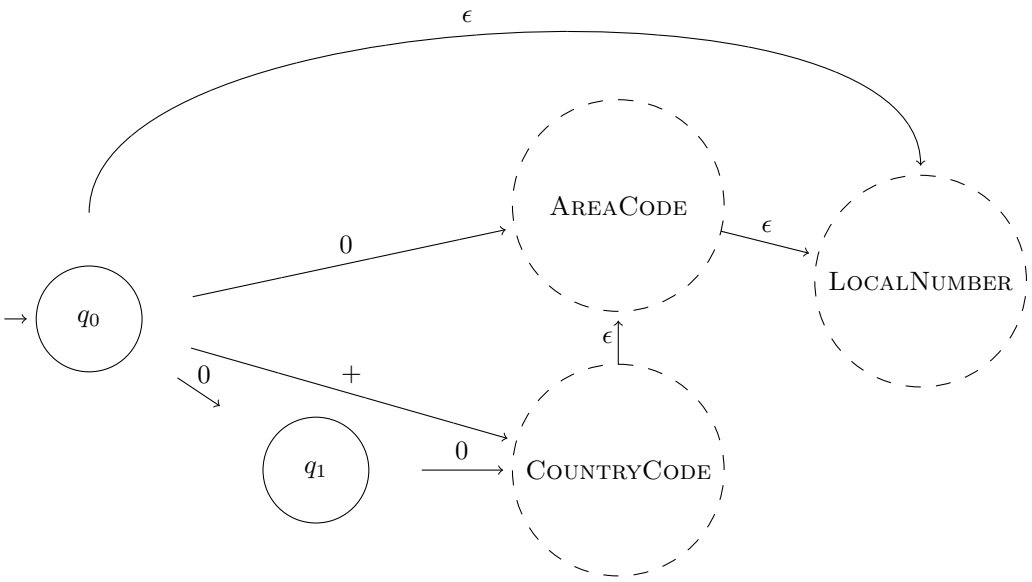


Figure 1.10: A modular NFSA for local and non-local phone numbers. The modules LocalNumber, AreaCode and CountryCode are connected by ϵ-transitions.

The third possibility is that there is no dialing prefix, and the input is just a local number. In this case, we want the automaton to enter the LocalNumber module immediately, without reading any input. Hence, the corresponding arc is labeled with the empty string symbol ϵ. Accordingly, it is called an *ϵ-transition*.[7]

The diagram demonstrates how ϵ-transitions help to keep the FSA modular. However, they also introduce a new kind of non-determinism (in addition to multiple initial states and transitions sharing both the source state and the input label).

Consider the FSA shown in figure 1.11.

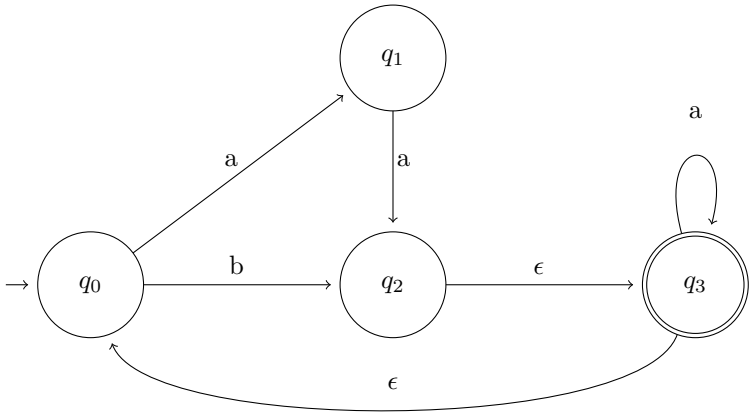


Figure 1.11: An FSA with ϵ-transitions.

Here, non-determinism occurs when the automaton reaches state q_2, e.g. after consuming the string b. The ϵ-transition from q_2 to q_3 means that the FSA might also be in state q_3 or — via the ϵ-transition $\langle q_3, \epsilon, q_0 \rangle$ — in q_0.

The introduction of ϵ-transitions requires a change to the definition of the transition relation.

[7] Such transitions are not allowed by the present NFSA definition, but their semantics is intuitively clear. The required extension of the notion of a non-deterministic automaton is captured by definition 7 on page 21.

Definition 7 *A non-deterministic finite-state automaton with ϵ-transitions (ϵ-NFSA) is a quintuple $A = (\Sigma, Q, I, F, \Delta)$ such that*

- *Σ is a finite alphabet*
- *Q is a finite set of states*
- *$I \subset Q$ is the set of initial states of A*
- *$F \subset Q$ are the final (accepting) states of A*
- *$\Delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$ is the transition function.*

The only difference to definition 4 is that the transition function may now contain transitions $\langle q, \epsilon, q' \rangle$ in addition to transitions of the form $\langle q, a, q' \rangle$, $a \in \Sigma$, already licensed by definition 4. However, determining whether or not a state q can be reached starting in I and consuming a string w can no longer be done in the same way as before. The reason is that the extension of Δ to $2^Q \times \Sigma^*$, as defined by formula (1.2), does not have a provision for ϵ-transitions. For example, the set R of states reachable in the automaton in figure 1.11 from set $I = \{q_0\}$ via string aa is not just $\Delta(I, aa) = \{q_2\}$: $R = \{q_0, q_2, q_3\}$ because q_0 and q_3 can be reached from q_2 via a chain of ϵ-transitions.

In general, whenever a state q can be reached from some $R \subset Q$ by consuming a string $w \in \Sigma^*$, so can any state q' such that there is a chain of ϵ-transitions leading from q to q'. The set of all such states is called the *ϵ-closure* of q. Its exact definition is as follows.

Definition 8 *Let $A = (\Sigma, Q, I, F, \Delta)$ be an ϵ-NFSA. For $q \in Q$, we define the ϵ-closure of q recursively as follows:*

- *$q \in \epsilon$-Closure(q);*
- *if $r \in \epsilon$-Closure(q) and $q' \in \Delta(r, \epsilon)$, then $q' \in \epsilon$-Closure(q).*

For the automaton depicted in figure 1.11, we obtain the following ϵ-closures:

$$\begin{aligned} \epsilon\text{-Closure}(q_0) &= \{q_0\} \\ \epsilon\text{-Closure}(q_1) &= \{q_1\} \\ \epsilon\text{-Closure}(q_2) &= \{q_0, q_2, q_3\} \\ \epsilon\text{-Closure}(q_3) &= \{q_0, q_3\} \end{aligned}$$

The definition of the ϵ-closure of a state can be extended to sets of states. If $R \subset Q$, then

$$\epsilon\text{-Closure}(R) = \bigcup_{q \in R} \epsilon\text{-Closure}(q)$$

In other words, the ϵ-closure of a set R of states is the union of the ϵ-closures of all states $q \in R$.

This allows us to extend the transition function Δ of an ϵ-NFSA to strings as follows:

$$\Delta_\epsilon(R, u) = \begin{cases} \epsilon\text{-Closure}(R) & : u = \epsilon \\ \epsilon\text{-Closure}(\Delta(\Delta_\epsilon(R, v), a)) & : u = va, v \in \Sigma^*, a \in \Sigma \end{cases} \tag{1.3}$$

The first of the two cases distinguished in this formula is clear: the states reachable from R without consuming any input are exactly the states in the ϵ-closure of R. Figure 1.12 illustrates this case.

The second case of formula (1.3) takes care of the recursive definition of $\Delta_\epsilon(R, u)$. If $u = va$, where a is a single alphabet symbol, then we first determine $\Delta_\epsilon(R, v)$, and then take the union of the ϵ-closures of all states reachable from $\Delta_\epsilon(R, v)$ via a, as shown in figure 1.13.

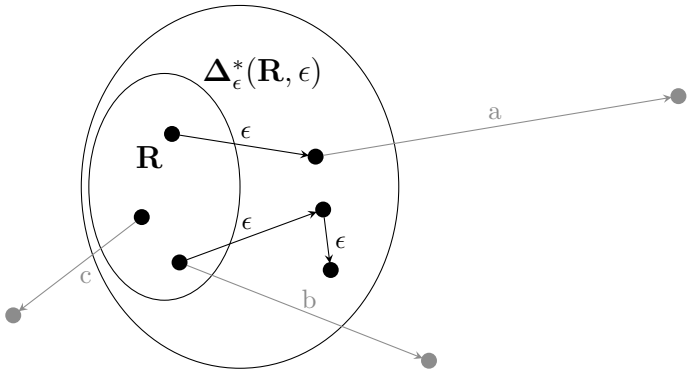


Figure 1.12: States reachable from set R without consuming any input.

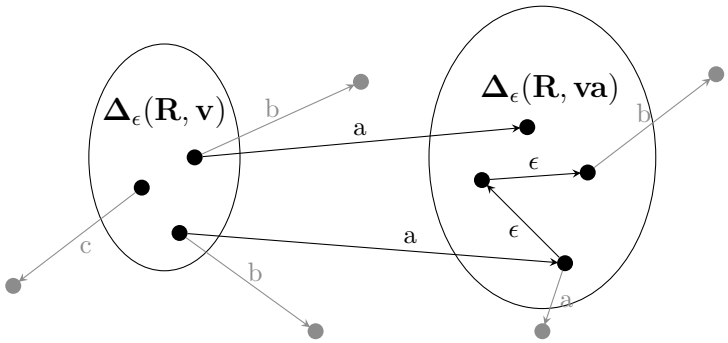


Figure 1.13: Recursive construction of $\Delta_\epsilon(R, va)$ from $\Delta_\epsilon(R, v)$.

Definition 9 *Let $A = (\Sigma, Q, I, F, \Delta)$ be an ϵ-NFSA. Then we say that*

- *A consumes a string $w \in \Sigma^*$ if $\Delta_\epsilon(I, w) \neq \emptyset$;*
- *A accepts w if $\Delta_\epsilon(I, w) \cap F \neq \emptyset$.*

Like ϵ-free NFSAs, NFSAs containing ϵ-transitions can be determinized. However, subset construction (algorithm 1.4.1) fails to produce correct results because it does not have any provision for ϵ-transitions. Recall that, given a state $R \subset Q$ of the DFSA being constructed, the target state of the transition leaving R and consuming a symbol $a \in \Sigma$ is determined according to the following formula:

$$\delta(R, a) = \Delta(R, a) = \{q \in Q : \exists r \in R : (r, a, q) \in \Delta\}$$

Obviously, this formula fails to take into account states that might be reached from a state $r \in R$ by a non-ϵ-transition and a number of ϵ-transitions.

There are two possible solutions. Firstly, it is possible to transform the input NFSA A into an equivalent ϵ-free NFSA A', and then determinize A' using algorithm 1.4.1. The second possibility is to modify algorithm 1.4.1 directly so that it allows for ϵ-transitions. We will explore these two options separately in the following two subsections.

1.5.1 ϵ-Elimination

The ϵ-elimination algorithm proceeds in two steps. First, the procedure Compute-ϵ-Closures() (algorithm 1.5.1) determines the ϵ-closures of the states of the original automaton. Based on this

step, the ϵ-transitions are removed and replaced by equivalent non-ϵ-transitions, ensuring that the resulting automaton is equivalent to the original one.

Step 1: Computation of ϵ-Closures

In order to compute the ϵ-closures of all states in an FSA, we first simplify its structure by eliminating ϵ-*cycles*, i.e. loops consisting entirely of ϵ-transitions. Such a loop is shown in figure 1.14 on the left, where we can go from q_1 to q_2, from q_2 to q_3 and from there back to q_1 without consuming any input.

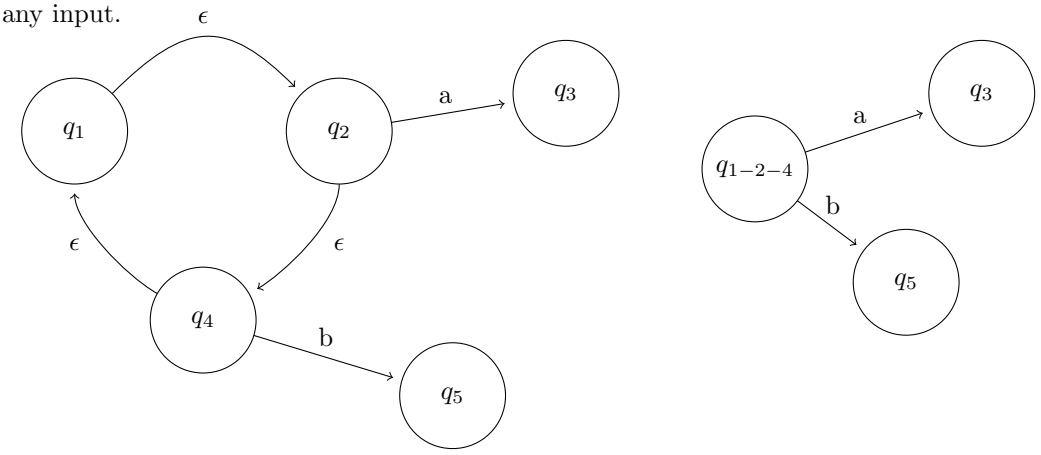


Figure 1.14: States q_1, q_2 and q_4 form an ϵ-loop and can be collapsed into a single state q_{1-2-4}.

The states q_1, q_2 and q_3 form what is called a *strongly connected component* in graph theory: each of them can be reached from any other state in the component (Cormen, Leiserson, Rivest and Stein 2001). Since no input is consumed when we move between the states, they are equivalent and can be merged into a single state, as shown in figure 1.14 on the right. All transitions leaving or entering any of the three states in the ϵ-cycle have to be made to leave/enter the new state (q_{1-2-4}), respectively.

Once ϵ-cycles have been eliminated, the following algorithm, based on depth-first search, can be used to determine the epsilon closures of the states.

Algorithm 1.5.1: COMPUTE-ϵ-CLOSURES$(\Sigma, Q, I, F, \Delta)$

```
for q ∈ Q
   PROCESSED[q] ← false
   ϵ-Closure[q] ← {q}
for q ∈ Q
   if PROCESSED[q] = false
      COMPUTE-ϵ-CLOSURE(q)

procedure COMPUTE-ϵ-CLOSURE(q)
 PROCESSED[q] ← true
 for r ∈ Δ(q, ϵ)
    if PROCESSED[r] = false
       COMPUTE-ϵ-CLOSURE(r)
    ϵ-Closure[q] ← ϵ-Closure[q] ∪ ϵ-Closure[r]
```

The main idea behind this algorithm is that the ϵ-closure of a state q is the union of the set $\{q\}$ and the ϵ-closures of all states $r \in \Delta(q, \epsilon)$, as shown in figure 1.15. Since the automaton is ϵ-cycle-free, the order in which states are considered corresponds to their *topological sort*: each state

q is considered *after* all states r such that $r \in \Delta(q, \epsilon)$. Therefore, the computation of ϵ-Closure(r) is complete before ϵ-Closure(q) $= \bigcup_{r \in \Delta(q,\epsilon)} \epsilon$-Closure($r$) is determined.

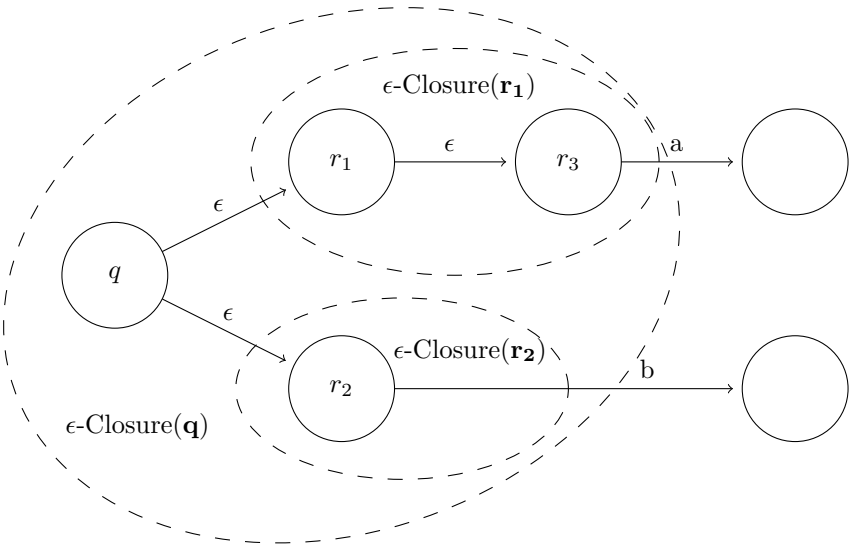


Figure 1.15: Recursive definition of the ϵ-closure of a state q: ϵ-Closure(q) = ϵ-Closure(r_1)$\cup\epsilon$-Closure(r_2)$\cup$ $\{q\}$.

Thus, for each $q \in Q$ we call the subroutine COMPUTE-ϵ-CLOSURE(q), which makes sure that the ϵ-closures of all states $r \in \Delta(q, \epsilon)$ are computed before we take their union with $\{q\}$ and make it the value of the ϵ-closure of q. The procedure calls itself recursively for all states $r' \in \Delta(r, \epsilon)$, etc.

For each state $q \in Q$, we maintain the Boolean flag PROCESSED[q] which tells us whether or not q has already been processed, i.e., the ϵ-closure of q has been computed.

Initially, PROCESSED[q] = $false$ for all q. The value of this flag is set to $true$ as soon as the procedure COMPUTE-ϵ-CLOSURE() is called with argument q. This makes sure the closure computation is done exactly once for each state. As a result, the procedure COMPUTE-ϵ-CLOSURE() is invoked at most $|Q|$ times. The cost of each call to this function depends on the implementation of the set union operation. It is possible to keep it constant, in which case the complexity of algorithm 1.5.1 is $O(|Q|)$.

Step 2: Elimination of ϵ-Transitions

Once the computation of ϵ-closures has been completed, the algorithm creates a new ϵ-free NFSA $A' = (\Sigma, Q', I', F', \Delta')$ equivalent to A.

The basic idea is illustrated by figure 1.16, which shows the path of a string in an ϵ-NFSA. Note that the path is composed of:

- a (possibly empty) sequence of ϵ-transitions starting in an initial state q_0 and ending in some state p;
- a sequence of paths of the form: a non-ϵ-transition followed by a (possibly empty) sequence of ϵ-transitions (here: $q_a \xrightarrow{a} q'_a \xrightarrow{\epsilon} \ldots \xrightarrow{\epsilon} q_b$, $q_b \xrightarrow{b} q'_b \xrightarrow{\epsilon} \ldots \xrightarrow{\epsilon} q_c$ and $q_c \xrightarrow{c} q'_c \xrightarrow{\epsilon} \ldots \xrightarrow{\epsilon} q_f$).

It is easy to see that a slight modification of the automaton can create an ϵ-free path accepting abc by:

- starting in state p straight away (thus making p an initial state);
- folding all paths of the form $q \xrightarrow{x} q' \xrightarrow{\epsilon} \ldots \xrightarrow{\epsilon} q''$ into a single transition $q \xrightarrow{x} q''$.

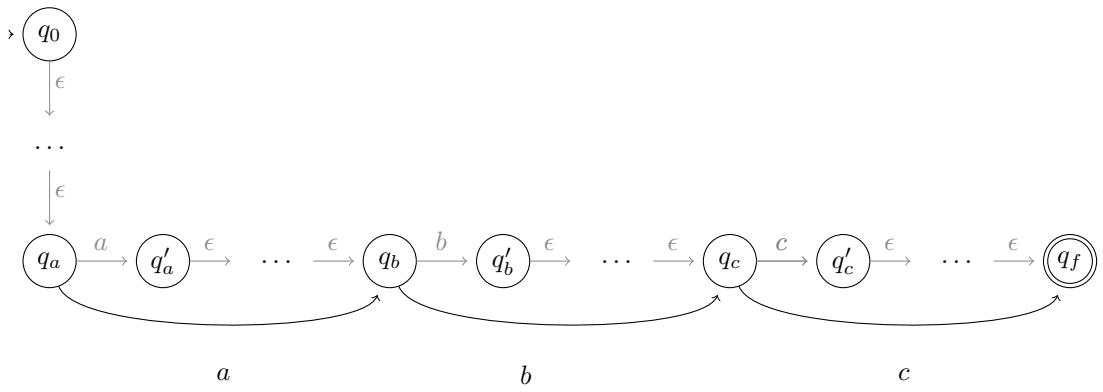


Figure 1.16: Path of the string abc in an ϵ-NFSA and the corresponding ϵ-free path created by merging transitions of the form $q \xrightarrow{x} q' \xrightarrow{\epsilon} \ldots \xrightarrow{\epsilon} q''$ into a single transition $q \xrightarrow{x} q''$.

In order to make sure such an ϵ-free path exists for each string accepted by the original NFSA, we need to:

- make sure that all states reachable from an initial state via ϵ-transitions are initial: $I' = \epsilon$-Closure(I);
- for each transition $\langle q, a, r\rangle$, replace the ϵ-transitions defining the ϵ-closure of r by transitions $\langle q, a, r'\rangle$ such that $r' \in \epsilon$-Closure(r).[8] This step is illustrated by figure 1.17.

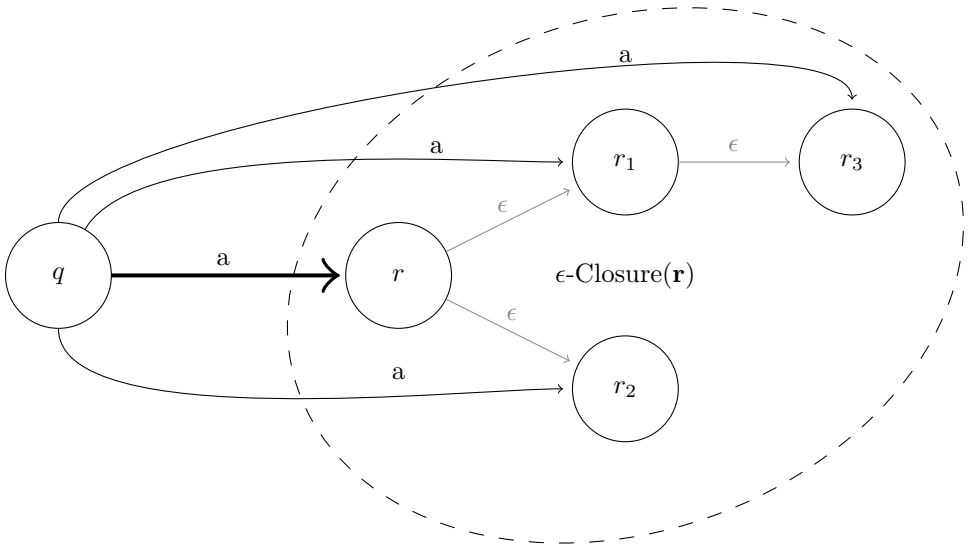


Figure 1.17: Processing of the transition $\langle q, a, r\rangle$ (marked by the thick arrow) in the ϵ-elimination algorithm. The ϵ-transitions (marked in grey) defining the ϵ-closure of q are replaced by transitions labeled a, from q to the respective target states.

The pseudocode of Remove-ϵ() is listed below. It starts with the computation of ϵ-closures. Then all states reachable from I without consuming any input are declared initial ($I' \leftarrow \epsilon$-Closure(I)). The stateset of A' is identical to the stateset Q of A. The set of final states F' and the transition relation Δ' are initially empty.

[8] Note that $r \in \epsilon$-Closure(r), so the original non-ϵ-transition $\langle q, a, r\rangle$ is kept.

After initialization, the algorithm traverses the set of all non-ϵ-transitions in the original automaton A and performs the ϵ-closure expansion illustrated by figure 1.17. Each non-ϵ-transition $\langle q, a, r\rangle$ in A generates transitions of the form $\langle q, a, r'\rangle$ such that r' is in the ϵ-closure of r.
After this operation, the algorithm loops through the states of A' and forms F' out of those states q from which a final state in A can be reached without consuming any input (ϵ-Closure$(q) \cap F \neq \emptyset$).

Algorithm 1.5.2: REMOVE-$\epsilon(\Sigma, Q, I, F, \Delta)$

COMPUTE-ϵ-CLOSURES(A)
$I' \leftarrow \epsilon$-Closure(I)
$F' \leftarrow \emptyset$
$\Delta' \leftarrow \emptyset$
for $\langle q, a, r\rangle \in \Delta, a \neq \epsilon$
  for $r' \in \epsilon$-Closure(r)
    $\Delta' \leftarrow \Delta' \cup \{\langle q, a, r'\rangle\}$
for $q \in Q$
  if ϵ-Closure$[q] \cap F \neq \emptyset$
    $F' \leftarrow F' \cup \{q\}$
return $(A' = (\Sigma, Q, I', F', \Delta'))$

The resulting automaton A' is ϵ-free, so it can be determinized using the basic version of the subset construction algorithm (algorithm 1.4.1).

1.5.2 Subset Construction with ϵ-Transitions

The notion of ϵ-closures can be directly incorporated into the subset construction algorithm. As in the basic algorithm 1.4.1, the main idea is to perform an off-line computation of sets reachable via strings consumed by the original non-deterministic automaton. For an ϵ-free NFSA, these are the possible values of $\Delta(I, w)$ for $w \in \Sigma^*$. For an ϵ-NFSA, we need to consider the possible values of the function $\Delta_\epsilon(I, w)$, as defined by equation (1.3). This consideration leads to the following changes to the algorithm:

The initial state. With $w = \epsilon$, we obtain $\hat{q}_0 = \epsilon$-Closure(I).

The transition function. In the original algorithm, the value $\delta(R, a)$ of the deterministic transition function is determined according to the recursive clause of equation (1.2), which extends $\Delta : 2^Q \times \Sigma \to 2^Q$ to the domain $2^Q \times \Sigma^*$. Thus, $\delta(R, a) = \Delta(R, a)$.

In an ϵ-NFSA, we need to substitute the corresponding clause of equation 1.3, which yields:

$$\delta_\epsilon(R, a) = \epsilon\text{-Closure}(\Delta(R, a)) \tag{1.4}$$

This formula yields $\delta_\epsilon(R, a) = \emptyset$ if no transitions labeled a leave any of the states in R. We interpret this situation as equivalent to $\delta_\epsilon(R, a)$ undefined.

The final states. For a string w to be accepted by A, we require that $\delta_\epsilon(I, w) \cap F \neq \emptyset$ (see definition 9 on page 22). This immediately yields the definition of $\hat{F}$:

$$\hat{F} = \{R \in \hat{Q} : R \cap F \neq \emptyset\}$$

($\hat{Q} \subset 2^Q$ is the stateset of the deterministic automaton).

The actual pseudocode is given in algorithm 1.5.3, which is identical in structure to the original subset construction (algorithm 1.4.1), the only difference being the definition of the initial state ($\hat{q}_0 \leftarrow \epsilon$-Closure$(I)$ as opposed to $\hat{q}_0 \leftarrow I$) and the transition function ($\delta_\epsilon(R, a) \leftarrow \epsilon$-Closure$(\bigcup_{r \in R} \delta(r, a))$ as opposed to $\delta_\epsilon(R, a) \leftarrow \bigcup_{r \in R} \delta(r, a)$).

Algorithm 1.5.3: ϵ-DETERMINIZE$(\Sigma, Q, I, F, \Delta)$

COMPUTE-ϵ-CLOSURES(A)
$\hat{q}_0 \leftarrow \epsilon$-Closure$(I)$
$\hat{Q} \leftarrow \{\hat{q}_0\}$
$\delta_\epsilon \leftarrow \emptyset$
$\hat{F} \leftarrow \emptyset$
ENQUEUE$(Queue, \hat{q}_0)$
while $Queue \neq \emptyset$
  $R \leftarrow$ DEQUEUE$(Queue)$
  for $a \in \Sigma$
    $\delta_\epsilon(R, a) \leftarrow \epsilon$-Closure$(\Delta(R, a))$
    if $\delta_\epsilon(R, a) \notin Q$
      $\hat{Q} \leftarrow \hat{Q} \cup \{\delta_\epsilon(R, a)\}$
      ENQUEUE$(Queue, \delta_\epsilon(R, a))$
      if $\delta_\epsilon(R, a) \cap F \neq \emptyset$
        $\hat{F} \leftarrow \hat{F} \cup \{\delta_\epsilon(R, a)\}$
return $(\Sigma, \hat{Q}, \hat{q}_0, \hat{F}, \delta_\epsilon)$

1.6 Equivalence of Automata

In section 1.4, we saw two different automata accepting exactly the same language, but having a different number of states and transitions (figures 1.7 and 1.8). We call such automata A, A' *equivalent* and write $A \equiv A'$ if $\mathcal{L}(A) = \mathcal{L}(A')$. Obviously, it would be advantageous to be able to determine whether or not two given automata are equivalent. The practical implications are straightforward: for example, replacing the larger automaton by the smaller one would save space (this idea will be fully explored in section 1.7, which deals with *DFSA minimization*).

In the particular case of figures 1.7 and 1.8, the smaller automaton is part of the larger one, the remainder being redundant because the extra states and transitions cannot be accessed from the initial state. However, it may also happen that two completely different automata accept the same language, as shown in figure 1.18.

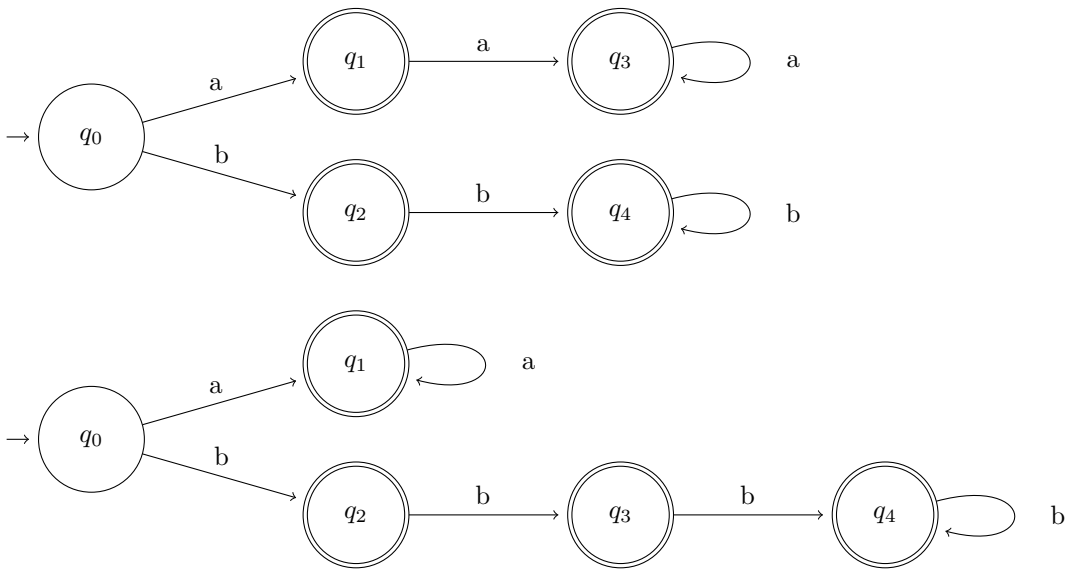


Figure 1.18: Two equivalent DFSAs.

The reader may check that the two DFSAs indeed accept the same language: a non-empty sequence of a's or a non-empty sequence of b's. In order to find an algorithmic way of determining the equivalence of two automata $A = (\Sigma, Q, q_0, F, \delta)$ and $A' = (\Sigma, Q', q_0', F', \delta')$, we start with a very naïve idea, which we will eventually make work realistically. Note that all strings in Σ^* can be enumerated in order of ascending length: ϵ, a, b, aa, ab, ba, bb, aaa, ... We may check for each string w, in this order, whether or not A and A' accept w. If $A \not\equiv A'$, then $w \in \mathcal{L}(A)$ but $w \notin \mathcal{L}(A')$ for some w (or vice versa). As soon as we discover such a w, we know the automata are not equivalent.

Obviously, the outlined procedure may never terminate when applied to a pair of equivalent DFSAs as it keeps checking the membership of increasingly longer w's forever, never finding a counterexample. Fortunately, it turns out that we only need to check strings up to a certain length. Consider the two non-equivalent automata shown in figure 1.19.

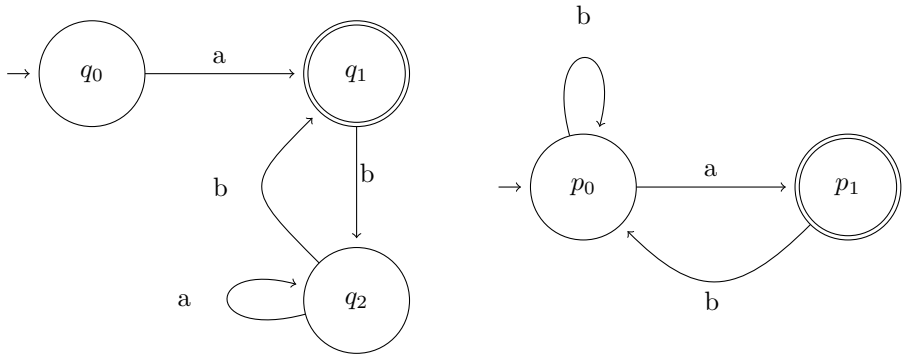


Figure 1.19: Two automata that are not equivalent.

The non-equivalence of the automata is attested e.g. by the string $w = ababbab$, which is accepted by the one on the left (A), but not by the one on the right (A'): $\delta(q_0, ababbab) = q_1 \in F$, but $\delta'(p_0, ababbab) = p_0 \notin F'$.

Aligned, the paths for $w = ababbab$ in both automata yield a sequence of 8 pairs of states $\langle q, p \rangle \in Q \times Q'$.

u	ϵ	a	ab	aba	$abab$	$ababb$	$ababba$	$ababbab$
$\delta(q_0, u)$	q_0	q_1	$\mathbf{q_2}$	q_2	q_1	$\mathbf{q_2}$	q_2	q_1
$\delta'(p_0, u)$	p_0	p_1	$\mathbf{p_0}$	p_1	p_0	$\mathbf{p_0}$	p_1	p_0

However, since $|Q| = 3$ and $|Q'| = 2$, there are only $|Q \times Q'| = 6$ distinct pairs $\langle q, p \rangle$. Therefore, the sequence of state pairs must contain repetitions. Indeed, the pair $\langle q_2, p_0 \rangle$ is reached first after consuming the string ab and then after consuming $ababb$. In other words, the sequence of state pairs contains a *cycle*, as shown in figure 1.20 (the cycle is marked in grey).

The existence of a cycle means that the pair $\langle q_1, p_0 \rangle$ can be reached from $\langle q_0, p_0 \rangle$ via a string shorter than $ababbab$. We can skip the cycle altogether and follow the path $\langle q_0, p_0 \rangle \to \langle q_1, p_1 \rangle \to \langle q_2, p_0 \rangle \to \langle q_2, p_1 \rangle \to \langle q_1, p_0 \rangle$ (marked in black), corresponding to the string $abab$. Thus, if we consider candidate strings in order of increasing length, the non-equivalence of A and A' will be discovered before $ababbab$ is considered.

This result holds for any string w of length $|Q| \cdot |Q'|$ or larger: if the non-equivalence of A and A' is attested by w, then it is also attested by a string w' shorter than w. Therefore, the non-equivalence of A and A' will be discovered before the algorithm reaches the first word of length $|Q| \cdot |Q'|$. As a result, we only need to check words shorter than that; if all of them pass the test,

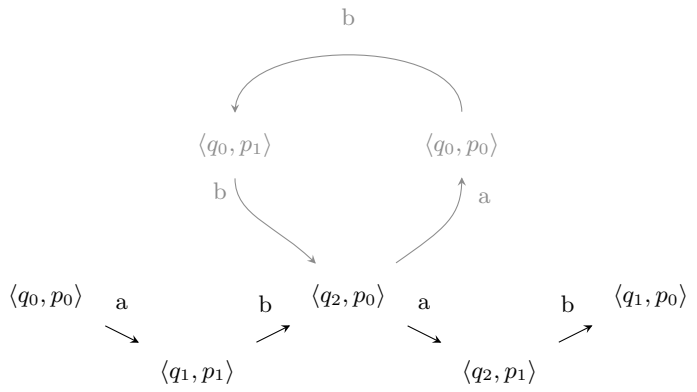


Figure 1.20: The aligned paths of string *ababbab* in two DFSAs A and A', containing a cycle.

we can be sure A and A' are equivalent.

Still, checking every string separately is obviously suboptimal. For example, if aab is the current string, we know that its prefixes ϵ, a and aa have already been tested before. Hence, whenever we reach a pair $\langle q, q' \rangle$ via some string w, we can keep the pair as a partial result which we will pick up when we move to strings of the form wa, $a \in \Sigma$, because then $\delta^*(q_0, wa) = \delta(q, a)$ and $\delta'^*(q'_0, wa) = \delta'(q', a)$.

In order to consider states and strings in this particular order, we may use the technique of exploring all paths in a DFSA using a queue of state pairs, in a way analogous to the queue of states in the subset construction algorithm in section 1.4.

Note that checking each string now requires only one lookup in the transition table of the DFSA per automaton. In the end — at latest after $|Q| \cdot |Q'|$ entries have been processed — there is no way a new pair can be created, and the algorithm terminates.

In addition, a data structure is needed to store the "old" pairs of states, which is crucial in ensuring that the algorithm actually terminates. A hash table can guarantee that this lookup is done in nearly constant time.

The algorithm is presented in the following piece of pseudocode.

Algorithm 1.6.1: CHECKEQUIVALENCE$((\Sigma, Q, q_0, F, \delta), (\Sigma, Q', q'_0, F', \delta'))$

```
if (q_0 ∈ F and q'_0 ∉ F') or (q_0 ∉ F and q'_0 ∈ F')
  return (False)
ENQUEUE(Queue, ⟨q_0, q'_0⟩)
while Queue ≠ ∅
  ⟨q, q'⟩ ← DEQUEUE(Queue)
  for a ∈ Σ
    if both δ(q, a), δ'(q', a) are defined
      if (δ(q, a) ∈ F and δ'(q', a) ∉ F') or (δ(q, a) ∉ F and δ'(q', a) ∈ F')
        return (False)
      else if ⟨δ(q, a), δ'(q', a)⟩ is a new pair
        ENQUEUE(Queue, ⟨δ(q, a), δ'(q', a)⟩)
    else if only one of δ(q, a), δ'(q', a) is defined
      return (False)
return (True)
```

The algorithm checks each accessible state pair $\langle q, q' \rangle$ exactly once, inspecting all transitions leaving q and q'. Since there are at most $|\Sigma|$ transitions leaving each state, the running time of the algorithm is bounded by $O(|Q| \cdot |Q'| \cdot |\Sigma|)$.

1.7 Minimization

The previous section shows that automata accepting the same language may vary in form and size. For obvious reasons, it is desirable to keep an automaton as small as possible. Therefore, two closely related questions arise:

- Can a large automaton be transformed into a smaller one, provided such a smaller one exists?
- If A is a DFSA, is there a *minimal* automaton A_{min} equivalent to A? Is there an algorithm for constructing A_{min} from A?

It turns out that the answer to both questions is "yes". Consider the two DFSAs shown in figure 1.21.

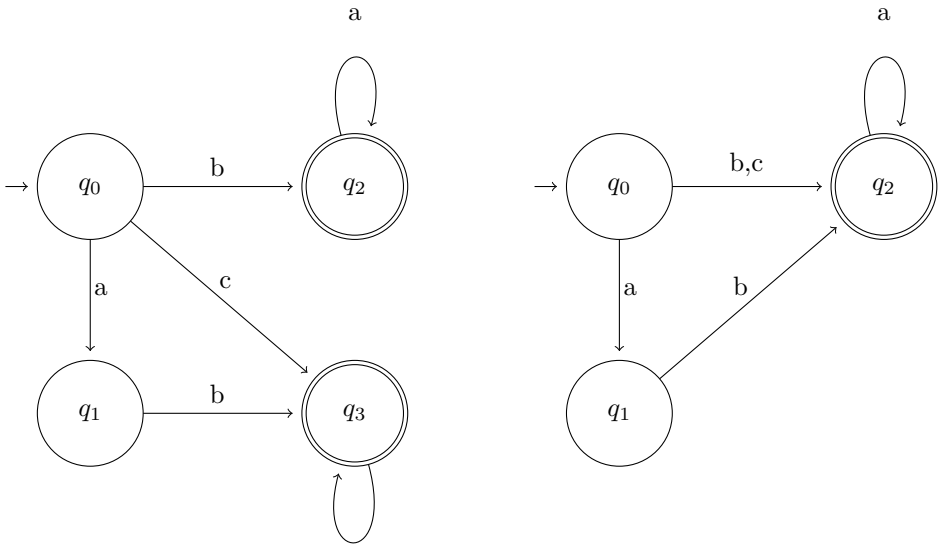


Figure 1.21: Two equivalent DFSAs.

It is obvious that both DFSAs accept the same language: b, c or ab followed by any number of a's (including 0). The DFSA on the right contains only three states, as opposed to the four states in the DFSA on the left.

A closer inspection of the structure of both automata reveals that the smaller DFSA can be created from the larger one by deleting q_3 and redirecting all transitions leading to q_3 so that they point to q_2 instead. This redirection is possible because q_2 and q_3 "do the same job" in the automaton: once the DFSA has entered either state, it stays in it accepting any number of a's (and failing on seeing any symbol other than a). Thus, replacing q_3 with q_2 does not change the functionality of the automaton.

States that "do the same job", such as q_2 and q_3, are called *equivalent*. In the remainder of this section, we will see that a DFSA can be minimized by merging such equivalent states.

In order to formally define the notion of state equivalence, one first needs to formalize the concept of the "job" of a state q in a DFSA. It turns out that the best option is to look at the strings that lead to a final state when we start accepting them in the state under consideration: the set of such strings is $\{\epsilon, a, aa, aaa, \ldots\}$ for both q_2 and q_3. Such a set is called the *right language* of the state q. Its formal definition is as follows.

Definition 10 *(Right language of a state) Let $A = (\Sigma, Q, q_0, F, \delta)$ be a DFSA. The* right language *$\overrightarrow{\mathcal{L}}(q)$ of a state $q \in Q$ is defined as the set of all strings accepted by A starting in state q:*

$$\overrightarrow{\mathcal{L}}(q) = \{w \in \Sigma^* : \delta^*(q, w) \in F\}$$

Obviously, $\overrightarrow{\mathcal{L}}(q_0) = \mathcal{L}(A)$. Furthermore, if the right languages of some two states q and q' are identical, the states are interchangeable as targets of transitions, as we saw in the preceding example.

This definition allows us to capture the notion of state equivalence.

Definition 11 *(State equivalence)*
Let $A = (\Sigma, Q, q_0, F, \delta)$ be a DFSA. If $q, q' \in Q$, then we say that q and q' are equivalent (written $q \equiv q'$) if and only if

$$\overrightarrow{\mathcal{L}}(q) = \overrightarrow{\mathcal{L}}(q')$$

The relation $\equiv$ is an *equivalence relation.*[9] It partitions the stateset Q into a number of equivalence classes, i.e., disjoint sets $Q_1, \ldots Q_m$ such that $\bigcup_{i=1}^{m} Q_i = Q$ and $q \equiv q'$ for all $q, q' \in Q_i$. The *count* of $\equiv$, i.e. the number of equivalence classes, is denoted $|\equiv|$.

Figure 1.22 shows such a partition of a DFSA stateset into 5 classes. The reader may check that a) the right languages of all states belonging to each class are identical (e.g. $\overrightarrow{\mathcal{L}}(q_1) = \overrightarrow{\mathcal{L}}(q_5) = \overrightarrow{\mathcal{L}}(q_6) = \{\epsilon, a, aa, aaa, \ldots\}$) and b) whenever two states q, q' belong to two different classes, $\overrightarrow{\mathcal{L}}(q) \neq \overrightarrow{\mathcal{L}}(r')$.

It is easy to see that a DFSA $A = (\Sigma, Q, q_0, F, \delta)$ containing two equivalent states q and q' can be transformed into a smaller but equivalent automaton $\hat{A} = (\Sigma, Q\backslash\{q'\}, q_0, F\backslash\{q'\}, \hat{\delta})$,[10] where $\hat{\delta}$ is the original transition function δ with all transitions leading to q' redirected to q:

$$\hat{\delta}(r, a) \quad = \begin{cases} q & : \delta(r, a) = q' \\ \delta(r, a) & : \text{ otherwise} \end{cases} \tag{1.5}$$

In order to reduce the size of a given DFSA A, one can follow the following two-step procedure.

- determine all pairs of equivalent states q, q';
- apply the above reduction step until no such pair q, q' is left in the automaton.

It can be proved (the Myhill-Nerode theorem) that the result is indeed the smallest DFSA (in terms of the size of Q) accepting the language $\mathcal{L}(A)$.

Informally, the argument goes as follows. Since we never merge two equivalence classes, their number remains constant throughout the reduction steps. At the end, we are left with one state per equivalence class. This is intuitive: we cannot do any better, because we need $|\equiv|$ states in order to distinguish the $|\equiv|$ right languages found in in $\mathcal{L}(A)$. On the other hand, as long as $|Q| > |\equiv|$, we can reduce the size of Q by applying the above reduction step.

This leads to the following minimality criterion:

Proposition 1 *(Minimality criterion for DFSAs)*
Let $A = (\Sigma, Q, q_0, F, \delta)$ be a DFSA. A is minimal if and only if there is no pair of distinct but equivalent states in Q:

$$\forall q, q' \in Q : q \equiv q' \iff q = q'.$$

The result of applying the minimization procedure to the automaton in figure 1.22 is shown in figure 1.23.

If $q \equiv q'$, then the choice of the state to be eliminated is of course arbitrary. We adopt the convention of always keeping the state with the smaller index. As a result, each of the remaining states is the one with the smallest index in its respective equivalence class.

[9] A relation $\equiv$ on the Cartesian product $Q \times Q$ is called an equivalence relation if it is reflexive ($q \equiv q$), transitive (if $q \equiv q'$ and $q' \equiv q''$, then $q \equiv q''$) and symmetric (if $q \equiv q'$ then $q' \equiv q$).
[10] $A\backslash B$ denotes the *difference* of the sets A and B, e.g. $\{1, 2, 3, 4\}\backslash\{2, 4, 5\} = \{1, 3\}$.

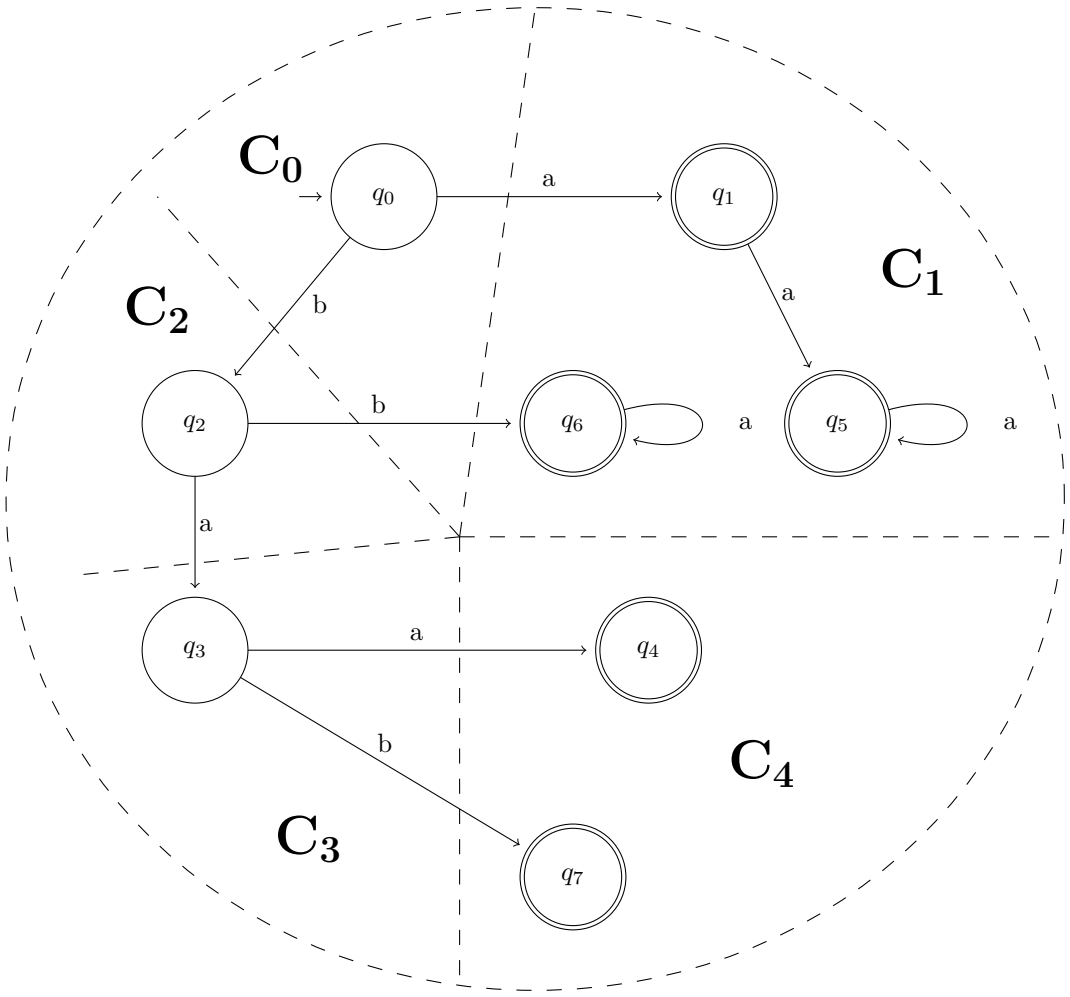


Figure 1.22: Equivalence classes defined on the stateset of a DFSA by $\equiv$.

The pseudocode for the generic minimization procedure is given in algorithm 1.7.1.

Algorithm 1.7.1: MINIMIZE($A = (\Sigma, Q, I, F, \delta)$)

$EqClass[] \leftarrow$ PARTITION(A)
$q_0 \leftarrow min(EqClass[q_0])$
for $\langle q, a, q' \rangle \in \delta$
  $\delta(q, a) \leftarrow min(EqClass[q'])$
for $q \in Q$
  if $q \neq min(EqClass[q])$
    $Q \leftarrow Q \backslash \{q\}$
    if $q \in F$
      $F \leftarrow F \backslash \{q\}$

The algorithm starts with a call to PARTITION(A). This procedure determines all equivalence classes of states with respect to the relation $\equiv$ and returns an array that associates each state q with its equivalence class ($EqClass[q]$). We assume that Q is a set of non-negative integers ($Q \subset \mathbb{N}_0$), and always keep the state $min(C)$ as a placeholder for class C.

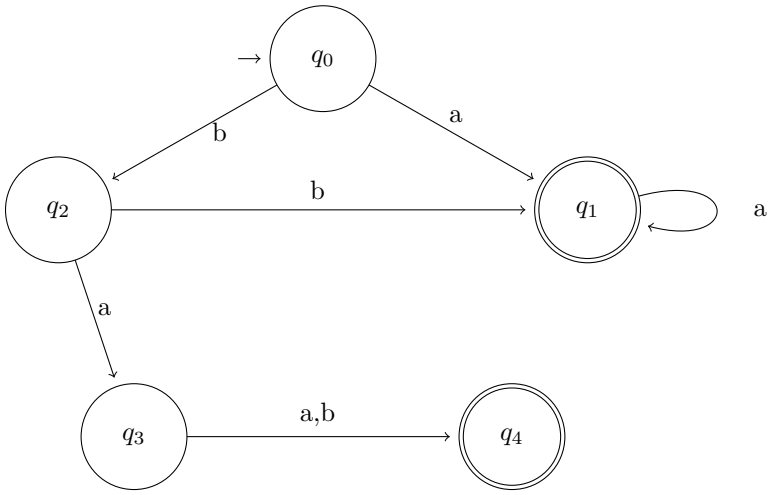


Figure 1.23: The result of applying minimization to the DFSA in figure 1.22.

In the next step, the algorithm redirects all transitions $\langle q, a, q' \rangle$ to point to $min(EqClass[q'])$ (the only state to be left in the equivalence class of q'). The last loop of the algorithm removes all redundant states from Q and F.

In this way, we have reduced the problem of minimizing a DFSA to the task of computing the equivalence relation $\equiv$ on the stateset. Thus, the exact implementation of PARTITION(A) is now the actual challenge.

One possible, although naïve, solution would be to check the equivalence of q and q' separately for every pair $q, q' \in Q$. If $A_r = (\Sigma, Q, r, F, \delta)$ denotes the automaton constructed out of $A = (\Sigma, Q, q_0, F, \delta)$ by making $r \in Q$ the initial state, then obviously $\overrightarrow{\mathcal{L}}(r) = \mathcal{L}(A_r)$, and hence $q \equiv q'$ if and only if $\mathcal{L}(A_q) = \mathcal{L}(A_{q'})$.

The pseudocode for the naïve minimization procedure is given in algorithm 1.7.2.

Algorithm 1.7.2: NAÏVEPARTITION$(A = (\Sigma, Q, I, F, \delta))$

for each $q \in Q$
  $EqClass[q] = \{q\}$
for each $q \in Q$
  for each $q' \in Q$
    if $EqClass[q] \neq EqClass[q']$ **and** CHECKEQUIVALENCE$(A_q, A_{q'}) = True$
      $EqClass[q] \leftarrow EqClass[q] \cup EqClass[q']$
      $EqClass[q'] \leftarrow EqClass[q]$

The algorithm uses an array $EqClass$ of pointers to disjoint sets to represent the sets of equivalence states. The sets are initialized to $\{q\}$ for each state q in the first **for each**-loop. Then, in the two nested loops, whenever we discover that $q \equiv q'$, the respective equivalence classes are merged, and the pointers $EqClass[q]$ and $EqClass[q']$ are both made point to the result.

As for the running time, both the outer and the inner loop of the algorithm execute $|Q|$ times. The running time of the procedure CHECKEQUIVALENCE(A, A'), which is invoked inside the inner loop, is $O(|Q|^2 \cdot |\Sigma|)$. As a result, the running time of the naïve minimization algorithm is $O(|Q|^4 \cdot |\Sigma|)$. In the next few subsections, we will see how we can improve on that.

1.7.1 A Dynamic Programming Solution

The main reason for the inefficiency of the naïve algorithm is that it traverses the whole automaton $A_q/A_{q'}$ each time it wants to determine if q and q' are equivalent, and then never re-uses the result.

However, results of previous equivalence checks are often significant and can save processing time. Consider the structure shown in figure 1.24.

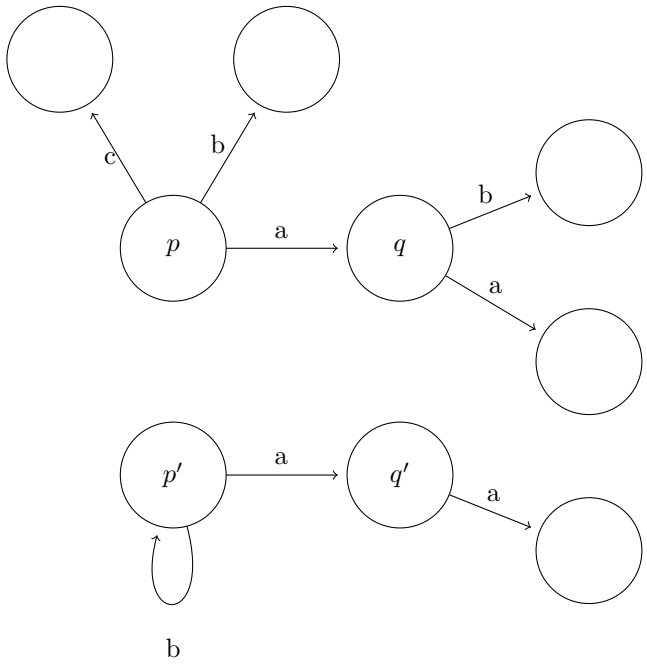


Figure 1.24: Fragment of a DFSA. If $q \not\equiv q'$ then $p \not\equiv p'$.

Suppose we have already checked the pair $\langle q, q' \rangle$ and established that $q \not\equiv q'$. If in some future step we are to check the equivalence of p and p', it is obviously not necessary to traverse the whole automaton. By inspecting the two transitions labeled a and leaving p and p', respectively, we can make an important observation.

Since $q \not\equiv q'$, we know that $\overrightarrow{\mathcal{L}}(q) \neq \overrightarrow{\mathcal{L}}(q')$, i.e. there exists a string u that is in one of the right languages $\overrightarrow{\mathcal{L}}(q)$, $\overrightarrow{\mathcal{L}}(q')$, but not in the other one. The problem is symmetric, so we may assume that $u \in \overrightarrow{\mathcal{L}}(q)$, and $u \notin \overrightarrow{\mathcal{L}}(q')$.

Since we can get from p to q consuming a, it follows that $au \in \overrightarrow{\mathcal{L}}(p)$. On the other hand, $au \notin \overrightarrow{\mathcal{L}}(p')$, because consuming the first character (a) gets us to state q' with the string u left, which cannot be accepted since $u \notin \overrightarrow{\mathcal{L}}(q')$. This in turn means $\overrightarrow{\mathcal{L}}(p) \neq \overrightarrow{\mathcal{L}}(p')$, i.e. $p \not\equiv p'$.

In more abstract terms, this means that we can *propagate* the non-equivalence property: whenever we find out that $q \not\equiv q'$ for some $q, q' \in Q$, we can infer that $p \not\equiv p'$ for all states p, p' such that $\delta(p, a) = q$ and $\delta(p', a) = q'$ for some symbol $a \in \Sigma$.

The propagation itself has to start somewhere, preferably where the non-equivalence $q \not\equiv q'$ is easy to establish *locally*, without looking at other states. This is possible when only one of the states q, q' is final.[11] Therefore, the propagation algorithms are initialized with the final-nonfinal partition of the stateset.

Another case of easily establishable non-equivalence is shown in figure 1.25 below. Both states q and q' are non-final, but the transition $\delta(q', a) = r$ is defined, while $\delta(q, a)$ is not. As a result, $\overrightarrow{\mathcal{L}}(q')$ does not contain any strings starting with a. On the other hand, such strings are present in $\overrightarrow{\mathcal{L}}(q)$ provided $\overrightarrow{\mathcal{L}}(r) \neq \emptyset$, which in turns means that at least one final state can be reached starting in r.

[11] Obviously, $\overrightarrow{\mathcal{L}}(q) \neq \overrightarrow{\mathcal{L}}(q')$ if $q \in F$ and $q' \notin F$, because $\epsilon \in \overrightarrow{\mathcal{L}}(q)$, but $\epsilon \notin \overrightarrow{\mathcal{L}}(q')$.

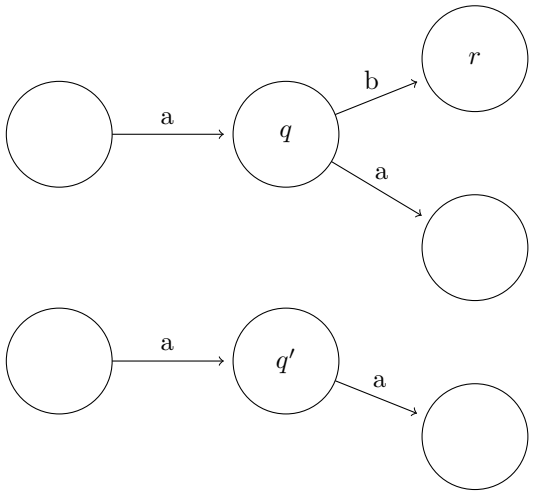


Figure 1.25: Fragment of a DFSA. The transition $\delta(q, a)$ is defined, while $\delta(q, a) = r$ is not. If the DFSA is trim, this means that $q \not\equiv q'$.

Checking if $\overrightarrow{\mathcal{L}}(r) = \emptyset$ is a non-local operation, but states r such that $\overrightarrow{\mathcal{L}}(r) = \emptyset$ are obviously redundant since reaching r means failure. As a result, all transitions entering such a state can be removed from the automaton without affecting its functionality. In a similar way, one can remove all states not reachable from an initial state — the automaton will never be able to enter them anyway.

Thus, a state can always be removed unless it belongs to a path for an initial to a final state. States that do belong to such paths are called *coaccessible*. The removal of non-coaccessible states and transitions is called *trimming*. An FSA is called *trim* if it does not contain non-coaccessible states.

In effect, the following two criteria can be used to establish the non-equivalence of two states q and q' in a trim DFSA:

- only one of q, q' is final;
- there is a symbol $a \in \Sigma$ such that $\delta(q, a)$ is defined, but $\delta(q', a)$ is not, or vice versa.

There exist several algorithms that propagate the non-equivalence information into the DFSA. In the following sections, we review the three most commonly used ones. Since the original versions of some of them were formulated for complete automata, we present a sightly modified version of them that have been adapted to automata with partial transition functions, which are — as already mentioned — much more common in programming practice.

Table Filling Algorithm (Hopcroft & Ullman)

This first procedure to be discussed here represents the equivalence relation as a $|Q| \times |Q|$ table $Equiv$. Each cell of the table can have one of the Boolean values $True$ (represented by the bit value 1) and $False$ (bit value 0).

The table is initially filled with 1's, while 0's are successively filled into the appropriate cells. Accordingly, the algorithm is sometimes referred to as the *table-filling* minimization method. The initial setting for the automaton in figure 1.22 is as shown in table 1.1.

Since the equivalence relation $\equiv$ is symmetric ($q \equiv q' \iff q' \equiv q$), there is no need to represent both $Equiv[q, q']$ and $Equiv[q', q]$ by separate cells. Therefore, each cell $Equiv[q, q']$ is assumed to be for $q < q'$. This reduces the space required by the algorithm by more than 50%.

At each intermediate step, $Equiv[q, q'] = 0$ means that *we already know* the states q and q' are not equivalent. On the other hand, $Equiv[q, q'] = 1$ means that so far, we haven't seen anything

state	q_0	q_1	q_2	q_3	q_4	q_5	q_6
q_7	1	1	1	1	1	1	1
q_6	1	1	1	1	1	1	
q_5	1	1	1	1	1		
q_4	1	1	1	1			
q_3	1	1	1				
q_2	1	1					
q_1	1						

Table 1.1: Initial setting of the table filling algorithm.

indicating that q and q' are not equivalent. When the algorithm terminates, the remaining cells filled with 1's indicate the actual equivalencies.

The algorithm is initialized by finding all pairs of states $\langle q, q' \rangle$ for which the non-equivalence $q \not\equiv q'$ can be established locally using the two criteria derived in section 1.7.1: either only one of the two states is final, or only one of $\delta(q, a)$ and $\delta(q', a)$ is defined for some $a \in \Sigma$. In order for the second criterion to be correct, the DFSA is required to be trim. Procedure LocalEquivalenceCheck(q, q') implements this check.

Algorithm 1.7.3: LocalEquivalenceCheck(q, q')

if ($q \in F$ and $q' \notin F$) or ($q \notin F$ and $q' \in F$)
  return $(False)$
if $\exists a \in \Sigma$ such that only one of $\delta(q, a), \delta(q', a)$ is defined
  return $(False)$
return $(True)$

Whenever this happens, we set $Equiv[q, q'] = 0$, and know that this new piece of information might be useful in inferring the non-equivalence of some other states. Thus, non-equivalence information is propagated *backwards* by calling procedure Propagate(p, p'). For each symbol $a \in \Sigma$, the procedure traverses all transitions $\langle p, a, q \rangle$, $\langle p', a, q' \rangle$ and, unless it has already been done, it sets $Equiv[p, p']$ to $False$, and invokes itself (Propagate(p, p')) recursively in order to propagate the non-equivalence information further.

The symbol $\delta_a^{-1}(q)$ denotes the set of all states p such that there is a transition from p to q labeled with symbol a.

Algorithm 1.7.4: Propagate(q, q')

for $a \in \Sigma$
  for $p \in \delta_a^{-1}(q)$
    for $p' \in \delta_a^{-1}(q')$
      if $Equiv[min(p, p'), max(p, p')] = 1$
       $Equiv[min(p, p'), max(p, p')] \leftarrow 0$
       Propagate(p, p')

The order in which the pairs q, q' are tested for equivalence is arbitrary. Let us assume that the pair q_4, q_5 is picked first. Both states are final, but $\delta(q_4, a)$ is defined while $\delta(q_5, a)$ is not. Thus, we set $Equiv[q_4, q_5]$ to 0 and invoke Propagate(q_4, q_5).

The procedure discovers that q_4 and q_5 can be reached from q_3 and q_1, respectively, via the symbol a. Thus, $q_1 \not\equiv q_3$, and hence $Equiv[q_1, q_3] \leftarrow 0$. Another call to Propagate() follows, this time with the parameters q_1 and q_3, establishing that $q_0 \not\equiv q_2$. The subsequent call to Propagate(q_0, q_2) does not lead to any further propagation of the available information because there are no transitions entering q_0 (recall that non-equivalence is propagated backwards!).

Table 1.2 shows the state of the equivalence table after this step.

state	q_0	q_1	q_2	q_3	q_4	q_5	q_6
q_7	1	1	1	1	1	1	1
q_6	1	1	1	1	1	1	
q_5	1	1	1	1	0		
q_4	1	1	1	1			
q_3	1	0	1				
q_2	0	1					
q_1	1						

Table 1.2: State equivalence table after the first call to PROPAGATE() in the main loop.

The reader may check that the algorithm eventually arrives at the following values for the cells of the table, which encode the partition shown in figure 1.22.

state	q_0	q_1	q_2	q_3	q_4	q_5	q_6
q_7	0	0	0	0	1	0	0
q_6	0	1	0	0	0	0	
q_5	0	1	0	0	0		
q_4	0	0	0	0			
q_3	0	0	0				
q_2	0	0					
q_1	0						

Table 1.3: Final setting of the table filling algorithm.

The main loop of the algorithm is shown in algorithm 1.7.5 below.

Algorithm 1.7.5: TABLEFILLINGPARTITION($A = (\Sigma, Q, I, F, \delta)$)

```
for q, q' ∈ Q, q < q'
  Equiv[q, q'] ← 1
for q ∈ Q
  for q' ∈ Q, q < q'
    if Equiv[q, q'] = 1 and LocalEquivalenceCheck(q, q') = False
      Equiv[q, q'] ← 0
      Propagate(q, q')
```

Note that the procedure PROPAGATE(q, q') is never called twice for one pair of states because the program checks if $Equiv[q, q']$ is already set to $False$ before calling it. Therefore, the running time of the algorithm is bounded by the number of possible state pairs ($|Q|^2$) times the size of the alphabet: $O(|Q|^2 \cdot |\Sigma|)$. This is a significant improvement over the $O(|Q|^4 \cdot |\Sigma|)$ running time of the naïve algorithm.

The main disadvantage of the table filling algorithm is due to its high space requirements. The table $Equiv[]$ needs cells in an order of magnitude of $|Q|^2$, which is unfeasible for large statesets (even if we only need one bit per pair). Note that this is not a *worst-case* estimate, such as the $O(2^{|Q|})$ upper bound on the size of the deterministic automaton in subset construction (algorithm 1.4.1). A $|Q| \times |Q|$ table must be preallocated for any input automaton. Thus, $|Q|^2$ is a *tight upper and lower bound* on the space requirements of the table-filling algorithm. Formally, this is abbreviated $\Theta(|Q|^2)$ (see also the frame on page 38).

Θ and Ω -Notation

With the O-notation introduced earlier in this book we can express the worst-case time or space complexity of algorithms and data structures, i.e., when we say that a complexity of some algorithm or data structure is $O(f(n))$ we merely state that for all $n > n_0$ this complexity is bounded from above by $c \cdot f(n)$ for some constant c. However, this does not say anything about the best case, for which a more favorable asymptotic approximation may exist.

In other cases, we can estimate that a runtime or space requirement is exactly and tightly bound to a certain function for all or almost all values of n (so to speak, best case = worst case). For this purpose one can use the so called Θ-*notation*. Formally speaking, $\Theta(f(n))$ denotes the set of all functions which grow exactly as $f(n)$, i.e., $g(n) \in \Theta(f(n))$ if there exist constants c_1 and c_2 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for $n > n_0$ for some n_0. Formally $f(n)$ is called an *asymptotic tight bound* for $g(n)$. So when we say that a complexity is $\Theta(f(n))$ we express the fact that for each value of n the running time oscillates around the value of $f(n)$.

Additionally, Ω-*notation* is used for expressing *asymptotic lower bounds*, i.e., $\Omega(f(n))$ denotes the set of all functions which grow at least as fast as $f(n)$. The Ω-notation can be used for expressing the best-case running time, or lower bounds on arbitrary inputs. When we say that the running time of an algorithm is $\Omega(f(n))$, we mean that there exists a non-negative constant c such that the running time is at least $c \cdot f(n)$ for all inputs of size n.

Aho, Sethi & Ullman

In the table-filling algorithm, information about the partitioning of Q according to the relation $\equiv$ is split into singular statements of the form $q \equiv q' / q \not\equiv q'$, each filling one cell in the table. For a stateset of size n, we therefore need $n(n-1)/2 = \Theta(n^2)$ cells.

A more space-efficient representation is to associate each state with the ID of the *block* it belongs to. The *blocks* are equivalence classes in the making: the elements of the partition of Q that is defined by the $q \not\equiv q'$ information propagated so far. Obviously, this representation requires only $\Theta(|Q|)$ space.

As in the table-filling algorithm, we initialize the partition with the blocks F and $Q \backslash F$, and then successively propagate the available non-equivalence information. This time, propagation means successively splitting existing blocks. Technically, this process is referred to as *refining* an equivalence relation.

In each iteration, the algorithm selects a symbol $a \in \Sigma$ and inspects the blocks to which the source and target states of all transitions labeled a belong. For each block B on the source side, two cases may occur:

Case 1: If the transitions end up in different blocks, the current block is split accordingly, as shown in figure 1.26. Note that this split may induce the split of some block C (marked in grey) in the next step.

Case 2: If all the transitions end up in the same block, the block is kept (figure 1.27).

Since each split may induce further splits (as in figure 1.26, where the splitting of B causes C to be split), repeating the above procedure makes sure the non-equivalence information is propagated.

Note that each block B is split according to the blocks $B^1 \dots B^m$ reachable from B via the current symbol a: all q such that $\delta(q, a) \in B^i$ end up in the same block after the split. Therefore, the algorithm for splitting the blocks can be stated informally as follows:

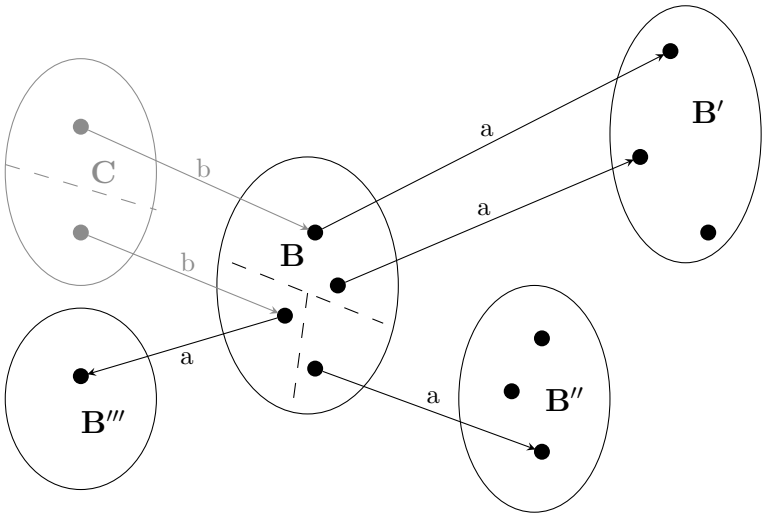


Figure 1.26: The block B is split because transitions leaving B and labeled a end up in three different blocks: B', B'' and B'''. The split (marked by the dashed lines) reflects the blocks of the target states. Note that this split of B will also cause the block C to be split when the algorithm inspects transitions labeled b and leaving C.

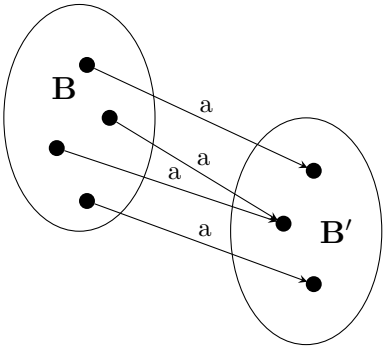


Figure 1.27: Block B is not split because all transitions leaving B and labeled a end up in one block.

Algorithm 1.7.6: AhoSethiUllmanPartition$(\Sigma, Q, I, F, \delta)$

Split Q into blocks F and $Q \backslash F$
repeat
 for each block B
  for each $a \in \Sigma$
   Identify blocks $B^1, \ldots, B^m$ reachable from B via a
   if $m > 1$
    for $i = 1$ **to** m
     Put all q such that $\delta(q, a) \in B^i$ into a new block $\hat{B}^i$
until No more blocks can be split

The basic principle being clear, the actual challenge is to select data structures that would allow us to implement the procedure efficiently.

The assignment of states to blocks can be represented by a vector $Block[]$ of $|Q|$ integers, each one being a block ID. Since each block is a non-empty subset of Q, we can represent it unambiguously by the smallest state index occurring in it. As a convention, we will subsript the block symbol B with the index $i = min(\{j : q_j \in B_i\})$. For example, if $\{q_4, q_8, q_{11}\}$ is a block, then it is denoted B_4.

We can also assume that the state indices are the first $|Q| - 1$ non-negative integers: $Q =$

$\{q_i : i = 0, 1, \ldots n-1\}$ for some $n \in \mathbb{N}_0$. The initial split into blocks is done by the procedure InitializeEqClasses() (algorithm 1.7.7).

Algorithm 1.7.7: InitializeEqClasses($Block, (\Sigma, Q, I, F, \delta)$)

$fin \leftarrow min(\{i : q_i \in F\})$
$nonfin \leftarrow min(\{i : q_i \notin F\})$
for $q \in F$
  $Block[q] \leftarrow B_{fin}$
for $q \in Q \backslash F$
  $Block[q] \leftarrow B_{nonfin}$

The procedure starts by determining fin and $nonfin$, the smallest index of a final and a non-final state respectively. These indices are the used as identifiers of the initial two blocks: B_{fin} containing all final states, and B_{nonfin} all non-final states. These block IDs are then assigned to the states. The result for the DFSA in figure 1.22 in shown in the following table:

state	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7
block	B_0	B_1	B_0	B_0	B_1	B_1	B_1	B_1

Table 1.4: The initial split of states into two blocks (final/non-final states) for the DFSA in figure 1.22.

The equivalence classes are $B_0 = Q \backslash F = \{q_0, q_2, q_3\}$ and $B_2 = F = \{q_1, q_4, q_5, q_6, q_7\}$.

In the first refinement step, we inspect transitions leaving block B_0 and labeled a. We discover that $\delta(q_0, a) = q_1$ and $\delta(q_3, a) = q_4$ are in block B_1 while $\delta(q_2, a) = q_3$ belongs to B_0. Therefore, q_2 is not equivalent to any of the remaining states in B_0, so we need to split block B_0 into $\{q_0, q_3\}$ (denoted B_0) and $\{q_2\}$ (denoted B_2). The reader may repeat this procedure for the other symbols until no changes occur and check that the resulting partition is indeed the one shown in diagram 1.22.

The refinement of the blocks of Q is best implemented in the following way. Let $a \in \Sigma$ be the current alphabet symbol. For each state q in the $Block[]$ vector, we determine $\delta(q, a)$, i.e. the state that can be reached from q via a. If $\delta(q, a)$ is undefined, we set $\delta(q, a) = \perp$, where $\perp$ may be viewed as a special rejecting *dead state* such that $\delta(\perp, \alpha) = \perp$ for all $\alpha \in \Sigma$).

The values of $Block[q]$, $\delta(q, a)$ and $Block[\delta(q, a)]$ for the first refinement step are given in the rows 2, 3 and 4 of table 1.5.

state (q)	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7
$Block[q]$	B_0	B_1	B_0	B_0	B_1	B_1	B_1	B_1
$\delta(q, a)$	q_1	q_5	q_3	q_4	$\perp$	q_5	q_6	$\perp$
$Block[\delta(q, a)]$	B_1	B_1	B_0	B_1	$B_\perp$	B_1	B_1	$B_\perp$
block after refinement	B_0	B_1	B_2	B_0	$B_\perp$	B_1	B_1	$B_\perp$

Table 1.5: The refinement operation in the Aho-Sethi-Ullman algorithm.

As pointed out above, a block B of Q is left unchanged only if all transitions labeled a leaving states in B lead to the same block B'. Accordingly, if the transitions point to different blocks, B is split in order to reflect the differences. The last row in table 1.5 shows the result of such a split.

This means that each distinct pair $\langle Block[q], Block[\delta(q, a)]\rangle$ identifies a separate block *after* the refinement step for the current symbol. For example, in table 1.5, the pair $\langle B_1, B_\perp\rangle$ occurs in the columns corresponding to the states q_4 and q_7, and thus stands for the block $B_4 = \{q_4, q_7\}$.

This can be implemented efficiently by mapping each occurring pair $\langle B, B'\rangle$ from the second and fourth row of the table to the value $Map[\langle B, B'\rangle] \leftarrow q_i$, where q_i is the first state (in ascending order) for which the pair $\langle B, B'\rangle$ occurs in the table.

The new value of $Block[q]$ is then set to $Map[\langle B, B' \rangle]$, where B and B' are the values in, respectively, the second and the fourth row of the table for q.

The pseudocode of the refinement procedure is given below in algorithm 1.7.8. Recall that we make the assumption that the state indices i are chosen from the set $\{0, 1, \ldots |Q| - 1\}$, so we can iterate through all $q_i \in Q$ for $i = 0, \ldots |Q| - 1$.

The procedure computes the new block IDs for the states and writes them back into the vector $Block[]$. The Boolean flag $Changed$ is initially set to $False$. The algorithm sets it to $True$ as soon as the block of one of the states changes (i.e., the original block is refined). The final value of this flag is also the return value of the function. Thus, REFINE() returns $True$ if the input partition has actually been refined, and $False$ if it stays unchanged.

Algorithm 1.7.8: REFINE($Block, a$)

```
Changed ← False
for i ← 0 to |Q| − 1
   if ⟨Block[q_i], Block[δ(q_i, a)]⟩ ∉ Map
      Map(⟨Block[q_i], Block[δ(q_i, a)]⟩) ← B_i
   NewBlock ← Map(⟨Block[q_i], Block[δ(q_i, a)]⟩)
   if NewBlock ≠ Block[q_i]
      Changed ← True
      Block[q_i] ← NewBlock
return (Changed)
```

The pseudocode for the partitioning of Q into equivalence classes is given in algorithm 1.7.9.

Algorithm 1.7.9: AHOSETHIULLMANPARTITION(Σ, Q, I, F, δ)

```
INITIALIZEEQCLASSES(A, Block)
Changed ← true
while Changed = true
   Changed ← false
   for a ∈ Σ
      if REFINE(Block, a) = true
         Changed ← true
```

The algorithm starts by calling the procedure INITIALIZEEQCLASSES() (algorithm 1.7.7), which performs the initial split of Q into two blocks: final and non-final states.

Then, the **while** loop performs the actual refinement of the blocks by calling REFINE($Block, a$) for all symbols $a \in \Sigma$. The variable *Changed*, initialized to **true** before entering the loop, indicates whether or not at least one block has been changed in the current iteration. If the value of *Changed* is **false** after an iteration, it means that we have tried to refine the current partition using each alphabet symbol, but failed. Thus, another execution of the loop will not change anything: the current partition expresses the final partition, i.e. the equivalence classes with respect to the relation $\equiv$. Accordingly, the **while** loop terminates.

The space requirements of the Aho-Sethi-Ullman algorithm are linear in the size of the stateset: if $|Q| = n$, then we need a vector of n integers: $Block[]$. In asymptotic terms, we thus need $\Theta(n)$ memory, which is a significant improvement over the $\Theta(n^2)$ table-filling algorithm.

The running time of the algorithm is bounded by $O(|\Sigma| \cdot |Q|^2)$. The first (initializing) loop obviously runs in time $\Theta(|Q|)$. The second (**while**) loop, in which non-equivalence information is propagated, executes at most $|Q|$ times. In order to see this, observe that the number of distinct blocks must increase between subsequent executions of the loop (otherwise, the blocks are not changed and the algorithm terminates). There cannot be more blocks than states (each state must belong to exactly one block). Thus, at latest after $|Q| - 1$ iterations no more block splitting is possible and the algorithm terminates.

Each iteration of the **while** loop contains a loop iterating over all alphabet symbols $a \in \Sigma$, which in turn calls the procedure REFINE(), whose running time is $O(|Q|)$. By multiplying the number of executions of the two nested loops and REFINE(), we indeed arrive at $O(|\Sigma| \cdot |Q|^2)$, the same as the asymptotic complexity of the table-filling algorithm. In the next section, we shall see that we can do better than that.

Hopcroft

The Aho-Sethi-Ullman way of partitioning Q into equivalence classes improves over the table-filling algorithm in terms of space requirements ($\Theta(|Q|)$ as opposed to $\Theta(|Q|^2)$), but the running time of both algorithms is identical: $O(|Q|^2)$, which may be a problem for large statesets.

In order to improve on that, let us focus on the mechanism used to propagate non-equivalence information in Q. There is some perceived inefficiency in the propagation step: in every call to the procedure REFINE() in algorithm 1.7.9, we iterate through all blocks of the DFSA although typically only a few of them trigger the split of a block.

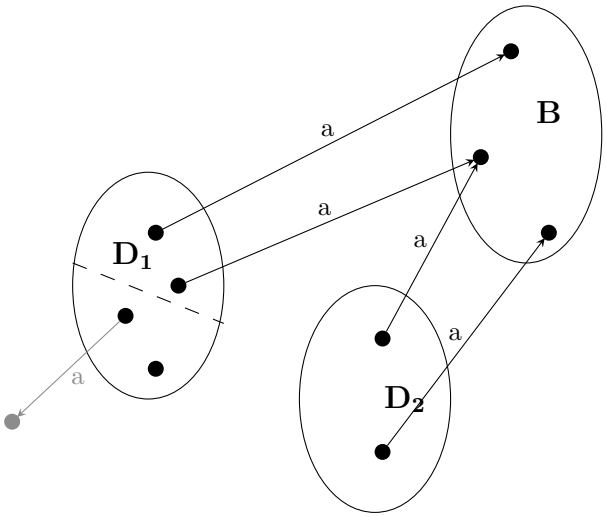
It turns out that a better strategy is possible. We can explicitly distinguish between two types of blocks:

Active blocks: an active block has the potential of refining the current partition of Q by inducing the split of one of the existing blocks.

Inactive blocks: an inactive block does not split any of the blocks in the current partition.

The idea is to keep track of which block is active and which is not, and only consider the active blocks for propagating non-equivalence information.

This time, propagation is done in a slightly different way. We select a single block B and a symbol $a \in \Sigma$, and focus on the blocks in which the *incoming* transitions labeled a originate. The following figure exemplifies the two situations that may occur.



Here, transitions labeled a and entering block B originate in two blocks: D_1 and D_2. The difference between them is as follows.

- Not all of the states in D_1 are the source of a transition labeled a and entering block B. Thus, they may not be equivalent, and D_1 is split in order to reflect this difference (the split is indicated by the dashed line). In such a situation, we say that B *splits* D_1 *by symbol* a.

  As in the previous algorithms, these newly split blocks may induce further block splits in subsequent iterations.

- All transitions starting in D_2 and labeled a enter B. Therefore, D_2 is not split.

Note that, after inspecting all blocks D for all symbols $a \in \Sigma$, the "splitting potential" of block B has expired as each block D left after this step has the property that, for each $s \in \Sigma$, either

$$\forall q \in D : \delta(q, s) \in B$$

or

$$\forall q \in D : \delta(q, s) \notin B.$$

As a result, B can be marked as inactive.

The algorithm relies on there being an active block C inducing the split of some block B, active or inactive, into two blocks B' and B''. Since the distinction between active and inactive blocks is crucial for the algorithm, the question arises how B' and B'' should be marked with respect to the active-inactive property.

B is inactive. Some of the existing blocks D for which $\forall q \in D : \delta(q, s) \in B$ (for some $s \in \Sigma$) may become splittable, as illustrated by figure 1.28.

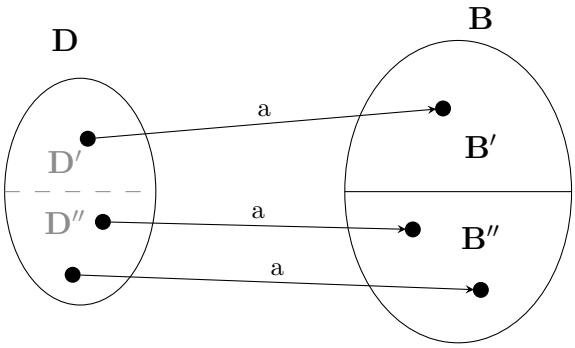


Figure 1.28: The split of block B into B' and B'' induces a split of block D.

Here, there is no trigger for splitting D as long as B remains in one piece. However, as soon as B is split into B' and B'', D must be split into D' and D'' in order to reflect the difference.

Also observe that the splitting of D is symmetric: it does not matter whether it is triggered by B' or B''. As long as B does not split D by any symbol $s \in \Sigma$ (which is guaranteed by the fact that B has been processed previously), the split in D induced by B' is identical to the split in D induced by $B'' = B \backslash B'$. As a practical consequence, there is no need to propagate non-equivalence information from both B' and B''. Hence, one of them needs to be marked as active. As we shall soon see, it is preferable to activate the smaller block for efficiency reasons.

B is active. In this case, B may have the potential to split some block D by some symbol b, as illustrated by figure 1.29.

However, it may happen that B is split into B' and B'' *before* the pair (B, b) is due to be considered. This split may induce a split of D into D' and D'', as shown in figure 1.30.

The split in D induced by splitting B into B' and B'' is orthogonal to the split in D that could have been induced by B and b. As a result, B' splits D' (but not D'') and B'' splits D'' (but not D'). Thus, both B' and B'' must be marked as *active* on their creation.

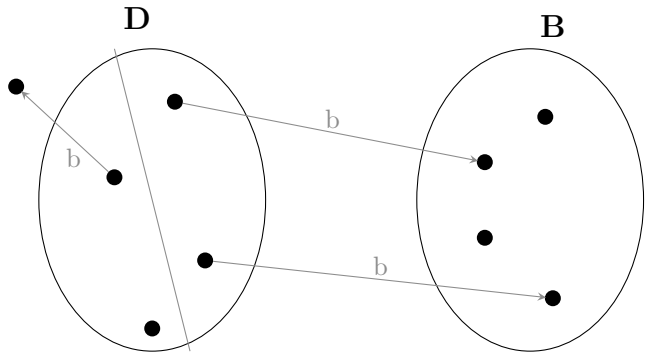


Figure 1.29: The active block B has the potential to split block D.

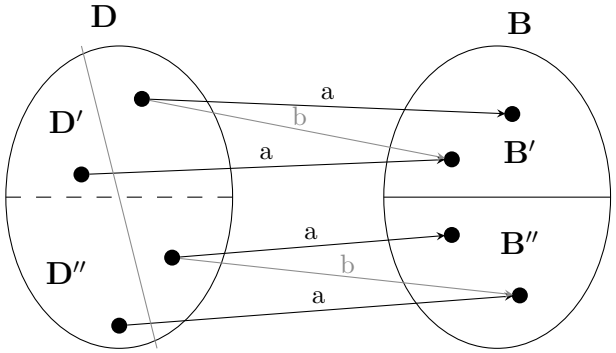


Figure 1.30: The split of an active block B into B' and B'' induces a split of block D into D' and D''. Note that B' splits D' and B'' splits D'' by symbol b.

This strategy can be stated as the following (informal) algorithm (recall that $\delta_s^{-1}(B)$ is the set of all states $q \in Q$ such that $\delta(q, s) \in B$).

Algorithm 1.7.10: HopcroftPartition$(\Sigma, Q, I, F, \delta)$

Initialize the partition with the blocks F and $Q \backslash F$
Mark both initial blocks as active
while there exists an active block B
  Inactivate B
  for $s \in \Sigma$
    $C \leftarrow \delta_s^{-1}(B)$
    for each block D split by B
      $D' \leftarrow D \cap C$
      $D'' \leftarrow D \backslash C$
      replace D by D', D''
      if D is active
        activate D' and D''
      else
        activate the smaller one of D' and D''

For an efficient implementation of the above algorithm, it is important that each iteration of the main nested loop is done in time linear in $|C|$. In order to achieve that, we can represent each block D as an aggregation of three data structures:

- the doubly linked list *Elements*, in which the elements of D are stored;
- the Boolean field *Active* which tells whether or not the block is active;
- the auxiliary doubly linked list *Intersection*, which is used to store the elements of the intersection of D and the currently processed block C.

In addition, we maintain two vectors $BlockId[]$ and $Link[]$, both indexed by the states $q \in Q$. $BlockId[q]$ is a pointer to the block q belongs to. $Link[q]$ is a pointer to the actual entry for q in the doubly linked list representing the block.

The following diagram shows the structure of an active block $D = \{q_1, q_4, q_5, q_8\}$.

$Active$	$=$	$True$
$Elements$	$=$	$\langle q_1, q_4, q_5, q_8 \rangle$
$Intersection$	$=$	NIL

Now suppose $C \cap D = \{q_4, q_8\}$. When the algorithm iterates through C, it finds both these states and moves the corresponding list entries from *Elements* to *Intersection*, resulting in:

$Active$	$=$	$True$
$Elements$	$=$	$\langle q_1, q_5 \rangle$
$Intersection$	$=$	$\langle q_4, q_8 \rangle$

Note that the pointers $Link[q]$ make it possible to move each of the the linked list entries in constant time.

As a result, *Intersection* holds the intersection $D \cap C$, while *Elements* holds the set difference $D \backslash C$. These two lists correspond to the values of the blocks D' and D'' into which D is split. Therefore, we now create a new block and make the *Intersection* list of the old one the value of its *Element* field. Since the states in this list are moved to a new block, we make $BlockId[q]$ to the new block for each of these states q. The *Intersection* field on both blocks is set to NIL. Since the original block was active, both blocks are activated.

In the end, we are left with the following two structures:

$Active$	$=$	$True$
$Elements$	$=$	$\langle q_1, q_5 \rangle$
$Intersection$	$=$	NIL

and

$Active$	$=$	$True$
$Elements$	$=$	$\langle q_4, q_8 \rangle$
$Intersection$	$=$	NIL

The vector $BlockId[]$ is updated accordingly.

Note that it might happen that $D \cap C = D$, i.e. $D \backslash C = \emptyset$ and D is not split. In such a case, we end up with the following structure:

$Active$	$=$	$True$
$Elements$	$=$	NIL
$Intersection$	$=$	$\langle \ldots \rangle$

In such a case, we just swap *Elements* and *Intersection*. This restores the original representation of D (no new block is created, of course). The pseudocode of the block refinement procedure is given below:

Algorithm 1.7.11: HOPCROFTREFINE(C)

```
for each q ∈ C
  MOVE(Intersection[BlockId[q]], Link[q])
for each q ∈ C
  if Intersection[BlockId[q]] ≠ NIL
    SPLITBLOCK(BlockId[q])

procedure SPLITBLOCK(D)
 if Elements[D] ≠ NIL
    D' ← create a new block
    Elements[D'] ← Intersection[D]
    Intersection[D'] ← NIL
    Intersection[D] ← NIL
    for each q ∈ Elements[D']
      BlockId[q] ← D'
    if Active[D]
       Active[D'] ← true
     else
       if length[Elements[D']] < length[Elements[D]]
         Active[D'] ← true
       Active[D] ← false
        else
          Active[D'] ← false
  else
    Block[D] ← Intersection[D]
```

The first **for each** loop moves each $q \in C$ from the *Elements* list to the *Intersection* list in the block to which q belongs.

The second **for each** loop accesses each of the modified blocks, checks if the intersection contains any elements and passes such a block to the procedure SPLITBLOCK(), which splits it into two if required (i.e., whenever both lists *Elements* and *Intersection* are non-empty). Note that several states $q \in C$ may point to the same block D. However, after the first call to SPLITBLOCK(D), its intersection list is set to NIL, and therefore SPLITBLOCK(D) is never invoked again.

The state partitioning algorithm can now be restated more formally as follows:

Algorithm 1.7.12: HOPCROFTPARTITION(Σ, Q, I, F, δ)

```
Elements[B_fin] ← F
Elements[B_nonfin] ← Q − F
Active[B_fin] ← true
Active[B_nonfin] ← true
Intersection[B_fin] ← NIL
Intersection[B_nonfin] ← NIL
while there exists an active block B
  Inactivate B
  for s ∈ Σ
    C ← δ_s^{-1}(B)
    HOPCROFTREFINE(C)
```

The running time of the algorithm is $O(|\Sigma| \cdot |Q| \cdot log|Q|)$. To see this, consider the following. Let $B^{(i)}$ be the block considered in the i-th iteration of the **while** loop. Let $C^{(i,s)}$ denote $\delta_s^{-1}(B^{(i)})$. The running time of HOPCROFTREFINE(C) is linear in C. The running time of the entire loop is thus proportional to $\Sigma_{i=1}^{K} \Sigma_{s \in \Sigma} |C^{(i,s)}|$, where K is the total number of iterations. But:

$$\begin{aligned}\Sigma_{i=1}^{K}\Sigma_{s\in\Sigma}|C^{(i,s)}| &\leq \Sigma_{i=1}^{K}\Sigma_{q\in B^{(i)}}\Sigma_{s\in\Sigma}|\delta_s^{-1}(q)| = \Sigma_{i=1}^{K}|B^{(i)}|\cdot|\Sigma|\cdot|Q| \\ &\leq \Sigma_{q\in Q}f(q)\cdot|\Sigma|\cdot|Q| \qquad (1.6)\end{aligned}$$

where $f(q)$ is the number of times state q appears in the current block $B^{(i)}$.

On the other hand, we can derive a bound on $f(q)$ in the following way. The first time $Block[q]$ is selected in the **while** loop, the size of $Block[q]$ is bounded by the size of Q only. Then, $Block[q]$ is deactivated and remains so until it is split. After the split, the new $Block[q]$ can only be active if it is the *smaller* of the two new blocks created by splitting the old value. This means that $|Block[q]| < \frac{|Q|}{2}$. Accordingly, every time $Block[q]$ is selected in the **while** loop, its size is halved. As a result, after being selected $log|Q|$ times at latest, $Block[q]$ becomes a singleton set: $Block[q] = \{q\}$.

If such a singleton is still active, it may be selected at most once. Then it is inactivated and cannot be re-activated anymore because activation only happens when a block is split, and singleton blocks are unsplittable. Thus, $f(q) \leq log|Q| + 1$. Together with (1.6), this yields:

$$\Sigma_{i=1}^{K}\Sigma_{s\in\Sigma}C^{(i,s)} \leq (log|Q| + 1)\cdot|\Sigma|\cdot|Q|$$

Thus, the running time of Hopcroft's algorithm is $O(|\Sigma|\cdot|Q|\cdot log|Q|)$. This is the best known DFSA minimization algorithm for the general case.

1.7.2 Brzozowski's Algorithm

All the minimization algorithms presented so far are based on partitioning the stateset of a DFSA into equivalence classes with respect to the state equivalence relation $\equiv$. The last algorithm to be presented in this section works differently.

Its application is very simple. Given a DFSA $A = (\Sigma, Q, q_0, F, \delta)$, the algorithm consists of four consecutive steps: reversal, determinization, another reversal, and another determinization.

- First, A is reversed, creating a non-deterministic automaton $A^{-1} = (\Sigma, Q, F, \{q_0\}, \delta^{-1})$. Note that F is the set of *initial* states of A^{-1}, while q_0 is its only final state. The transition function δ^{-1} is defined as $\delta^{-1}(q, a) = \{q' \in Q : \delta(q, a) = q'\}$.

  Note that $\mathcal{L}(A^{-1}) = \mathcal{L}(A)^{-1}$: A^{-1} accepts the strings accepted by A, but in reverse order. For example, if A accepts *aaabb*, then A^{-1} accepts *bbaaa*.

- The reversed automaton A^{-1} is determinized, creating a new DFSA $Det(A^{-1})$. Obviously, $\mathcal{L}(Det(A^{-1})) = \mathcal{L}(A^{-1}) = \mathcal{L}(A)^{-1}$.

- $Det(A^{-1})$ is reversed, which creates another NFSA $Det(A^{-1})^{-1}$. The language accepted by the NFSA is the reverse of $\mathcal{L}(Det(A^{-1})) = \mathcal{L}(A)^{-1}$: $\mathcal{L}(Det(A^{-1})^{-1}) = (\mathcal{L}(A)^{-1})^{-1} = \mathcal{L}(A)$. I.e., the NFSA is equivalent to the original automaton A.

- The NFSA $Det(A^{-1})^{-1}$ is determinized, creating a DFSA $Det(Det(A^{-1})^{-1})$. Obviously, $\mathcal{L}(Det(Det(A^{-1})^{-1})) = \mathcal{L}(A)$.

In order to see that the result is minimal, consider the right languages of two states q, q' in the NFSA $Det(A^{-1})^{-1}$ created in the third step by reversing the DFSA $Det(A^{-1})$. It is easy to observe that $\overrightarrow{\mathcal{L}}(q) \cap \overrightarrow{\mathcal{L}}(q') = \emptyset$ if $q \neq q'$. Otherwise, there would be a string $w \in \Sigma^*$ such that $w \in \overrightarrow{\mathcal{L}}(q)$ and $w \in \overrightarrow{\mathcal{L}}(q')$. But then w would lead to two distinct states in $Det(A^{-1})$ (starting in the initial state of $Det(A^{-1})$), which would contradict the determinicity of $Det(A^{-1})$.

The subset construction algorithm used to build the final result $Det(Det(A^{-1})^{-1})$ creates states that are subsets of the stateset of the NFSA $Det(A^{-1})^{-1}$. Consider two states $\hat{q} = \{q_1, \ldots q_k\}$ and $\hat{r} = \{r_1 \ldots r_l\}$. Obviously, $\overrightarrow{\mathcal{L}}(\hat{q}) = \overrightarrow{\mathcal{L}}(q_1) \cup \ldots \cup \overrightarrow{\mathcal{L}}(q_k)$ and $\overrightarrow{\mathcal{L}}(\hat{r}) = \overrightarrow{\mathcal{L}}(r_1) \cup \ldots \cup \overrightarrow{\mathcal{L}}(r_l)$. According to the property stated in the previous paragraph, the languages $\overrightarrow{\mathcal{L}}(q_1), \ldots, \overrightarrow{\mathcal{L}}(q_k), \overrightarrow{\mathcal{L}}(r_1), \ldots, \overrightarrow{\mathcal{L}}(r_l)$

are pairwise disjoint with the exception of pairs $\overrightarrow{\mathcal{L}}(q_i), \overrightarrow{\mathcal{L}}(r_j)$ such that $q_i = r_j$ for some i, j. Thus, $\overrightarrow{\mathcal{L}}(\hat{q}) = \overrightarrow{\mathcal{L}}(\hat{r})$ if and only if $k = l$ and each q_i is identical to some r_j. But this is equivalent to $\{q_1, \ldots, q_k\} = \{r_1, \ldots, r_l\}$, i.e. $\hat{q} = \hat{r}$. Formally:

$$\overrightarrow{\mathcal{L}}(\hat{q}) = \overrightarrow{\mathcal{L}}(\hat{r}) \iff \hat{q} = \hat{r}.$$

According to the criterion stated in proposition 1, this proves that the DFSA is minimal.

1.7.3 Comparison of Minimization Algorithms

Table 1.6 compares the running time and the memory requirements of the minimization algorithms introduced in this chapter. The size of the alphabet ($|\Sigma|$), which is a constant factor in all the complexity bounds derived so far, is omitted for better readability, leaving the size of the stateset ($|Q|$) as the only parameter.

Method	running time	space requirements
Hopcroft & Ullman	$\Theta(\|Q\|^2)$	$\Theta(\|Q\|^2)$
Aho, Sethi & Ullman	$O(\|Q\|^2)$	$\Theta(\|Q\|)$
Hopcroft	$O(\|Q\| \cdot lg\|Q\|)$	$\Theta(\|Q\|)$
Brzozowski	$O(2^{\|Q\|})$	$O(2^{\|Q\|})$

Table 1.6: Comparison of different minimization algorithms.

Hopcroft's method is clearly the winner in both categories: its memory requirements are linear in $|Q|$, while the running time is $O(|Q| \cdot lg|Q|)$. As a result, the algorithm outperforms the $O(|Q|^2)$, especially for large statests. If $|Q| = 1024$, then $|Q|^2 = 1048576$, while $|Q| \cdot lg|Q| = 10240$.

Out of the two algorithms with quadratic time complexity, the Aho-Sethi-Ullman method has the better space behavior (linear in $|Q|$). The $\Theta(|Q|^2)$ memory requirements of the Hopcroft & Ullman's table-filling algorithm make it very hard to apply in practice. Also note that the quadratic bound on its running time is not just a worst-case situation. The table-filling algorithm does have to iterate through all state pairs, while the Aho-Sethi-Ullman method might actually terminate before the bound is reached.

As for Brzozowski's algorithm, its $O(2^{|Q|})$ complexity seems to make it a bad choice. However, its actual performance depends strongly on the structure of the automaton being minimized. In many cases, the observed running time is not worse than that of the other algorithms.

The Hopcroft algorithm is the fastest known minimization procedure for general DFSAs. Faster algorithms exist only for specialized types of automata, such as acyclic ones (i.e. automata that do not contain loops). Such automata are often used to encode dictionaries. Therefore, this topic is treated more extensively in the applications chapter, in section 3.3.

1.8 Further Reading

Finite-state automata are one of the best-investigated areas of automata theory. Their practical applications in Computer Science range from compiler design to pattern matching. Out of the vast literature on this subject, Hopcroft, Motwani and Ullman (2001) provide a well-written and easy to read introduction to formal language and automata theory. Finite-state automata are covered in somewhat less detail than in our book. Applications to compiler design, such as building scanners, are handled extensively by Aho, Sethi and Ullman (1988). Aho, Hopcroft and Ullman (1974) address several aspects of finite-state machines from an algorithmic point of view.

As for NLP-oriented literature, we highly recommend the introductory chapter of the collection published by Roche and Schabes (1997), which covers both automata and transducers, as well as further related notions such as *bimachines*.

Chapter 2

Regular Expressions

In this chapter, we look at finite-state automata from a different perspective: we show how *regular expressions*, defined in the sections 2.1 and 2.2, can be viewed as a convenient notation for FSAs. Section 2.3 deals with the actual *compilation* of regular expressions into FSAs. This leads to the notion of a *regular language* (section 2.4), defined as the set of all strings accepted by a particular automaton. In the remainder of the chapter, we show how to implement a number of operations on regular languages in a finite-state framework.

2.1 Regular Expressions and Finite-State Automata

Designing automata by hand is a tedious and very error-prone task. Therefore, it is almost never done in practice. In order to make their design more user-friendly, we need an intuitive formal language for the specification of automata. *Regular expressions*, for short *regexps*, familiar from Perl or Unix commands such as `grep` or `sed`, are such a language.

Consider the following example. You have seen a positive review of a book, and want to buy it. You can look for it in a large text database but, unfortunately, you do not remember the author's name. The only thing you do remember is that it starts with either *Bo* or *Pa* and ends in *son*.

The best solution would be to formulate a regular expression subsuming all strings satisfying these criteria, e.g. `(Bo|Pa)[a-z]*son`. This expression consists of three parts:

`(Bo|Pa)` stands for either *Bo* or *Pa*. The brackets are just a grouping operator.

`[a-z]*` stands for a possibly empty sequence of characters from the range *a* to *z* (`*` denotes an arbitrary number of repetitions of the preceding expression, including the empty string).

`son` matches the string *son*.

It turns out that we can use this regular expression as an instruction to build a non-deterministic finite-state automaton. We do it step-by-step, beginning with the automaton depicted in figure 2.1.

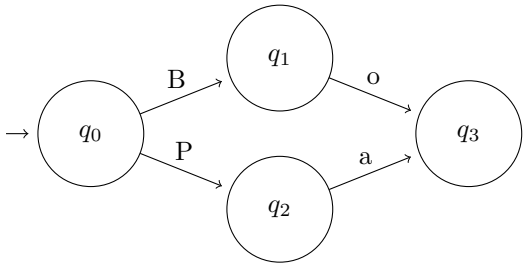


Figure 2.1: FSA corresponding to the sub-expression `(Bo|Pa)`

Starting in the initial state q_0, the automaton may consume either *Bo* (via the states q_1 and q_3) or *Pa* (via q_2 and q_3). In both cases, it ends up in state q_3, which is where the processing of the next part of the string (to be matched against `[a-z]*`) should start.

The subexpression `[a-z]*` can be accounted for by adding a loop accepting any lower-case ASCII character in state q_3, as shown in figure 2.2. Note that after consuming the prefix of the string corresponding to the sub-expression `(Bo|Pa)[a-z]*`, the automaton is still in state q_3.

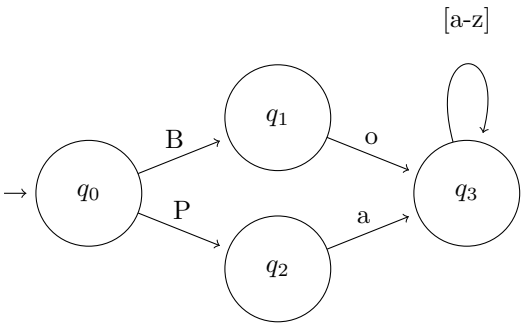


Figure 2.2: FSA corresponding to the sub-expression `(Bo|Pa)[a-z]*`

Finally, we can make the automaton accept the remaining suffix *son* by adding a chain of three states (q_4, q_5, q_6) and connecting them with transitions starting at q_3, as shown in figure 2.3. The state q_6 is made final in order for all strings matching the regular expression to be accepted.

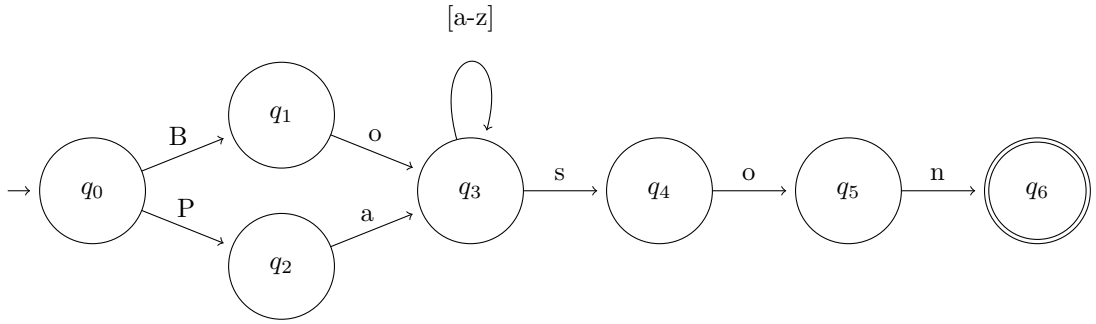


Figure 2.3: FSA corresponding to the sub-expression `(Bo|Pa)[a-z]*`

This rather informal example illustrates a profound fact about the relationship between regular expressions and finite-state automata: they are equivalent. Every regular expression can be compiled into an FSA; for every FSA, there is a regular expression matching exactly the strings accepted by this automaton. The example also shows how to perform such compilation by dividing the regular expression into simpler subexpressions, turning them into FSAs and then joining these FSAs to produce an automaton for the top-level expression.

The remainder of this chapter describes the syntax and semantics of regular expressions (section 2.2), a method for their compilation into FSAs (section 2.3), and some properties of *regular languages* — the class of formal languages denoted by regular expressions/accepted by FSAs (section 2.4).

2.2 Syntax of Regular Expressions

The regular expression `(Pa|Bo)[a-z]*son` considered in the previous section contains the alphabet symbols `B`, `P`, `a`, `n`, `o`, `s`, `z` together with the special characters `|`, `*`, `(`, `)`, `[`, `]` and `-`, which do

not belong to the alphabet. These special characters are a notation for *regular operators*, which combine alphabet symbols into more complex structures.

In this sense, the regular operators perform the same function as the arithmetic operators $+$, $-$, $\cdot$ and :, which join numbers into arithmetic expressions such as $4+9$, $5\cdot 4+88$ or $4\cdot 77-89:3$.

In order to describe a system of operators, one needs to come up with rules specifying a) the legal combinations of operators and alphabet symbols (syntax) and b) what these combinations actually mean (semantics). In the case of arithmetics, the syntax states that $4\cdot 6+7$ is a well-formed arithmetic expression while $+8-9\cdot$ is not. The semantics states that the number 31 is the value (=meaning) of $4\cdot 6+7$.

The same is needed for regular expressions. Their syntax can be described in terms similar to the syntax of arithmetic expressions. As for their semantics, it is most convenient and intuitive to formulate it by specifying what strings match a particular regexp.

The possible combinations of alphabet symbols and regular operators are defined recursively by the following set of rules. At the semantic level, the rules specify the conditions under which strings match a particular class of regexps.

Definition 12 *(Regular expressions)*
Let Σ be an alphabet. Then the set of regular expressions over Σ is defined as follows:

Atomic regular expressions: *Each $a \in \Sigma \cup \{\epsilon\}$ is a regular expression matching only itself, i.e. the string a.*

Concatenation: *If R_1 and R_2 are regular expressions, then R_1R_2 is a regular expression. R_1R_2 matches a string w iff R_1 matches a prefix v of w and R_2 matches the remainder of w.*

Disjunction: *If R_1 and R_2 are regular expressions, then $R_1|R_2$ is a regular expression. $R_1|R_2$ matches a string w iff either R_1 or R_2 matches w.*

Reflexive-transitive closure (Kleene star): *If R is a regular expression, then so is R^*. R^* matches the empty string and any string w that can be split into n substrings $w_1, \ldots, w_n$ such that each w_i matches R.*

Bracketing: *If R is a regular expression, then so is (R). (R) matches a string w if and only if R matches w.*

The reader may have realized that some familiar concepts are missing from this definition. For instance, the range construct `[a-z]` used in the first section of this chapter is not accounted for. It turns out that this and other regular operators are just variants of the above operators introduced for notational convenience. Thus, the range `[a-z]` is nothing else than a disjunction of all the symbols from `a` to `z`, and can be written `a|b|c|...|z`. This and other extensions are discussed in section 2.2.1; for now, we will just stick to the four basic regular operators introduced in definition 12.

In addition to the above recursive definition, we need to define *operator precedence rules*. Again, analogy with arithmetic operators is helpful: the expression $3+4\cdot 5=23$ is evaluated as $3+(4\cdot 5)$ rather than $(3+4)\cdot 5$ because $\cdot$ has precedence over $+$. For regular operators, we assume that the reflexive-transitive closure (`*`) has the highest precedence, followed by concatenation ($\cdot$) and disjunction (`|`). Thus, `ab|c*` is assumed to have the structure `(ab)|(c*)`. Whenever the default structure defined by operator precedence needs to be overridden, we use brackets: `(a(b|c)*)`.

In order to make regexp derivations completely unambiguous, we assume both binary operators (disjunction and concatenation) to be right-associative, which means that `a|b|c` is assumed to have the derivation `a|(b|c)` rather than `(a|b)|c`.[1] With this assumption and the order of precedence for the regular operators, each regular expression has a unique derivation.

[1] This assumption is not required from the semantic point of view because `a|(b|c)` is equivalent to `(a|b)|c` and `a(bc)` is equivalent to `a(bc)`.

2.2.1 Extensions

This basic definition of regular expressions is typically extended by means of further regular operators such as:

Range of symbols: The range notation, e.g. $[abc]$ or $[A-Z]$, is just a shorthand for a disjunction of atomic symbols ($a|b|c$ and $A|B|C|\ldots|Z$, respectively).

Transitive closure: If R is a regular expression, then so is R^+, which is an abbreviation for RR^*.

Wildcard: The symbol . is a regular expression matching each symbol $a \in \Sigma$. It is equivalent to the disjunction of all alphabet symbols. For instance, if Σ is the set of all lower-case ASCII characters, then . is equivalent to $a|b|\ldots|z$.

Note that the extensions are defined in terms of the basic syntax described by the above rules. Thus, they can be treated just as syntactic sugar; for all purposes, it is sufficient to consider the basic syntax introduced in definition 12.

2.3 Compilation of Regular Expressions into FSAs

In section 2.1 we saw how an equivalent FSA can be built for a given regular expression. The technique employed was a good example of what is called a *divide-and-conquer* strategy. The regexp `(Bo|Pa)[a-z]*son` was split into substructures ("divide"), each of which was transformed into an automaton ("conquer"). In the final step, the automata were combined, yielding an FSA equivalent to the regular expression.

In this section, we will show how to generalize this informal example in order to produce a generic recursive compilation technique. The divide step will recursively follow the derivation of the regular expression, down to the atomic regular expressions at the leaves of the derivation. For each regular operator, the strategy is first to identify its operands, then compile them into FSAs, and finally combine the resulting automata according to the semantics of the respective rule in definition 12.

For uniformity, it is assumed that a regular expression is compiled into an ϵ-NFSA with exactly one initial state q^{in} and exactly one final state q^{out}. We formulate a separate compilation method for each of the rules stated in definition 12, starting with the simple case of atomic regexps.

2.3.1 Atomic Regular Expressions

An ϵ-NFSA corresponding to the atomic regexp `a` must accept exactly one string, namely a itself. The construction of such an automaton is trivial:

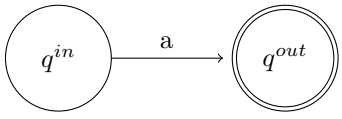


Figure 2.4: NFSA for the regular expression `a`.

Also trivial is the NFSA for the empty regexp ϵ:

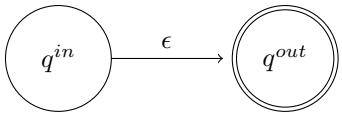


Figure 2.5: NFSA for the empty regular expression.

2.3.2 Complex Regular Expressions

A complex regular expression is a regular operator together with its operand(s). According to our divide-and-conquer strategy, we can assume that the operands, which are less complex regular expressions, have already been compiled into FSAs. The remaining task is to combine these FSAs into an automaton equivalent to the top-level expression.

Disjunction

The disjunction rule states that $R_1|R_2$ matches a string w if and only if at least one of the operands matches w. If A_1, A_2 are NFSAs equivalent to R_1 and R_2, respectively, then the automaton for $R_1|R_2$ shall accept a string w if and only if w is accepted by either A_1 or A_2. Such an automaton can be created out of A_1 and A_2 by adding a new initial state q_{in} and a new final state q_{out}, and introducing four ϵ-transitions: two linking q^{in} with the initial states of A_1 and A_2, and two linking the final states of either automaton to q^{out}. Figure 2.6 illustrates this construction.

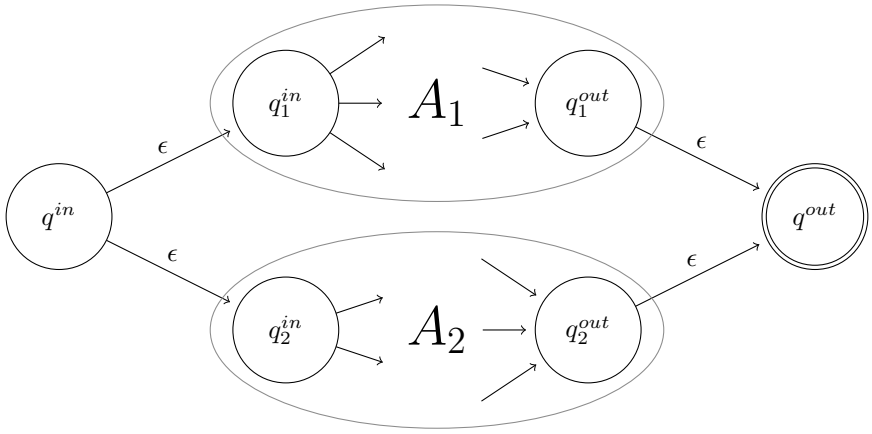


Figure 2.6: Construction of an NFSA for a disjunctive regular expression $R_1|R_2$. A_1 and A_2 are NFSAs equivalent to the respective operands.

Note that q_1^{in} and q_2^{out} are no longer initial. Likewise, q_1^{out} and q_2^{out} are no longer final.

Concatenation

According to definition 12, a concatenation R_1R_2 matches a string w if and only if R_1 matches a prefix of w and R_2 the remainder of w. If A_1 and A_2 are NFSAs equivalent to R_1 and R_2, respectively, then the desired functionality is nothing else than first running A_1 on w and stopping in the final state q_1^{out} after accepting some prefix of w, and then running A_2 on the remainder and stopping in the final state q_2^{out} of A_2.

This functionality is implemented by linking q_1^{out} to q_2^{in} with an ϵ-transition. Obviously, q_1^{out} is no longer a final state after this operation; q_2^{in} is no longer initial. Figure 2.7 illustrates the idea.

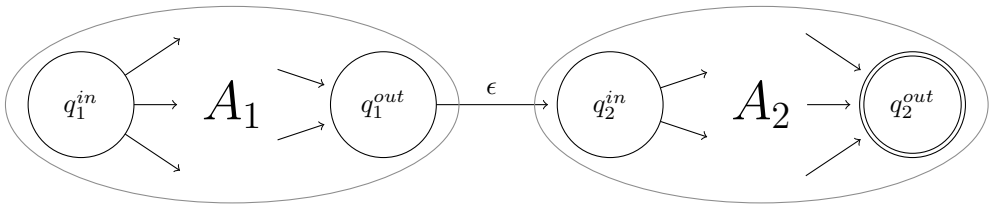


Figure 2.7: Construction of an NFSA for a regular expression of the form R_1R_2 (concatenation). A_1 and A_2 are NFSAs equivalent to the respective operands.

Reflexive-Transitive Closure

A string w matches a regular expression of the form R_1^* if and only if $w = \epsilon$ or $w = w_1 \cdot \ldots w_n$, where each w_i is a string that matches the operand R_1. Assume that A_1 is the NFSA resulting from the compilation of R_1, having exactly one initial state q_1^{in} and exactly one final state q_1^{out}.

We introduce a new initial state q^{in} and a new final one q^{out}. Since ϵ matches R_1^*, the two states are connected with an ϵ-transition.

For strings $w \neq \epsilon$, the matching of w against R_1^* can be implemented by successively accepting portions of w using automaton A_1: we start in state q_1^{in}, consume w_1 and end up in state q_1^{out}, from where we jump to state q_1^{in} and accept w_2, etc. The move from q_1^{out} to q_1^{in} after accepting w_i can be modeled by an ϵ-transition connecting the two states. Since q^{in} and q^{out} are the new initial and final state, we connect them to the original automaton A_1 by ϵ-transitions: one from q^{in} to q_1^{in} and one from q_1^{out} to q^{out}. The result is shown in figure 2.8.

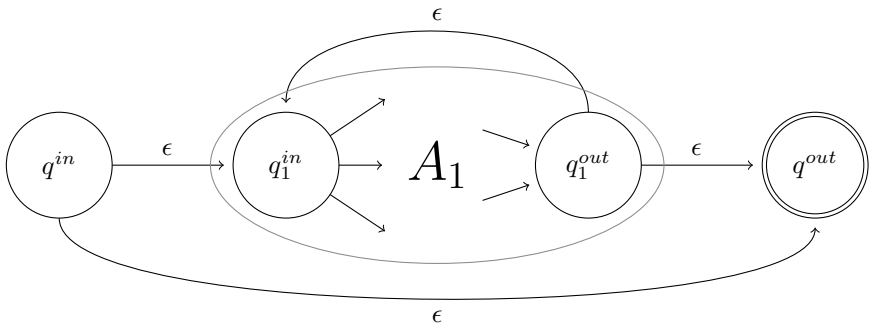


Figure 2.8: Construction of an NFSA for a regular expression of the form R_1^* (reflexive-transitive closure). A_1 is an NFSA equivalent to the operand R_1.

Bracketing

This last case is trivial. A bracketed regular expression (R) matches exactly the same strings as the operand regexp R. Therefore, if A is an NFSA equivalent to R, then A is also equivalent to (R).

2.4 Regular Languages

So far, the semantics of regular expressions has been described in terms of strings matching a regular pattern. This formulation is quite intutitive, especially for those readers who are familiar

with the regular expressions used in `grep` or `awk`. In section 2.3, this "matching semantics" was directly translated into the operational semantics of the FSAs built in the process of compiling a regexp into an automaton.

However, in order to be mathematically precise, we need to give regular expressions a *denotational semantics*, i.e. interpret each regexp by identifying some object or collection of objects as its meaning. Returning to the analogy to arithmetic operators, we can see that arithmetic expressions are interpreted as numbers: e.g. 14 is the meaning of $3 \cdot 4 + 2$. Now we need to find an analogous denotate for regular expressions.

It turns out that this is possible if a regular expression is viewed as a finite way of describing the (potentially infinite) set of strings it matches. For example, `a*` is a finite notation for the set of all sequences of a's, including the empty one: $\{\epsilon, a, aa, aaa, \ldots\}$. Thus, we can say that every regular expression *denotes* a language: namely the language of all strings that match this regular expression. Languages denoted by regular expressions are called *regular languages*.

The equivalence of regular expressions and finite-state automata means that:

- for each regular expression R there exists an FSA A accepting the language denoted ("matched") by R;
- conversely, for every FSA A, there is a regular expression denoting the language accepted by A.

Figure 2.9 illustrates the relationship between regular expressions, finite-state automata and regular languages.

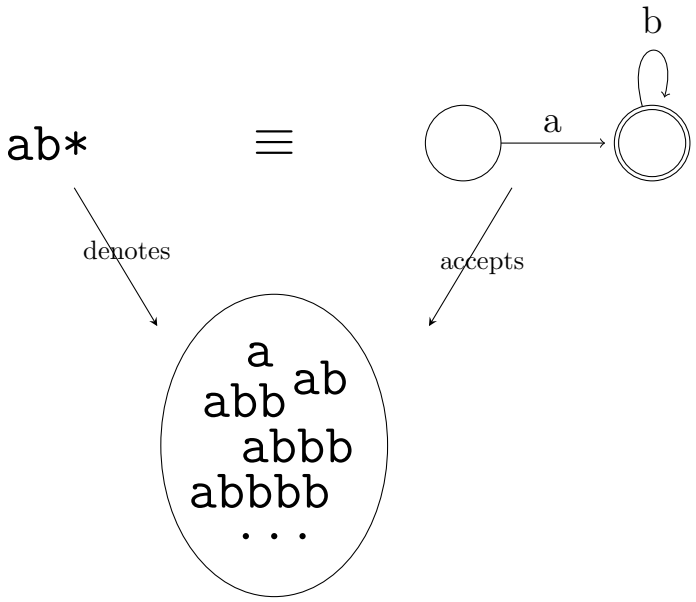


Figure 2.9: The regular expression `ab*`, the equivalent FSA, and the regular language they denote/accept.

Regular languages can be defined — as they were above — indirectly: either as languages denoted by regular expressions, or languages accepted by FSAs. However, an explicit set-theoretical formulation is possible, and often beneficial. It is a recursive definition that parallels the definition of regular expressions with the only difference that it deals with regular operators operating on languages (sets of strings) rather than regular expressions.

The set-theoretical notation for the regular operators is slightly different from the notation used for regular expressions in that the counterpart of disjunction $(R_1|R_2)$ is called *union* and denoted $L_1 \cup L_2$. For concatenation and the reflexive-transitive closure, the symbols $\cdot$ and $*$ are

used, respectively. Bracketing, which is a purely notational concept, does not have a set-theoretical counterpart.

The formal definition of the three operators is given below.

Union: the union $L_1 \cup L_2$ of two languages L_1 and L_2 is defined as the union of the two sets L_1 and L_2. For example, if $L_1 = \{a, aba, aaac\}$ and $L_2 = \{bcb, cc\}$, then $L_1 \cup L_2 = \{a, aba, aaac, bcb, cc\}$.

Concatenation: the concatenation $L_1 \cdot L_2$ of two languages L_1 and L_2 is defined as $L_1 \cdot L_2 = \{u \cdot v : u \in L_1, v \in L_2\}$. In other words, it is the set of all strings that can be created by concatenating a string from language L_1 with a string from language L_2. If L_1 and L_2 are as in the example above, then

$$L_1 \cdot L_2 = \{abcb, ababcb, aaacbcb, acc, abacc, aaaccc\}.$$

Reflexive-transitive closure: If L is a language, then its reflexive-transitive closure L^* is the set of all strings formed by concatenating any finite number (including 0) of strings in L:

$$L^* = \{w_1 \cdot \ldots \cdot w_n : n \in N\}$$

For example, if $L = \{a, b\}$, then any finite sequence of a's and b's is in the language L^*.

Regular languages can now be formally defined as follows.

Definition 13 *(Regular languages) Let Σ be a finite alphabet. Then*

1. *$\{\epsilon\}$ is a regular language,*
2. *for any $a \in \Sigma$, $\{a\}$ is a regular language,*
3. *if L_1 and L_2 are regular languages, then $L_1 \cup L_2$ is a regular language;*
4. *if L_1 and L_2 are regular languages, then $L_1 \cdot L_2$ is a regular language;*
5. *if L is a regular language, then L^* is a regular language;*
6. *nothing else is a regular language.*

The set-theoretical perspective on regular languages makes it easy to solve a number of problems. In the remainder of this section, we will focus on one specific class of problems, called *closure properties*. The idea is to determine if a set-theoretical operation such as union, intersection or complement produces a regular language when applied to sets that are regular languages. If this is always the case, then we say that regular languages are *closed* under this particular operation. For instance, it follows directly from definition 13 that regular languages are closed under union (point 3), concatenation (point 4) and the reflexive-transitive closure (point 5). For other set-theoretical operators, things are more tricky.

In the following, we assume that all the languages considered are defined over the same alphabet Σ.

2.4.1 Complement

The complement of a regular language L is denoted $\bar{L}$ and defined as the set of all strings Σ *not* in L:

$$\bar{L} = \{w \in \Sigma^* : w \notin L\}$$

In order to show that $\bar{L}$ is regular, we will construct an FSA that accepts $\bar{L}$. Due to the equivalence of regular languages and FSAs, this construction is sufficient to prove that regular languages are closed under complement.

Let $A = (\Sigma, Q, q_0, F, \delta)$ be a DFSA accepting L. Unlike the majority of algorithms in this book, we assume that the transition function δ is *total*, i.e., for each state $q \in Q$ and symbol $a \in \Sigma$, there is a transition labeled a leaving state q (cf. the discussion of complete and incomplete automata in section 1.2). As demonstrated on page 8, a partial transition function can easily be made total by adding a new non-final *dead state* q_{dead}, setting $\delta(q, a) = q_{dead}$ for all $q \in Q$ and $a \in \Sigma$ for which $\delta(q, a)$ was initially undefined (including $\delta(q_{dead}, a) = q_{dead}$ for all $a \in \Sigma$). Figure 2.10 illustrates this construction.

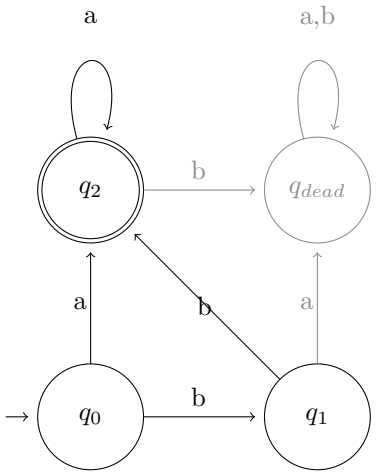


Figure 2.10: A partial FSA transition function (visualized as black arcs) is made a total function by introducing a non-final dead state (q_{dead}) and making all originally undefined transitions (grey arcs) point to q_{dead}. Once in state q_{dead} the automaton loops in that state whatever symbol is read.

With this additional assumption, $\delta(q_0, w)$ is defined for all $w \in \Sigma^*$, and we can distinguish two cases:

- if $w \in L$, then $\delta(q_0, w) \in F$;
- if $w \notin L$, then $\delta(q_0, w) \notin F$.

Since we want to build an automaton that accepts strings $w \notin L$ and rejects $w \in L$, we can keep the transitions and states of A, but swap the final and non-final states, creating an automaton $\bar{A} = (\Sigma, Q, q_0, Q \backslash F, \delta)$. Obviously, $\mathcal{L}(\bar{A}) = \bar{L}$. This proves that regular languages are closed under complementation.

The above proof is *constructive*: it not only shows that an automaton for $\bar{L}$ exists, but also demonstrates how such an automaton can be constructed.

It is worth stressing that the input to this construction must be a *deterministic* automaton. As a result, if we want to determine the complement of a regular expression according to the above construction, an extra determinization step is required (recall that the regexp compilation method outlined in section 2.3 produces non-deterministic automata). Since the worst-case running time of the subset construction algorithm used for determinization is exponential, the computation of the complement of a regular language may be expensive. For this reason, unrestricted complement is often not supported by regular expression libraries and tools.

A restricted version of the complement is the negated set of characters, e.g. `[^a-c]`, which is easily compiled into the disjunction of all characters not in the set.

2.4.2 Intersection

The intersection of two regular languages L_1 and L_2 is denoted $L_1 \cap L_2$ and defined as the set of all strings $w \in \Sigma^*$ such that w is in both L_1 and L_2:

$$L_1 \cap L_2 = \{w \in \Sigma^* : w \in L_1 \text{ and } w \in L_2\}$$

In order to check if the language $L_1 \cap L_2$ is regular, consider its complement $\overline{L_1 \cap L_2}$. If we can prove that $\overline{L_1 \cap L_2}$ is regular, then so is $L_1 \cap L_2 = \overline{\overline{L_1 \cap L_2}}$, because $\overline{\overline{A}} = A$ for any set A.

Furthermore, we can use an identity called *De Morgan's Law* and stating that $\overline{A \cap B} = \overline{A} \cup \overline{B}$ for arbitrary sets A and B.

$$\overline{L_1 \cap L_2} = \overline{L_1} \cup \overline{L_2}$$

Now, since both L_1 and L_2 are regular, so are their complements $\overline{L_1}$ and $\overline{L_2}$. Hence, $\overline{L_1 \cap L_2}$, being the union of two regular languages, is regular, too, and so is its complement $L_1 \cap L_2 = \overline{\overline{L_1 \cap L_2}}$. Thus, regular languages are closed under complementation.

Construction of an Intersection Automaton

As in the case of complementation, the proof that regular languages are closed under intersection is constructive, i.e. it defines an algorithm for the computation of an automaton that accepts the intersection of two regular expressions R_1 and R_2. For this, the regular expressions need to be compiled into two FSAs A_1 and A_2. Then we create the complement automata $\bar{A}_1$ and $\bar{A}_2$ and construct their union. The complement of the union automaton is a DFSA that accepts $\mathcal{L}(R_1) \cap \mathcal{L}(R_2)$.

This construction involves three computations of a complement FSA, which — as mentioned above in section 2.4.2 — may be expensive. It turns out that we can do much better.

Recall that a string $w \in \Sigma^*$ is in the intersection language if and only if it is accepted by both automata. The trick is thus to construct an NFSA $A = (\Sigma, Q, I, F, \Delta)$ that emulates the parallel application of both automata to w.

For this, let us assume that the states of A are actually pairs of states: one being the state of A_1 and the other one the state of A_2 for a particular configuration of both automata. In other words, if A_1 is in state q_1 and A_2 in state q_2 after consuming some string w, we say that A is in state $\langle q_1, q_2 \rangle$.

Obviously, we call such a pair initial if both q_1 and q_2 are initial (in the respective automata, i.e. $q_1 \in I_1$ and $q_2 \in I_2$). Also, $\langle q_1, q_2 \rangle$ is called final if both q_1 and q_2 are final.

As for the transition function Δ, observe that A moving from $\langle q_1, q_2 \rangle$ to $\langle r_1, r_2 \rangle$ via some symbol a is the same as A_1 moving from q_1 to r_1 (i.e. $r_1 \in \Delta_1(q_1, a)$) and A_2 moving from q_2 to r_2 (i.e. $r_2 \in \Delta_2(q_2, a)$) via a. Therefore, the set of transitions labeled a and leaving $\langle q_1, q_2 \rangle$ must be identical to the set of all such pairs $\langle r_1, r_2 \rangle$.

Formally, the intersection automaton $A = (\Sigma, Q, I, F, \Delta)$ is defined as follows.

$$Q = \{\langle q_1, q_2 \rangle \in Q_1 \times Q_2 : q_1 \in \Delta_1^*(I_1, w), q_2 \in \Delta_2^*(I_2, w) \text{ for some } w \in \Sigma^*\}.$$

$$I = I_1 \times I_2.$$

$$F = Q \cap (F_1 \times F_2).$$

$$\Delta(\langle q_1, q_2 \rangle, a) = \Delta_1(q_1, a) \times \Delta_2(q_2, a).$$

The definition of the stateset makes sure that only pairs $\langle q_1, q_2 \rangle$ reachable from I via a string $w \in \Sigma^*$ are in Q. The actual construction parallels the algorithms 1.4.1 (subset construction) and 1.6.1 (equivalence of automata). Starting in pairs of initial states, all paths through A_1 and A_2 are explored using a queue of state pairs (which become states of A). If $\langle q_1, q_2 \rangle$ is the pair at the

front of the queue, the algorithm sets $\Delta(\langle q_1, q_2\rangle, a) = \Delta_1(q_1, a) \times \Delta_2(q_2, a)$. If $\Delta_1(q_1, a) \times \Delta_2(q_2, a)$ contains any new pairs $\langle r_1, r_2\rangle$, these pairs are enqueued in order to be explored later.

Algorithm 2.4.1: INTERSECT$((\Sigma, Q_1, I_1, F_1, \Delta_1), (\Sigma, Q_2, I_2, F_2, \Delta_2))$

```
I ← ∅
F ← ∅
Δ ← ∅
for each q1 ∈ I1
  for each q2 ∈ I2
   I ← I ∪ {⟨q1, q2⟩}
   ENQUEUE(Queue, ⟨q1, q2⟩)
Q ← I
while Queue ≠ ∅
   ⟨q1, q2⟩ ← DEQUEUE(Queue)
   for a ∈ Σ
      Δ(⟨q1, q2⟩, a) ← ∅
      for each r1 ∈ Δ1(q1, a)
        for each r2 ∈ Δ2(q2, a)
         Δ(⟨q1, q2⟩, a) ← Δ(⟨q1, q2⟩, a) ∪ {⟨r1, r2⟩}
         if ⟨r1, r2⟩ ∉ Q
            Q ← Q ∪ {⟨r1, r2⟩}
            ENQUEUE(Queue, ⟨r1, r2⟩)
            if r1 ∈ F1 ∧ r2 ∈ F2
              F ← F ∪ {⟨r1, r2⟩}
return (A)
```

The total number of possible state pairs $\langle q_1, q_2\rangle$ is $|Q_1 \times Q_2| = |Q_1| \cdot |Q_2|$. The fact that only *new* pairs are enqueued makes sure that each of these pairs is processed at most once, which involves iterating through all transition pairs $\langle q_1, a, r_1\rangle$, $\langle q_2, a, r_2\rangle$ leaving q_1 and q_2 respectively. This happens in the two nested **for each** loops; the number of transitions considered for each automaton is bounded by $|\Delta_1|$ and $|\Delta_2|$, respectively. The running time of the intersection algorithm is thus $O(|Q_1| \cdot |Q_2| \cdot |\Delta_1| \cdot |\Delta_2|)$.

2.4.3 Difference

The difference of two regular languages L_1 and L_2 is denoted $L_1 \backslash L_2$ and defined as the set of all strings in L_1 that are not in L_2:

$$L_1 \backslash L_2 = \{w \in \Sigma^* : w \in L_1 \text{ and } w \notin L_2\}$$

But $L_1 \backslash L_2$ is the same as $L_1 \cap \overline{L}_2$. Since regular languages are closed under complementation and intersection, this means that $L_1 \backslash L_2$ is regular. Hence, regular languages are closed under set difference.

2.4.4 Reversal

The reversal L^{-1} of a language L is defined as the set of all strings created by reversing a string $w \in L$:

$$L^{-1} = \{w^{-1} : w \in L\}$$

For example, $\{\epsilon, ab, babb\}^{-1} = \{\epsilon, ba, bbab\}$.

If L is regular, then so is L^{-1}. An acceptor for L^{-1} can be constructed from an acceptor $A = (\Sigma, Q, I, F, \Delta)$ for L by swapping the initial and the final states and reversing the transitions of

A. The reversed automaton is denoted A^{-1} and defined formally as $A^{-1} = (\Sigma, Q, F, I, \Delta^R)$, where $\Delta^R = \{\langle q, a, r\rangle : \langle r, a, q\rangle \in \Delta\}$. The implementation of the reversal of a DFSA is straightforward. The complexity of this operation is $\Theta(|Q| + |\Delta|)$.

If R is a regular expression, then R^{-1} is a regular expression denoting the language $\mathcal{L}(R)^{-1}$.

2.5 Further Reading

As already mentioned above, regular expressions are an extremely well-investigated building block of the Unix operating system. There exists vast literature on regexp compilation, and the reader should bear in mind that the only very basic examples are treated in the present chapter.

The compilation method (called *Thompson's algorithm*) presented in section 2.3 is not the only one available, but has the advantage of simplicity. A more complex compilation method is the *Glushkov* or *McNaughton-Yamada algorithm* (Glushkov 1961, McNaughton and Yamada 1960). It creates automata without ϵ-transitions.

One of the main difficulties faced by the regular compilers employed by interpreted programming languages such as Perl or Awk is the need for fast construction of an automaton for a given regexp. The naïve way would be first to compile the regexp into an NFSA, and then apply the subset construction algorithm in order to create a DFSA. Due to the exponential complexity of the subset construction algorithm, this may lead to very slow — in fact, unacceptable — compilation. Thus, regular expressions are often only compiled into a non-deterministic automaton, or a *lazy* or *on-the-fly* determinization algorithm is employed which does not expand the whole DFSA at once.

In NLP applications, the requirements are often very different. The regular expressions (e.g., as part of a grammar) are typically available off-line and can be precompiled into a minimal deterministic FSA. Hence, the compilation efficiency issue does not arise to the same extent.

On the other hand, if the regexps are available off-line, then using libraries optimized for on-line construction and compilation may introduce inefficiencies due to non-determinism or determinization performed at runtime. As a result, built-in regular compilers in Perl or in the Unix environment are hardly the most efficient option for many NLP applications. In such a situation, it is advisable to use a dedicated finite-state library in order to perform the expensive operations off-line.

Chapter 3

Applications of Finite-State Automata

This chapter introduces the reader to several generic finite-state programming techniques. Based on problems often arising in NLP practice, we show how they can be solved efficiently using finite-state automata. The applications considered comprise tokenization (section 3.1), pattern matching (section 3.2) and the construction and encoding of dictionaries (section 3.3).

3.1 Tokenization

Text processing applications often start with tokenization, i.e. a segmentation of the input string into words, spaces, digits, punctuation marks, etc. The segments produced in this step are called *tokens* and typically carry information about the *token class* they belong to. Based on this information, tokens may be processed differently. For instance, all a spell checker looks up in a dictionary is words; spaces or punctuation marks are ignored.

The following figure shows a possible simple tokenization of the sentence *They were fined $100.* into five classes of tokens: *word* (any sequence of alphabetic characters), *space* (any sequence of whitespace characters), *number* (any sequence of digits), *punct(uation)* (a single punctuation character) and *symbol* (any other symbol, e.g. $, %, etc.).

They		were		fined		$	100	.
word	*space*	*word*	*space*	*word*	*space*	*symbol*	*number*	*punct*

It is easy to construct a deterministic FSA encoding these basic token classes (figure 3.1).

The automaton consists of six states: the initial state q_0 and one state per token class. On seeing a character, the automaton jumps to the state corresponding to the class of the token being processed. For example, the first character of the sentence (*'T'*) is recognized as the start of a *word* token, hence the automaton jumps to state q_{word}. There it stays, accepting as many alphabetic characters as possible, i.e. the remainder of the first token *They*.

After scanning the *y*, the automaton discovers a space for which there is no transition in state q_{word}. This means the end of the current token. Therefore, *They* is returned as the first token and classified as a *word*. The automaton goes back to the initial state and starts the recognition of the next token at the space character following *They*. The space is consumed, leading the automaton to state q_{space}. Accordingly, a *space* token is returned. This procedure is repeated until the whole string is consumed.

Two things are noteworthy here. Firstly, the final states are given a semantics: the state the DFSA ends up in determines the token class. Secondly, the input string is not accepted in one go, as in the examples in the preceding sections, but portionwise. Each time, the algorithm pursues a *longest match* strategy: once in a final state such as q_{word}, the automaton accepts as many

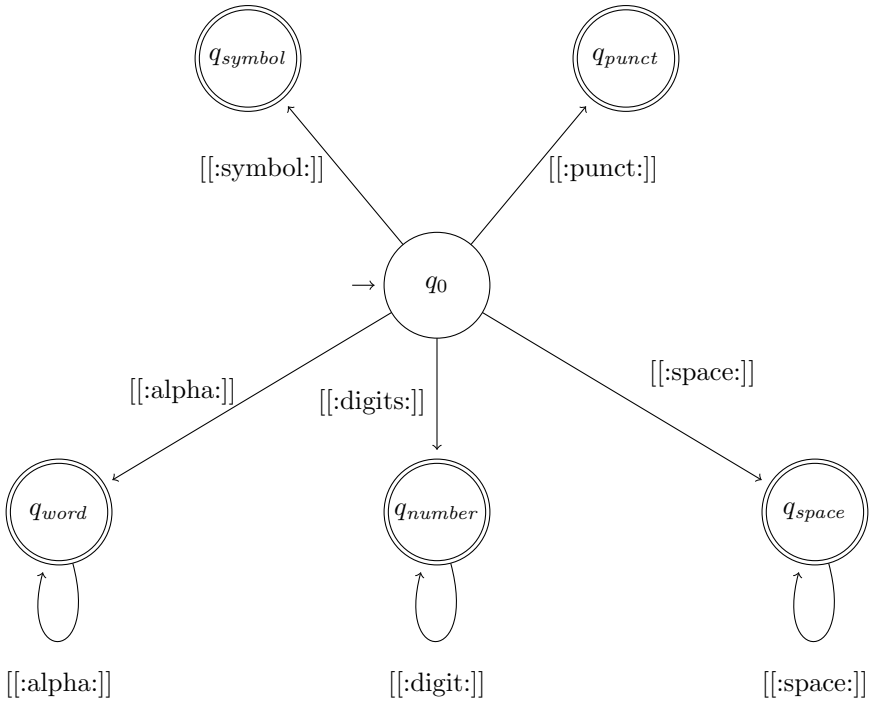


Figure 3.1: A basic tokenizer DFSA.

characters as possible. This is made possible by the loops in states q_{word}, q_{space} and q_{number}. The remaining final states do not contain such loops, hence the corresponding tokens (*punct* and *symbol*) are always one character long.

3.1.1 Finer Token Classes

The example tokenizer presented in the previous section is likely to be deemed too simple for most text processing applications. The token classes it recognizes are too coarse-grained. A user may be interested in refining this classification, for example by

- making the tokenizer distinguish between initials (token class *letter*) and longer words;
- defining an extra token class *float* for float-point numbers such as 524.12, which are currently split into two digits and a punctuation character in between.

The new classes are accounted for by extending the automaton with the final states q_{letter} and q_{float}, as shown in figure 3.2. On reading the first alphabetic character of a token, the DFSA moves from q_0 to q_{letter}. If there are no more alphabetic characters following it, the character is split off as a separate token, and the state the automaton is in determines the token class: *letter*. If the first alphabetic character is followed by more alphabetic characters, the automaton ends up in state q_{word}. This is indeed the intended interpretation: a *word* is now defined as a sequence of two or more alphabetic characters.

Integers and float-point numbers are acounted for by the states q_{number}, q_{sep} and q_{float}. On scanning the first digit of a number, the automaton goes to state *number* and remains there, possibly reading further digits. If that is all, the token is classified as *number*.

If the digits are followed by a dot, the automaton jumps to state q_{sep}. Note that this state is non-final because expressions such as 100. are not legal float-point numbers. Only when the automaton sees another digit, does it move to the final state q_{float} denoting floats.

The existence of q_{sep}, a state that is both non-initial and non-final, may enforce backtracking. Consider the application of the tokenizer to our initial example, the string *He was fined $100.*

After processing the token $ (recognized as a *symbol*), the DFSA consumes the remainder *100.*, ending up in q_{sep}. Since q_{sep} is non-final, it does not denote any legal token class. Thus, the automaton must backtrack to the last final state it has seen, namely q_{number}. Accordingly, the string *100* is returned as the current token and classified as a *number*. The remaining dot is read again and classified as a *punct* token.

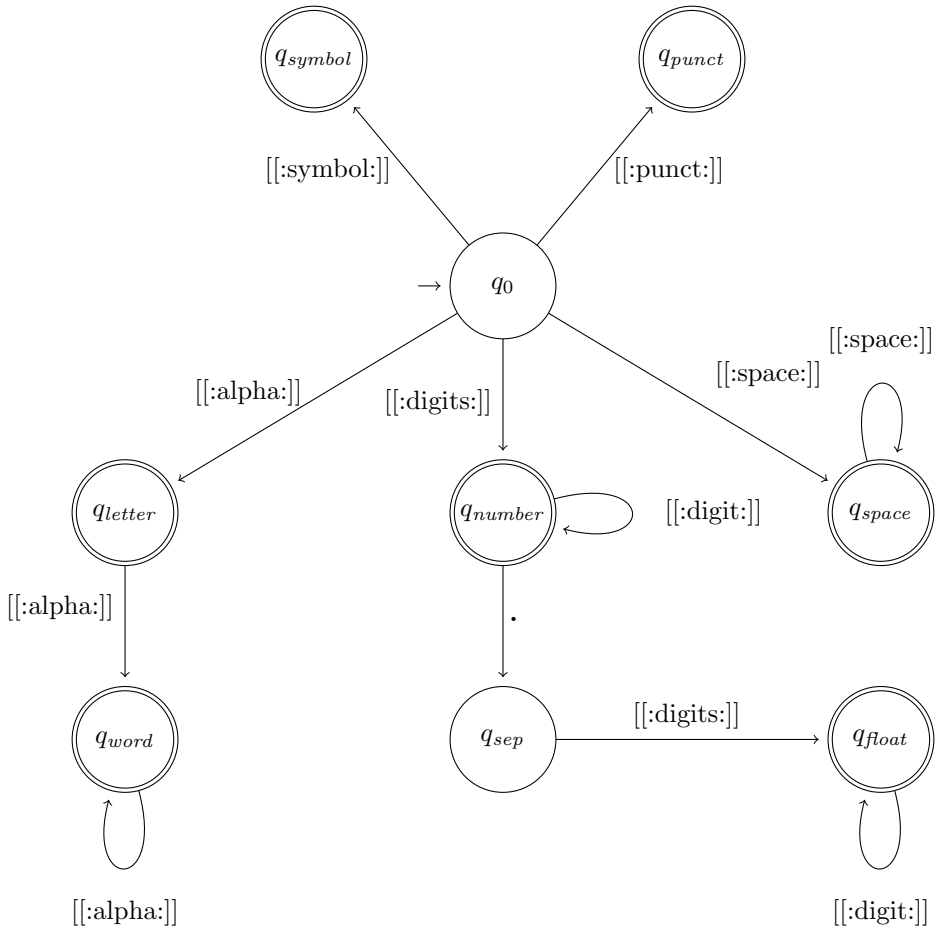


Figure 3.2: A basic tokenizer DFSA.

The pseudocode of the longest-match tokenization algorithm is presented in algorithm 3.1.1. The function TOKENIZE() is passed the tokenizer DFSA A, a string *Text*, and a mapping *TokenClass* from states of A to token classes. We assume that $TokenClass[q_0] = unclassified$. The string *Text* is successively consumed by repeated calls to the procedure LONGESTMATCH(). LONGESTMATCH() is passed two arguments: *Text* and the position *StartPos* where the next token is expected to start. It consumes the longest matching prefix of the remaining portion of *Text* and returns the end position of the recognized token (*StartPos*) together with the final state it ends up in (q_{fin}). The corresponding substring of *Text*, together with the token class information associated with q_{fin} and expressed by $TokenClass[q_{fin}]$ is then appended to the token list.[1]

[1] Note that LONGESTMATCH() may fail to consume any characters of the input string, for example if there is no transition for $Text[StartPos]$ leaving q_0. In such a case, the LONGESTMATCH() returns the pair $(StartPos, q_0)$. TOKENIZE() then splits off the current character as an unclassified one-symbol token (alternatively, such unclassified characters may be ignored by the tokenizer).

Algorithm 3.1.1: TOKENIZE(A, *Text*, *TokenClass*[])

```
StartPos ← 1
TokenList ← []
while StartPos ≤ |Text|
   (EndPos, q_fin) ← LONGESTMATCH(A, Text, StartPos)
   if EndPos = Pos
      EndPos ← EndPos + 1
   APPEND(TokenList, (Text[StartPos, EndPos], TokenClass[q_fin])
   StartPos ← EndPos
return (TokenList)

procedure LONGESTMATCH(A, Text, StartPos)
 q ← q_0
 q_fin ← q_0
 EndPos ← StartPos
 for i ← StartPos to |Text|
    if δ(q, Text[i]) defined
       q ← δ(q, Text[i])
       if q ∈ F
          q_fin ← q
          EndPos ← i
     else break
 return (EndPos, q_fin)
```

The running time of the longest-match tokenization algorithm can be measured by the total number of iterations of the **for** loop in function LONGESTMATCH(). At the very least, there are $|\mathit{Text}|$ iterations as each character of the input string needs to be processed. In addition, some overhead may be caused by backtracking. The maximum number $t_{backtracking}$ of backtracking steps depends on the automaton. In the above DFSA, backtracking is limited to one character per token, and the maximum number of tokens is $|\mathit{Text}|$, hence $2 \cdot |\mathit{Text}|$ is the upper bound on the entire number of iterations. In general, if the number of backtracking steps is limited by some fixed number K, there are at most $K \cdot |\mathit{Text}|$ iterations. Since the execution time of the operations performed in the loop of LONGESTMATCH() does not depend on *Text*, the running time of the longest match tokenization algorithm is $\Theta(|\mathit{Text}|)$.

If unlimited backtracking is possible, the running time of the algorithm is quadratic. To see that, suppose we want to recognize emoticons such as *smiley faces* as a separate token class. A smiley face can have different forms, e.g. `:-)`, `;->`, `:-------(`, `:--))))))))))`, etc. In general, it starts with a character encoding the eyes of the face (either `:` or `;`), followed by the nose (e.g. `-`), and finally the mouth (`(` or `)`). The eyes are optional because `--(` is also a potential emoticon.

The recognition of such face-like emoticons is the task of the DFSA shown in figure 3.3.

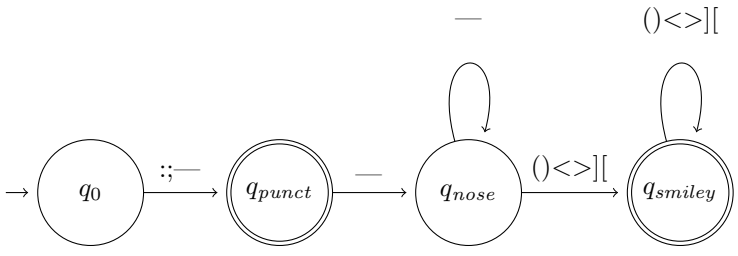


Figure 3.3: A DFSA encoding emoticons.

Now consider the application of this DFSA to a string of length n, starting with a colon followed by $n-1$ dashes: `:---...---`. On seeing the initial `:`, the automaton enters state q_{punct}. The colon is a legal *punct* token, but it might also start a smiley face, so the matching continues. The algorithm traverses the entire sequence of n dashes only to discover that the sequence is not terminated by a "mouth character". Hence, the DFSA reaches the end of the input string in the non-final state q_{nose} and must backtrack to the last previously seen final state, which is the occurence of q_{punct} after consuming the first character. Thus, the automaton has traversed $n-1$ characters, but the token returned is only one character long.

The next call to LongestMatch() is passed the remainder `---...---`. Since there is a transition from q_0 to q_{nose} labelled —, the automaton consumes all $n-1$ dashes, but does not end up in a final state. Hence, another *punct* token is returned, namely —.

This happens n times; at i-th time, the algorithm traverses the $n-i$ characters from the current position to the end of the string. Thus, the total number of iterations is $\Sigma_{i=1}^{n} i = \frac{n^2+n}{2} = O(n^2)$.

The above example shows that tokenizer automata should be designed carefully in order to avoid quadratic runtime behavior. Fortunately, patterns that give rise to unlimited backtracking are rare in text processing applications. As a result, tokenization can be assumed to be achievable in linear time.

3.1.2 Regexp-Based Tokenization

As already mentioned in chapter 2, automata are not normally designed by hand. Even if the tokenizer DFSAs constructed so far have been simple, further refinement of the token classes is hardly doable in the same way. We might want the tokenizer to distinguish between upper-case, lower-case, capitalized and mixed-case words, recognize currency amounts, etc. All this may and should be formulated using regular expressions:

Lower-case initial (LCI): `[[:lower:]]`

Upper-case initial (UCI): `[[:upper:]]`

Lower-case word (LCW): `[[:lower:]][[:lower:]]+`

Upper-case word (UCW): `[[:upper:]][[:upper:]]+`

Capitalized word (CW): `[[:upper:]][[:lower:]]+`

Number (N): `([[:digit:]]+,)*[[:digit:]](\.[[:digit:]]+)?`

Obviously, this is a more elegant and less error-prone formulation than the direct automata construction done in the previous tokenizer examples. The regular expressions can be compiled into FSAs and then used in tokenization.

One possible way of doing that would be to turn each token class regexp R_i into a DFSA A_i, and then match all A_i's against the input string w. The longest matching prefix of w would be split off as the next token. The R_i matching this prefix would define its token class.

This solution is correct, but not optimal. It involves running the n DFSAs separately, making the execution time of the algorithm directly dependent on the number of token classes. As a matter of fact, we have already seen that tokenization can be done with a *single* DFSA (cf. algorithm 3.1.1).

What we may do is to create a DFSA for the union $R_1|\ldots|R_n$ of all the token class regular expressions. According to the semantics of regular union, the expression $R_1|\ldots|R_n$ matches exactly the strings that match any of the regexps $R_1, \ldots, R_n$. The successive calls to LongestMatch() will thus return exactly the same strings as the naïve algorithm based on matching the regexps independently.

However, tokenization also requires the identification of the *token class* for each token. In algorithm 3.1.1, it is determined by the (final) state of the DFSA after it has read the characters

of the token. As long as the DFSA is constructed by hand, we can specify this explicitly. Thus, state q_{word} corresponded to the token class *word*, etc.

In an automatically compiled DFSA, such information is no longer available. The equivalence operations used during compilation may change state indices, so any connection between states and the patterns they originate in is lost.

In order to re-establish this relationship, we use a trick. For each token class R_i, we introduce a different *sentinel character* $\$_i \notin \Sigma$. Instead of computing the union of the R_i, we now append the sentinel characters to the respective patterns and create a deterministic acceptor for the regular expression $R_1\$_1|\ldots|R_n\$_n$.

A fragment of such an acceptor is shown in figure 3.4.

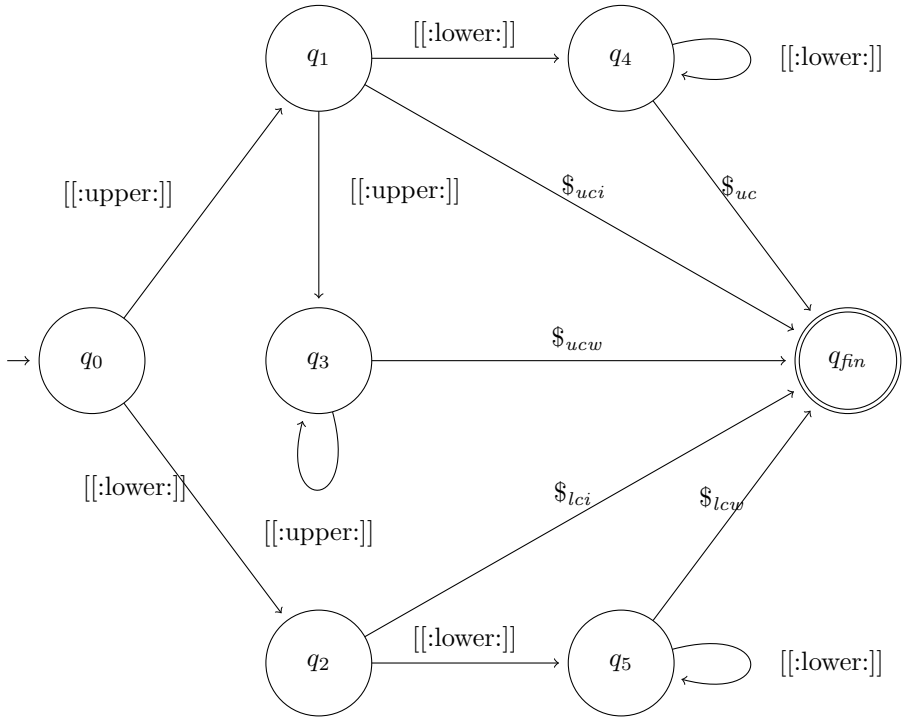


Figure 3.4: Fragment of a DFSA encoding token classes via final transitions.

Interestingly enough, the DFSA does not accept any string over the original alphabet Σ because all states reachable via characters $a \in \Sigma$ are non-final. In order to get to the final state q_{fin}, a sentinel character must be consumed. This is the expected behaviour because each of the patterns in the regexp $R_1\$_1|\ldots|R_n\$_n$ ends in a sentinel character.

Furthermore, suppose that some prefix u of the input string matches one of the original patterns R_i. This is equivalent to $u\$_i$ matching the pattern $R_i\$_i$. In other words, u matches a pattern R_i if and only if (a) $q = \delta(q_0, u)$ is defined, and (b) $\delta(q, \$_i) \in F$. Thus, if the automaton has reached a state q, the matching patterns can be identified by checking for transitions leaving q and labelled with sentinel characters. Such arcs are often called *final transitions*.

In figure 3.4, the final transitions indicate a match of the *upper-case initial* pattern whenever the automaton enters state q_1. Similarly, q_2 corresponds to *lower-case initial*, q_3 to *upper-case word*, q_4 to *capitalized word* and q_5 to *lower-case word*.

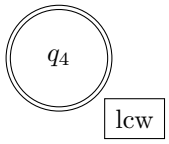
Encoding

If the tokenization procedure presented in algorithm 3.1.1 is applied to the DFSA shown in figure 3.4, it will not work because none of the states $q_1, \ldots q_5$, corresponding to the token patterns, is final — a condition for the function LONGESTMATCH() to report a token.

This incompatibility can be eliminated easily by:

- making the function *TokenClass*(q) return the token class of the sentinel character corresponding to the final transition leaving q, or **nil** if no such final transition exists;
- making final each state q such that *TokenClass*$(q) \neq$ **nil**;
- deleting q_{fin} and the final transitions, which are no longer needed since the information they express is already encoded in function *TokenClass*(q).

For better readability, final transitions will henceforth be represented not by arcs, but boxes attached to state labels:



Non-Disjoint Patterns

The token classes defined in the above examples are disjoint, i.e., each string belongs to at most one token class. Accordingly, the construction shown in figure 3.4 produces a DFSA in which, for each $q \in Q$, there is at most one final transition originating in q.

However, the construction can also deal with non-disjoint patterns. For example, consider the patterns `a`, `a(a|b)` and `a*`. Obviously, the string a matches the first and the third pattern, while aa matches the second and the third one. The compilation of the three patterns into a DFSA with final transitions produces the structure shown in figure 3.5.

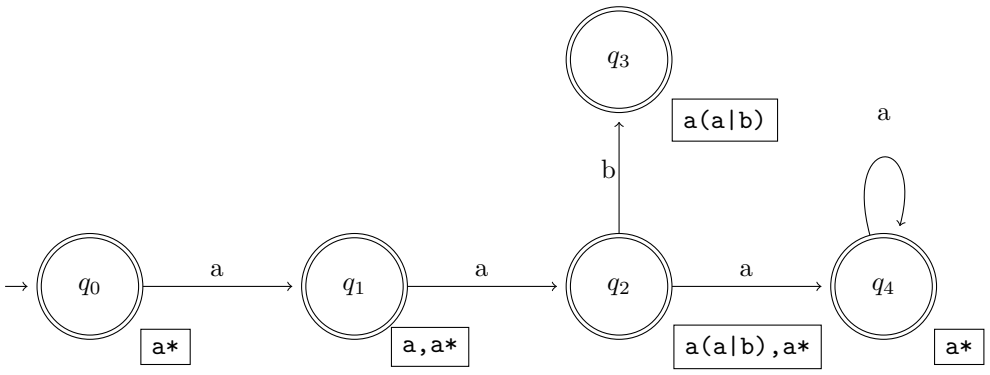


Figure 3.5: A DFSA with final transitions recognizing the non-disjoint patterns `a`, `a*` and `a(a|b)`. Note that states q_1 and q_2 correspond to the match of more than one pattern.

Note that the states q_1 and q_2 have more than one final transition associated with them, which indicates that the strings leading to these states match more than one pattern. In other words, the construction of a DFSA with final emissions permits *simultaneous* matching of several patterns.

This idea can be used to simplify token class definitions. So far, we have made sure that these definitions are disjoint, but this may be difficult. For example, the regexp definitions of *lower-case word*, *upper-case word* and *capitalized word* given above leave mixed-case words such as *WordPerfect* unaccounted for. Thus, a definition of the token class *mixed-case word* is required.

If we insist on this definition being disjoint from all the other token regexps, its formulation is very cumbersome and not intuitive at all:

```
[[:alpha:]]*[[:lower:]][[:upper:]][[:alpha:]]*|
[[:alpha:]]*[[:upper:]][[:upper:]][[:lower:]][[:alpha:]]*
```

Instead, it would be much easier to define the *mixed-case word* class as a default: `[[:alpha:]]+` *unless* the token matches any of the other three *word* or *initial* token classes. In this way, we no longer require the token class definitions to be disjoint, but state that potential conflicts are resolved according to the order in which the rules are formulated.

Lower-case initial (LCI): `[[:lower:]]`

Upper-case initial (UCI): `[[:upper:]]`

Lower-case word (LCW): `[[:lower:]]+`

Upper-case word (UCW): `[[:upper:]]+`

Capitalized word (CW): `[[:lower:]]+`

Mixed-case word (MCW): `[[:alpha:]]+`

The compilation of the union of the above regular expressions into a DFSA produces the structure shown in figure 3.6.

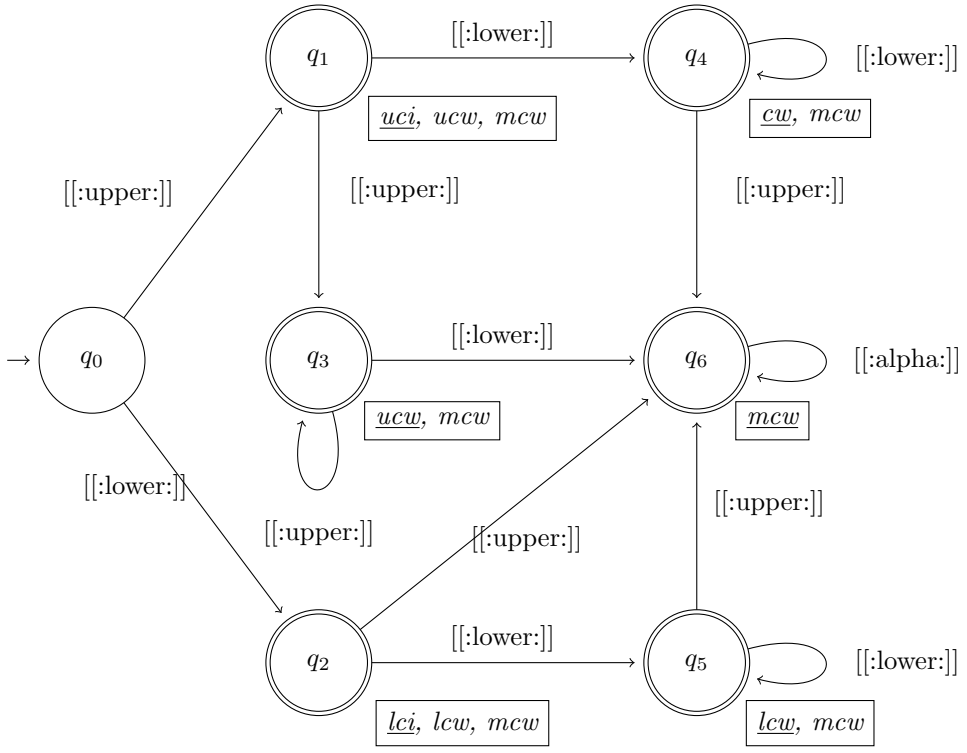


Figure 3.6: Tokenizer DFSA defined using prioritized non-disjunctive regular expressions.

Several final states are associated with multiple final transitions. For instance, q_1, reachable from q_0 via a single upper-case character, has 3 final transitions indicating the match of the patterns *lower-case initial*, *lower-case word* and *mixed-case word*. It is here that the prioritization of the patterns comes into play: *lower-case initial* comes before the other token regexp definitions, hence *lower-case initial* becomes the token class associated with q_1.

The same happens to the other states: the pattern that comes first in order of precedence is kept as the actual token class while the other ones are removed. In the state diagram above, the final transitions which are kept are underlined.

The token category *mixed-case word* is the last one in order of priority, hence it becomes the token class label of one state only, namely q_6. The reader may check that all strings leading to q_6 are mixed-case words, indeed, as well as that all mixed-case words lead to q_6.

If we assume that the patterns $R_1|\ldots|R_n$ are numbered in ordered of decreasing priority then the function *TokenClass* required by the tokenization algorithm is defined as follows:

$$TokenClass[q] \quad = \quad \begin{cases} min(FinalTr[q]) & \text{if } FinalTr[q] \neq \emptyset \\ \textbf{nil} & \text{otherwise} \end{cases}$$

where $FinalTr[q]$ denotes the set of all patterns for which a final transition starting at q exists.

The mechanism of final transitions is a commonly used technique. We will see it in other applications, too.

3.2 Pattern Matching

3.2.1 Finding Patterns in Time $O(|w|)$

In the section on tokenization, we saw how a text can be split into a contiguous sequence of substrings corresponding to consecutive pattern matches. A related but different problem occurs in search applications, where one wants to identify all occurrences of a pattern (e.g. an e-mail address) in a text. These occurrences are typically non-contiguous and interspersed with possibly large chunks of text not matching the pattern.

Even if we can formulate the pattern as a regular expression and compile it into a DFSA, this does not provide us with a device for efficient search. Consider a string $w = a_1 \ldots a_t$ in which $v = a_k \ldots a_l$ is the first match of a pattern R.

$$w = a_1 a_2 \ldots a_{k-1} \overbrace{a_k \ldots a_l}^{v \in \mathcal{L}(R)} a_{l+1} \ldots a_t$$

If R is compiled into a DFSA, the resulting automaton A_R can tell that v matches R, but we need to find out that it needs to be applied at string position k. This can be done in a naïve way by running A_R on the string $a_i \ldots a_t$ for $i = 1, \ldots, t-1$.

Algorithm 3.2.1: TrialAndFailureLocatePattern(*Text*)

```
for k ← 1 to |Text|
    q ← q0
    for l ← k to |Text|
        if δ(q, Text[l]) undefined
            break
        q ← δ(q, Text[l])
        if q ∈ F
            report pattern match at position (k, l)
```

Eventually, all matches of R can be found in this way, but the automaton may have to traverse the whole suffix $a_i \ldots a_t$ for each i, even if no matches are found. The total number of transitions looked up in this worst-case situation is $\Sigma_{i=1}^{|w|} i = \frac{|w|(|w|+1)}{2} = O(|w|^2)$. Since $|w|$ may be very large for typical text search applications, this solution is not satisfactory.

Instead, the problem may be reduced to matching the whole of w against a single regular expression. Observe that the prefix of w preceding v matches Σ^*:

$$w = \overbrace{a_1 a_2 \ldots a_{k-1}}^{\Sigma^*} \overbrace{a_k \ldots a_l}^{v \in \mathcal{L}(R)} a_{l+1} \ldots a_t$$

Hence, searching for v is equivalent to matching w against the regexp $\Sigma^* R$. For this, we can compile $\Sigma^* R$ into a DFSA, and run this DFSA on w. Whenever the automaton enters a final state, we know we have just finished reading a match of R.

Algorithm 3.2.2: LOCATEPATTERN(*Text*)

```
q ← q_0
for i ← 1 to |Text|
    q ← δ(q, Text[i])
    if q ∈ F
        report end of pattern at position i
```

This search strategy requires only one deterministic pass of the input string. Hence, it makes it possible to identify all occurrences of a regular pattern in time $O(|w|)$.

The algorithm can be generalized in two ways, as discussed below.

Locating the Start and the End of a Pattern. Note that the algorithm only reports the end of each match, but not its beginning. If we are more interested in the start positions of all matches, the solution is symmetric. We need to match the regexp $R\Sigma^*$ against *suffixes* vz of w. In order to be able to do so, we compile $R\Sigma^*$ into an FSA, then reverse and determinize it, and finally run the reverse DFSA from left to right on w.

If both the beginning and the end of each match are required (e.g. in order to replace the matching substring by something else), we need to keep two DFSAs: one for $\Sigma^* R$ and one for the reverse regular expression R^{-1}. As soon as a match of $\Sigma^* R$ has been reported for a prefix w' of w, we can run the acceptor for R^{-1} on w' from right to left in order to determine the starting point(s) of the match(es) v such that $w' = uv$. The worst-case complexity of this matching method is quadratic.

The task is easier if R is a fixed-length pattern, e.g. `[Cc].[Tt]`, or when we look for a word. In such a case, the start of the match is determined by subtracting the length of the pattern from the length of w'. A separate automaton for R^{-1} is obviously not required.

Search for Multiple Patterns. The algorithm outlined above also allows for the simultaneous matching of a collection of regular patterns $R_1, \ldots, R_n$. Instead of $\Sigma^* R$, the input string w must be matched against $\Sigma^*(R_1|\ldots|R_n)$. If, in addition to locating the pattern matches, one is interested in actually identifying the matching pattern R_i for each match, each pattern can be associated with a separate *final transition* $\$_i$, as shown in section 3.1.2. The input string is then matched against the pattern $\Sigma^* R_1\$_1|\ldots|R_n\$_n$.

3.2.2 Failure Function

The construction of a DFSA for the regexp $\Sigma^* R$ involves the determinization of the NFSA created by regexp compilation. Since the worst-case running time of the subset construction algorithm is exponential in the number of states of the NFSA, the construction may become very slow. In addition, the DFSA may also become very big, especially if the pattern R is complex and the input alphabet $|\Sigma|$ is very large.

The Aho-Corasick string matching algorithm provides a faster way of constructing a DFSA for a set of patterns. Also, the DFSA is smaller at the expense of having a slightly more complex encoding of the transition function. In its original version, the algorithm searches for occurrences of some strings $u_1, \ldots, u_n$ rather than general-case regular patterns. A generalisation to arbitrary patterns is possible, but we will concentate on the simpler problem. The solution required the introduction of a new data structure, called a *trie*, and a space-efficient encoding of the transition function. These two issues are discussed separately in the following two sections.

Data Structures: Tries

Suppose we want to be able to locate occurrences of the strings *a*, *abbb*, *ba* and *bb* in texts. The first step is to construct a DFSA for the language $\{a, abbb, ba, bb\}$. In order to do this *quickly*, we directly

construct a DFSA rather than follow the standard two-step compilation method, which first turns a regular expression into an NFSA and then determinizes it. More precisely, we construct a data structure called a *trie*, i.e. a tree-shaped DFSA shown in figure 3.7.

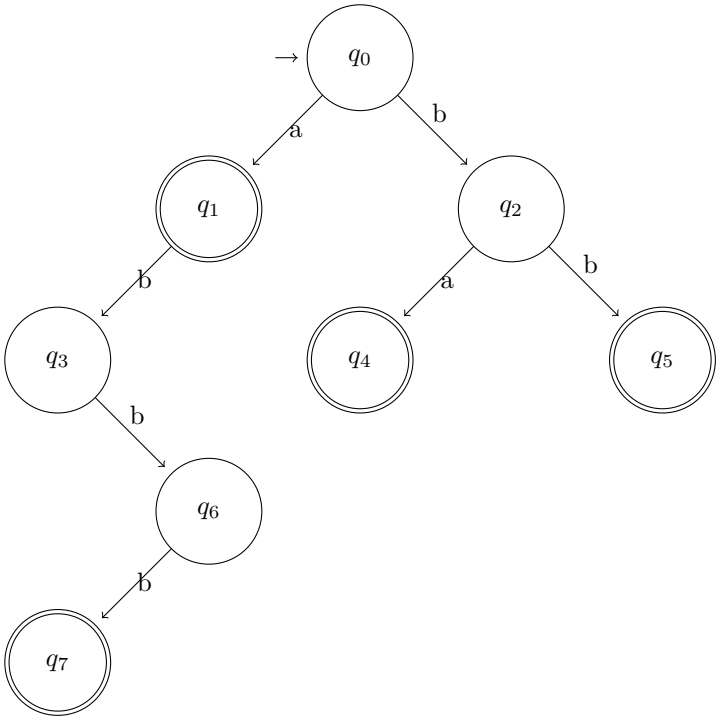


Figure 3.7: A trie encoding the language $\{a, abbb, ba, bb\}$.

Each state in the trie can be reached by exactly one string consumed by the automaton: q_0 by ϵ, q_1 by a, q_2 by b, etc. We call such a string the *key* of state q and denote it $key[q]$. A new string u can be inserted into a trie $A = (\Sigma, Q, q_0, F, \delta)$ by (a) identifying the longest prefix x of u such that $q = \delta(q_0, x)$ is defined, (b) appending a chain of states consuming the remainder of u to q, so that $\delta(q_0, u)$ is defined, and (c) making $\delta(q_0, u)$ final.

Figure 3.8 shows how the trie shown in figure 3.7 is created out of a trie for $\{a, ba, bb\}$ by inserting the string *abbb* into it. The original trie is marked in grey, the new states and transitions in black. The string a, consumed by A along the path q_0, q_1, is identified as the longest common prefix of *abbb* and A. The remainder *bbb* is accepted along the path q_1, q_3, q_6, q_7, where q_3 and q_6 and q_7 are new states (corresponding to the strings *ab*, *abb* and *abbb*, respectively). The state q_7 is added to F in order for *abbb* to be accepted.

The pseudocode of the insertion operation is presented in algorithm 3.2.3. *Word* is the word bing inserted into the automaton. The **while** loop traverses the prefix of *Word* that is already in the automaton. At the beginning of the i-th iteration of the loop, the variable q holds the value of $\delta(q_0, Word[1 \ldots i-1])$. Then $\delta(q_0, Word[1 \ldots i]) = \delta(q, Word[i])$ is determined. If it is not defined, we have arrived at the end of the prefix, in which case the procedure AddBranch() is called, which inserts a chain of states that accept the remainder of *Word*. AddBranch() returns the last state in the chain, which is made final in the last step.

The main **while** loop and the **for** loop in AddBranch() execute exactly $|Word|$ times. If we assume that the running time of the operations performed inside the loops (transition lookup, state allocation and the insertion of a new state into Q) is independent of the size of A, then the

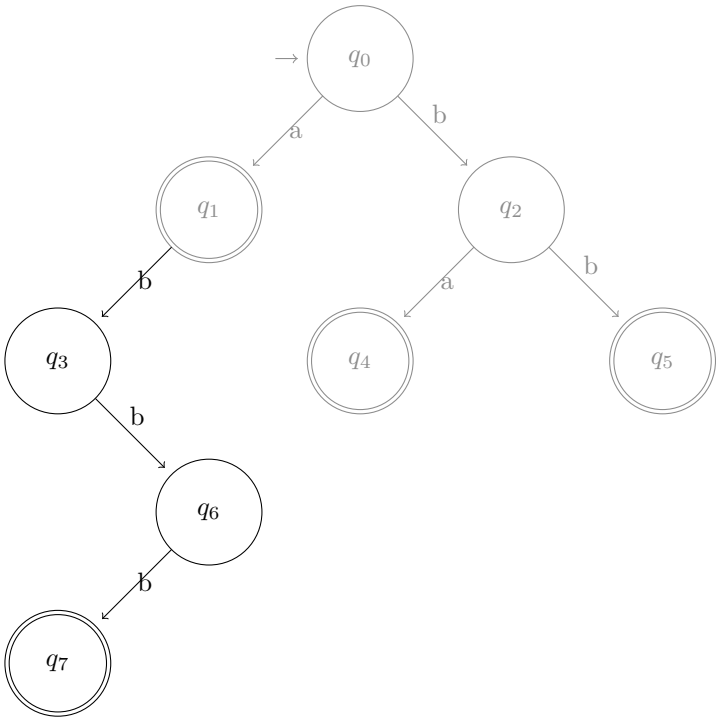


Figure 3.8: Insertion of the string *abbb* into a trie encoding the language $\{a, ba, bb\}$.

insertion algorithm runs in time $O(|\,Word|)$.

Algorithm 3.2.3: TRIEINSERT($A = (\Sigma, Q, q_0, F, \delta)$, *Word*)

$q \leftarrow q_0$
$i \leftarrow 1$
while $\delta(q,\, Word[i])$ is defined
  $q \leftarrow \delta(q,\, Word[i])$
  $i \leftarrow i + 1$
$q \leftarrow$ ADDBRANCH(q, $Word[i...|\,Word|]$)
if $q \notin F$
  $F \leftarrow F \cup \{q\}$

procedure ADDBRANCH(q, w)
 for $i \leftarrow 1$ **to** $|w|$
   $q_{new} \leftarrow$ ALLOCATENEWSTATE()
   $Q \leftarrow Q \cup \{q_{new}\}$
   $\delta(q, w[i]) \leftarrow q_{new}$
   $q \leftarrow q_{new}$
 return (q)

A trie for a language $\{u_1, \ldots, u_n\}$ can be constructed by successively calling TRIEINSERT(A, u_i), starting with an empty automaton containing only the initial state q_0. The total running time of the construction algorithm is $O(\Sigma_{i=1}^{n}|u_i|)$. As a result, a trie can be constructed without the exponential blow-up associated with the subset construction algorithm.

Search

Having constructed a trie for the strings $u_1, \ldots, u_n$, we can use it to locate their occurrences in an input string $w = a_1 \ldots a_t$. As in section 3.2.1, we start with the idea of first finding all u_i starting at a_1, then all u_i starting at a_2, etc. This can be done by running the trie DFSA on the suffixes $a_k \ldots a_t$ of w for $k = 1, \ldots, t$. Whenever the automaton enters a final state after consuming the string $a_k \ldots a_l$, the substring $a_k \ldots a_l$ is reported as one of the words we are searching for.

For example, consider the application of the trie encoding the words $\{a, abbb, ba, bb\}$, shown in figure 3.7, to the string *abbab*.

In the first iteration, the search algorithm tries to accept a prefix of the original string $w =$ *abbab*. A match of the pattern a (final state q_1) is obviously reported after consuming the first character of w. After consuming two more characters, the automaton ends up in state q_6. From there, there is no transition labeled with the next character a. This means that no more patterns encoded in the trie start at the first letter of the input string.

In the next step, we check if any of the words we are looking for start at the second character of w. This is equivalent of going back to state q_0 and running the automaton on the suffix *bbab*. This time, the automaton consumes the prefix *bb* and enters state q_5. This state is final, and therefore a match of the pattern *bb* is reported. The next character (a) cannot be accepted in q_5, so search restarts at the next string position, corresponding to the suffix *bab* of w.

The whole process is shown in figure 3.9. Dashed arcs denote the backtracking steps the DFSA has to take in order to move back to the next start position in the input string.

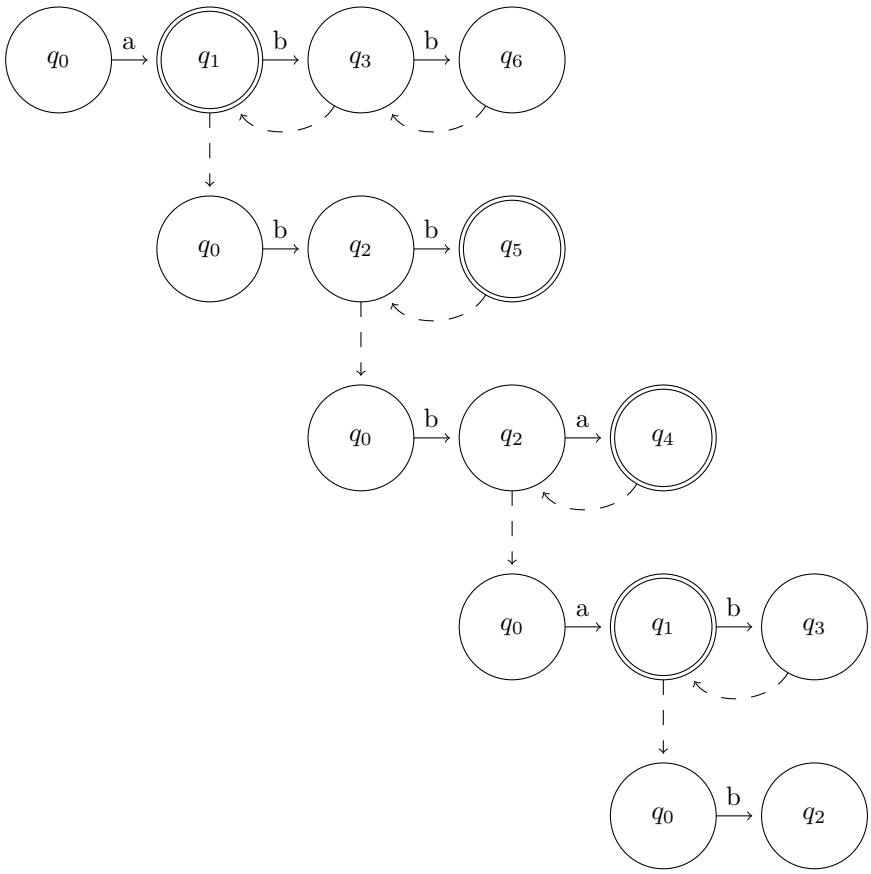


Figure 3.9: The application of naïve backtracking search in order to locate occurrences of words encoded in the trie shown in figure 3.7 in string *abbab*.

For each suffix $a_k \dots a_t$ of w, the trie automaton performs at most as many transitions as there are characters in the longest word u_i, a number we denote by L (formally, $L = max\{|u_i| : i = 1, \dots, n\}$). Since k ranges over $1, \dots, t$, the total running time of the algorithm is $O(L \cdot |w|)$.

This trie-based search technique is basically a version of algorithm 3.2.1 (TrialAndFailure-LocatePattern(), described in section 3.2.1), restricted to acyclic patterns. This restriction improves the running time from $O(|w|^2)$ to $O(L \cdot |w|)$, but the factor L still depends on the words we are looking for. The algorithm is obviously slower than the one based on running a DFSA for the pattern $\Sigma^* R$ on the input string: the latter does not involve any backtracking. In the following, we shall see how to avoid backtracking and perform a deterministic search using a trie.

Recall that each state $q \in Q$ in the trie uniquely identifies the string $key[q] = a_1 \dots a_t$ that leads from q_0 to q: $\delta(q_0, key[q]) = q$. On the other hand, we know that after the next backtracking step, the algorithm will have to process the characters $a_2 \dots a_t$, so they actually do not need to be re-read.

For example, if $q = q_6$, then $key[q] = abb$ and the string to be consumed after the next backtracking step starts with bb, which leads to state $\delta(q_0, bb) = q_5$. Thus, if we precompute the values of $\delta(q_0, v)$ for prefixes v of the patterns, the backtracking step can be replaced by a "shortcut" transition to the state reachable by the respective prefix. Such transitions are called *failure transitions*.

Figure 3.10 demonstrates how failure transitions (the vertical arcs) may be used to replace backtracking steps (marked in grey). It is obvious that they save a lot of processing time: now there are only 9 transitions instead of 25.

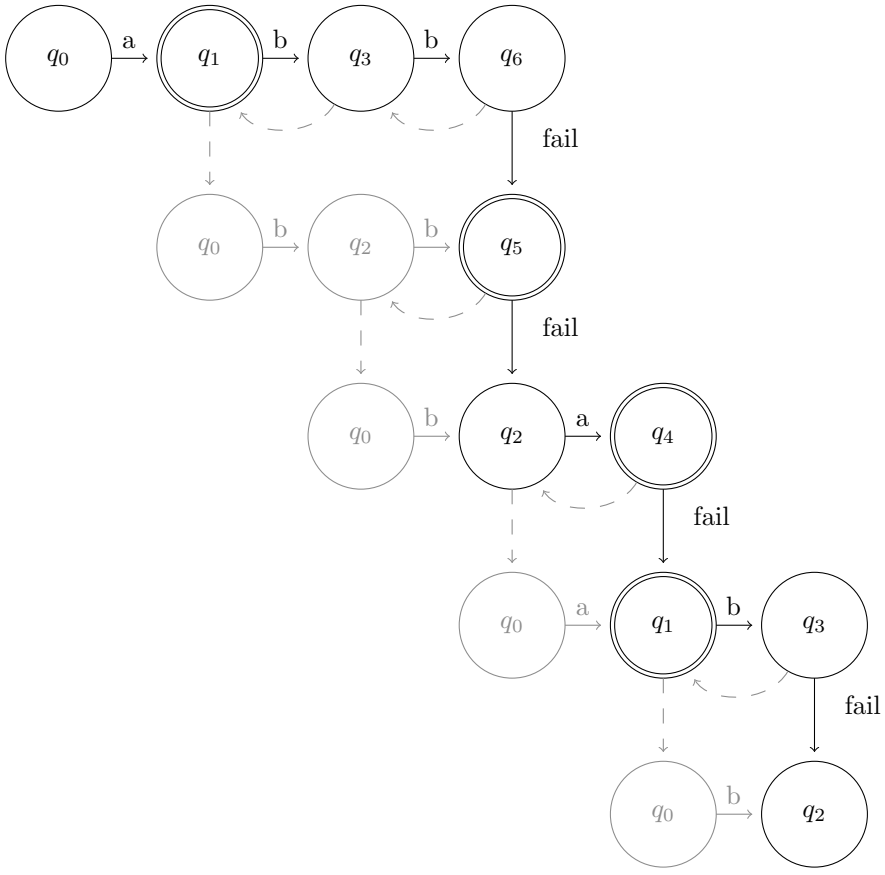


Figure 3.10: The application of the Aho-Corasick algorithm to the problem of locating occurrences of the string patterns $a, abbb, ba, bb$ in string $abbab$. The nodes and transitions actually traversed by the algorithm are drawn in black. The nodes and transitions additionally traversed by the naïve backtracking algorithm are marked in grey.

The failure transitions are best represented by a function $fail : Q \to Q$, so we can write $fail[q_6] = q_5$, $fail[q_5] = q_2$, etc.

Note that the failure function sometimes has to model the result of multiple backtracking steps. For example, the application to the string *abbbb*, shown in figure 3.11, involves the failure transition from state q_7 to q_5, which stands for *two* backtracking steps.

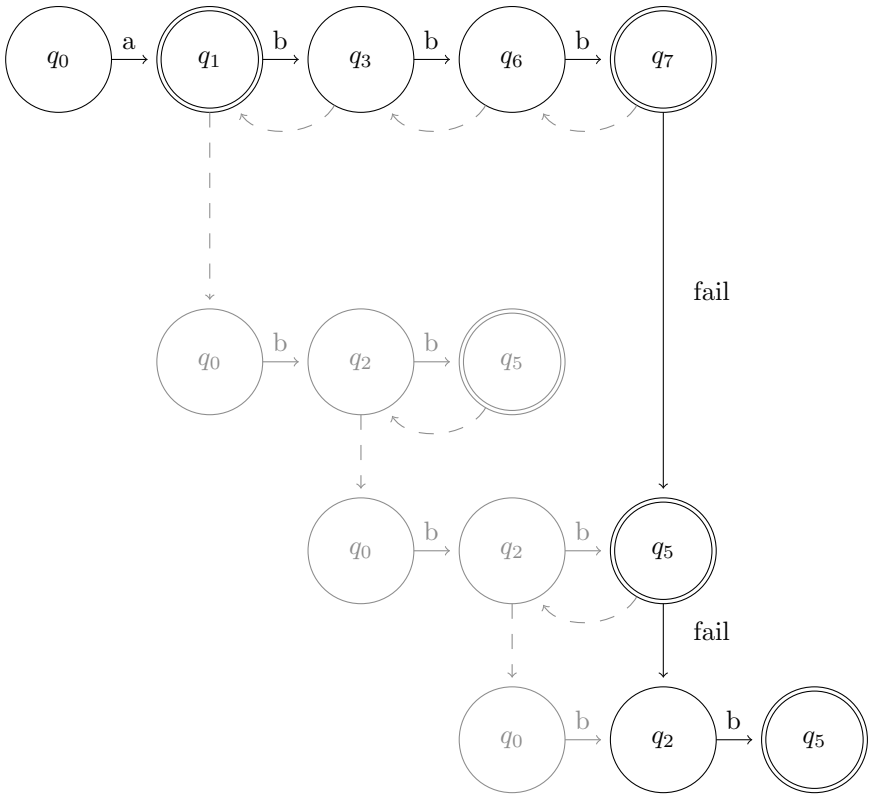


Figure 3.11: The application of the Aho-Corasick algorithm to the string *abbbb* using the trie shown in figure 3.7. The transitions required by the naïve $O(L \cdot |w|)$ search algorithm are marked in grey. Note that the first failure transition stands for two backtracking steps.

Thus, if $key[q] = a_1 \dots a_t$ is the string leading to state q, then the value of $fail[q]$ is the state we arrive at by successively applying the backtracking search to the suffix $a_2 \dots a_t$.

Also note that the state $fail[q]$ sometimes does not have a transition labeled with the current symbol: e.g. $fail[q_7] = q_5$, but $\delta(q_5, b)$ is not defined. The algorithm then backs off to $fail[q_5] = fail[fail[q_7]]$. In the general case, we need to inspect $fail[q]$, $fail[fail[q]]$, etc., until a transition labeled with the current symbol is found. Procedure NextState(q, a) (algorithm 3.2.4) implements the recursive application of the failure function in order to determine the state that can be reached from q by consuming symbol a.

Algorithm 3.2.4: NextState(q, a)

```
while q ≠ q0 and δ(q, a) undefined
   q ← fail[q]
if δ(q, a) defined
  return (δ(q, a))
 else return (nil)
```

Note that the length of $key[q]$ decreases in each iteration because we are trying shorter and

shorter suffixes. Therefore, if the condition that $\delta(q, a)$ be defined is not satisfied, the algorithm reaches $fail[\ldots fail[q]\ldots] = q_0$. Then, the procedure returns **nil**. This may happen, e.g., if we encounter a symbol not used in any of the patterns u_i, such as the letter c. In such a case, the symbol in question is skipped and search re-starts at the next string position in state q_0.

The actual top-level search algorithm 3.2.5 is a slightly modified version of algorithm 3.2.2, the only difference being that the call NextState(q, a) replaces the transition lookup $\delta(q, a)$. Since NextState(q, a) may return **nil**, this special value is handled by the DFSA skipping one character and jumping back to q_0.

Algorithm 3.2.5: LocatePattern(*Text*)

```
q ← q0
for i ← 1 to |Text|
   q ← NextState(q, Text[i])
   if q = nil
     q ← q0
    else if q ∈ F
      report pattern match at position i
```

The algorithm relies on final states being indicators of pattern matches. This assumption may require an extension of the original set F, as discussed in the following section.

Reporting Pattern Matches

The search strategy devised so far makes it possible to deterministically emulate the naïve backtracking search strategy by taking shortcuts called *failure transitions*. Recall that each failure transition replaces a path traversed by the naïve algorithm. For example, the failure transition from q_7 to q_5 in figure 3.11 replaces the path $q_7, q_0, q_2, q_5, q_0, q_2, q_5$. Now observe that the path contains the final state q_5, whose first occurrence indicates a match of the string pattern *bb* after consuming the prefix *abb*. Since the path is no longer traversed, this piece of information must be stored in another form: by marking the state q_6, reached by the algorithm after consuming the prefix *abb*, as final. Otherwise, algorithm 3.2.5 will not report a match.

In general, the Aho-Corasick algorithm can enter a state $q \in Q$ only if the string consumed so far ends in $key[q]$. Even if $q \notin F$, there may be a suffix v of $key[q]$ such that $\delta(q_0, v) \in F$. In other words, v is one of the patterns u_i we are looking for. As a result, we must mark as final all states q for which some u_i is a suffix of $key[q]$.

Interestingly, the above process may be performed solely based on the values of the failure function. If we align the paths of all suffixes of some string u accepted by the trie automaton, then we have the situation shown in figure 3.12, where $\delta(q_0, bb)$, $\delta(q_0, b)$ and $\delta(q_0, \epsilon)$ are shown in the column below state $\delta(q_0, abb) = q_6$.

Obviously, these are the states that are reached by backtracking from $\delta(q_0, abb) = q_6$ and consuming the next suffix of *abb*, in other words — the successive values of the failure function: $fail[q_6]$, $fail[fail[q_6]]$, $fail[fail[fail[q_6]]]$, ... The set of such values can be called the *failure closure* of q_6 and denoted $fail^*[q_6]$.

Therefore, instead of checking if $\delta(q_0, v) \in F$ for each prefix of *abb*, we may just inspect the set $fail^*[q_6]$: since $fail^*[q_6] \cap F \neq \emptyset$, we must add q_6 to F.

The benefits of this perspective may not appear obvious at first, but will become clear when we show how to incrementally construct the failure function in the following subsection.

Figure 3.13 shows the trie from figure 3.7 with the values of the failure function marked for each state (each node is labeled with the respective state symbol q_i followed by the value of the failure function: $q_i/fail[q_i]$). Note that state q_6 is now marked as final.

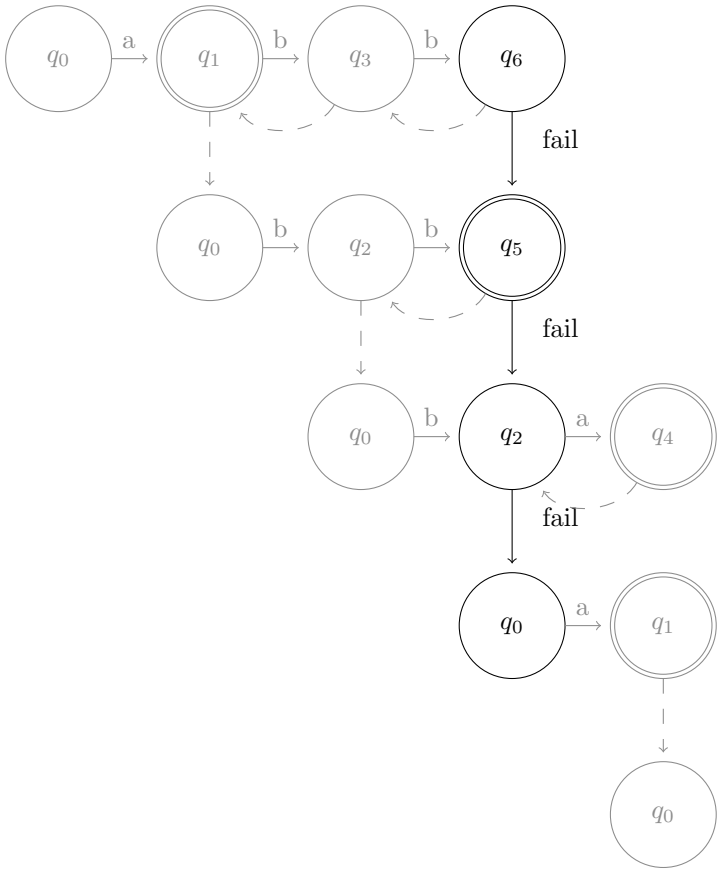


Figure 3.12: Alignment of the string *abb* and the paths for all prefixes of *abb*.

Computation of the Failure Function

In order to compute the failure function efficiently, assume that it is computed in increasing order of $|key[q]|$, starting with q_0. We set $fail[q_0] =$ **nil** as there is no state to backtrack to from the initial state. Similarly, if the search fails in q_1 or q_2, we can only backtrack to q_0. Hence, $fail[q_1] = fail[q_2] = q_0$.

Now suppose we want to determine $fail[q_7]$ (figure 3.14). Since $key[q_7] = abbb$, one possibility would be to process the suffix *bbb* using the naïve backtracking strategy, which involves backtracking 3 characters, moving back to q_0, then following the path q_7, q_0, q_2, q_5, backtracking and moving to q_0 again, and finally following the path q_0, q_2, q_5.

However, since $fail[q]$ has already been determined for all q such that $|key[q]| < |key[q_7]|$, we can simply backtrack to state q_6, and then take a shortcut to $fail[q_6] = q_5$ (instead of going $q_6 \to q_3 \to q_1 \to q_0 \to q_2 \to q_5$) and check if $\delta(q_5, b)$ is defined. Since it is not, we take another shortcut, going from q_5 to $fail[q_5] = q_2$ (instead of going $q_5 \to q_2 \to q_0 \to q_2$). This time, there is a transition consuming b: $\delta(q_2, b) = q_5$. Hence, we set $fail[q_7] = q_5$.

The recursive application of the failure function starting in q_6 is nothing else than a call to NextState(q_6, b).[2] As a result, $fail[\delta(q, s)] =$NextState(q, s) for any pair $q \in Q$, $s \in \Sigma$, and

[2]Note that NextState(q_6, b) makes use of the failure function, which is being constructed. There is no risk of looking up a still undefined value $fail[q]$ because — as already mentioned — NextState(q_6, b) considers states in *decreasing* order of $|key[q]|$.

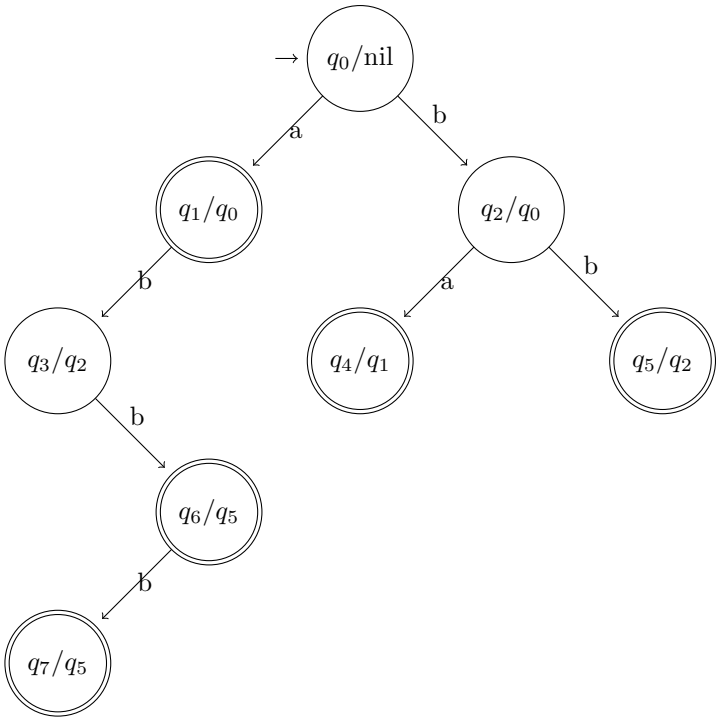


Figure 3.13: A trie encoding the language $\{a, abbb, ba, bb\}$ with the values of the failure function.

the computation of the failure function can be stated as follows.

Algorithm 3.2.6: ComputeFailureFunction$(\Sigma, Q, q_0, F, \delta)$

$fail[q_0] \leftarrow$ **nil**
Enqueue$(q_0, Queue)$
while $Queue \neq \emptyset$
  $q \leftarrow$ Dequeue$(Queue)$
  for each $s \in \Sigma$ such that $\delta(q, s)$ is defined
    $fail[\delta(q, s)] \leftarrow$ NextState(q, s)
    Enqueue$(\delta(q, s), Queue)$
    if NextState$(q, s) \in F$
      $F \leftarrow F \cup \{\delta(q, s)\}$

The algorithm uses a queue of states in order to explore the trie in a breadth-first manner so that the states $q \in Q$ are considered in ascending order of $key[q]$. For each state q, the algorithm visits all states r reachable from q via a single symbol s, and sets $fail[r] \leftarrow$ NextState(q, s).

Also note that whenever NextState(q, s) is final, the state $r = \delta(q, s)$ is made final itself. This makes sure that $p \in F$ for each state p such that $fail^*[p] \cap F \neq \emptyset$ (cf. the previous subsection).

Space Savings

Apart from fast construction, the greatest advantage of the failure function construct used in the Aho-Corasick algorithm is its space-efficiency. Note that the transition function δ does not encode all the transitions of the automaton: some of them are represented implicitly by the failure function and computed by NextState(q, s). In this sense, NextState(q, s) may be viewed as the actual transition function of the automaton, while δ becomes only an auxiliary structure.

The space savings are significant. The trie shown in figure 3.13 contains 8 states and 7 transitions. In addition, space for 8 values of the failure function (one per state) is needed. In contrast

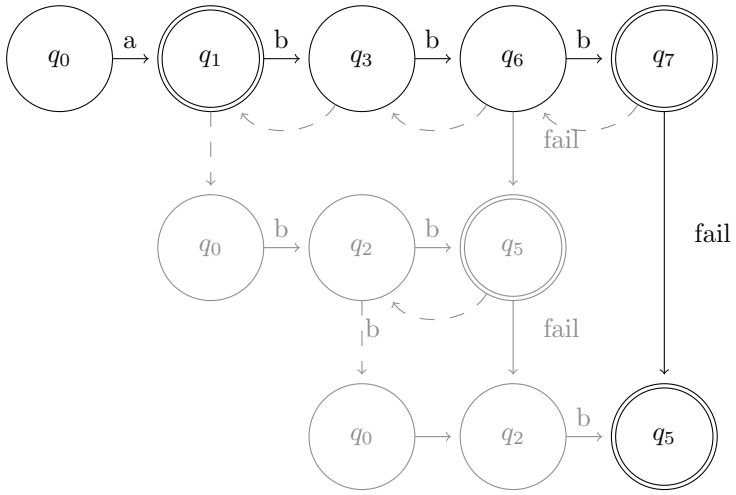


Figure 3.14: Computation of $fail[q_7] \leftarrow q_5$. The values $fail[q_7] = q_5$ and $fail[q_5] = q_2$ are assumed to have been determined before, and can replace the backtracking steps and paths they stand for.

to that, a conventional transition function would have to encode *all* $|Q| \cdot |\Sigma| = 24$ transitions. Moreover, the size of the failure function DFSA does not depend on the size of the alphabet while adding a new symbol to Σ adds $|Q|$ transitions to a conventional transition table.

3.3 Dictionaries

Dictionaries are necessary components of most NLP applications. They come in a variety of types, ranging from simple word lists to semantic lexicons. They are often very large: broad-coverage dictionaries of English contain hundreds of thousands of words; this number may easily increase to millions if proper names are included. As a result, compact storage of and fast access to lexical data are required. Finite-state machines provide us with a formal framework meeting both requirements. In section 3.3.1 we show how to store a dictionary as a finite-state automaton in form of a trie. This guarantees fast access. A dictionary trie can be minimized (section 3.3.2). In section 3.3.3, we introduce algorithms for the incremental construction of minimal automata. We finish with a discussion on how to associate additional information with words in a dictionary.

3.3.1 Dictionary Tries

Suppose we have a list of 2 000 000 English lowercase words. If the average word length is, say, 5.5 letters (bytes), we need 13 000 000 bytes to store them as text (in case you wonder why we need 6.5 bytes for each word, remember that we need one character to separate them). Pure text is not a good data structure for search. If we add a vector of 4-byte pointers, each pointing to the beginning of a word, we will be able to use binary search (see box on page 80) to find words, and our dictionary will grow to 19 000 000 bytes. This is not much for modern computers, but our dictionary is only a word list; syntactic or semantic dictionaries can be much larger.

In order to save space, we may consider dividing the dictionary into 26 sublists, one per character and holding words beginning with that character, as shown in figure 3.15.

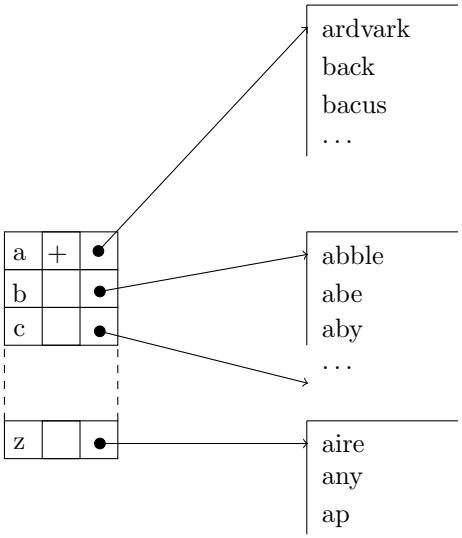


Figure 3.15: Division of a list of words into sublists. Each sublist contains words beginning with the same letter. The letter itself is removed from the words and stored in a pointer. The sign + indicates that the letter leading to a sublist is a full word.

Binary search

Binary search is an algorithm for searching a set of sorted data for a particular value which takes advantage of the characteristic of data being sorted in advance. An important prerequisite is random access to the data being searched.

The idea behind binary search is to repeatedly eliminate half of the search interval in each step of the algorithm until the searched value is found (or eventually not found). Let us consider as an example searching a sorted array of length n. We begin searching with an interval covering the whole array. If the value we are searching for matches the value of the middle element in the array, we are done. If, however, the value of the search key is less than the item in the middle of the interval, we narrow the interval to the lower half. Otherwise we narrow it to the upper half. Subsequently, we carry out the same procedure repeatedly until the value is found or the interval is empty, i.e., the search key has not been found.

The runtime complexity of binary search on a sorted list of length n is $O(logn)$ since half of the list is eliminated with each comparison and after performing $logn+1$ comparisons the search interval will be of length less than or equal to one. Below we give a step by step example of searching for the key 6 in the sorted array 1 3 5 6 7 12 23 34 35 43 50.

STEP	SEARCH SPACE	ACTION
1	1 3 5 6 7 <u>12</u> 23 34 35 43 50	SEARCH LEFT INTERVAL
2	1 3 <u>5</u> 6 7	SEARCH RIGHT INTERVAL
3	<u>6</u> 7	FOUND — TERMINATE

Please note that in the third step the length of the subarray to be searched is of even length. In such a case we choose without loss of genericity the first element in the second half of the array as the 'middle' element.

Since all words in every sublist begin with the same letter, we do not need to store it with each word; we can store only subsequent letters. Our dictionary is now a list of 26 pointers, and a list of 26 sublists of words that are stored without their first letter. As there may be one-letter words, like *a*, a flag (here marked by +; at most one byte) must be associated with each sublist to indicate whether this is the case for a particular letter. The new data structure is stored on $2\,000\,000 - 26 \times (4 + 1) = 1\,999\,870$ fewer bytes, or 10% smaller, than the original one. The time needed to find a particular word is also shorter. By examining a single letter, we reduce the search

space by a factor of 26.

Encouraged by our initial success, we can repeat the process on the sublists. Then we can move to their sublists, and so on. In this way, we create a hierarchical structure such as the one shown in figure 3.16.

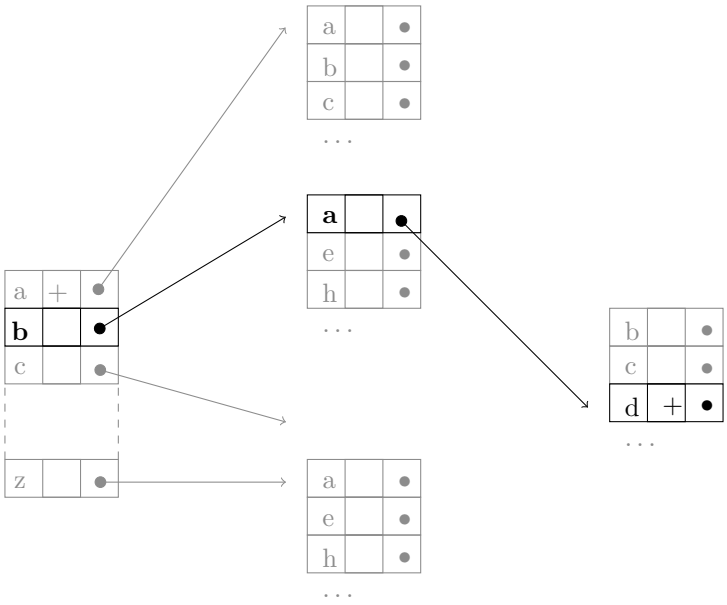


Figure 3.16: The result of hierarchical division of a word list into sublists according to the n-th character, $n = 1, 2, \ldots$. A word is found by looking up its consecutive characters in the respective pointer lists. The highlighted search path corresponds to the word *bad*.

In order to look up a word in such a structure, we go to the pointer list at the root and follow the pointer associated with the first character of the word. The pointer points to another pointer list, in which the second character of the word is looked up. Finally, if the word is in the dictionary, we arrive at the pointer list corresponding to the last character and check the extra bit, which must be set in this case.

The tree-shaped structure constructed in this way is nothing else than a *trie*, defined in section 3.2.2. If each of the pointers is thought of as a transition (labeled with the letter associated with the pointer), then the connection to finite-state automata becomes apparent: by successively dividing a word list into lists of pointers (transitions), we have arrived at an FSA encoding method for dictionaries.

Note that the lookup time for a word has become independent of the size of the dictionary: for each letter of the word, we look it up in the appropriate pointer list, which may have at most $|\Sigma|$ entries. Depending on the encoding of the list, it may take from $O(1)$ to $O(|\Sigma|)$ steps. Thus, the lookup time for a word w is bounded by $O(|\Sigma| \cdot |w|)$. In other words, the trie encoding of a dictionary guarantees fast lookup.

The other objective, namely reducing the size of the dictionary, turns out to be more difficult to achieve. If we keep a list of $|\Sigma| = 26$ pointers to represent the outgoing transitions at every level of the tree, some of the pointers become void. For example, the letter q in many other languages can only be followed by u, so all other pointers on the list reached by q will be void. This amounts to a waste of space: in addition to the 26 pointers going from the root, we would have $26^2 = 676$ for the next level, $26^3 = 17\,576$ for the next one, and so one. With the increasing level (letter position in a word), most of the pointers will become void. The whole trie will take much more space than the actual dictionary! A remedy is to store the non-void pointers only. However, the result will still be likely to be larger than the original dictionary in ASCII format.

3.3.2 Dictionary Automata

Instead of compressing the dictionary, we have achieved quite the opposite effect. However, since it is now encoded as a deterministic finite-state automaton, its size may be reduced using one of the minimization algorithms introduced in section 1.7. Actually, some programs do just that. However, because a trie can be an order of magnitude larger than the corresponding text, it may not fit into the main memory of the computer.

3.3.3 Constructing Minimal Dictionaries

The solution to the above problem is to run two processes in parallel: one that constructs a trie, and another one that minimizes it. Constructing a trie is a straightforward process covered by algorithm 3.2.3. By changing the methods for minimization and synchronization of both processes, we get different construction algorithms that avoid holding the entire trie in the memory.

Minimization was introduced in section 1.7. Note that the structure to be minimized is a tree. It does not contain any loops — such structures are called *acyclic*. Therefore, special minimization algorithms can be used that benefit from that property. An automaton in the form of a tree can be minimized in time proportional to its size. Recall from page 10 that we call such an algorithm linear. Recall also that the best general minimization algorithm — Hopcroft's algorithm (cf. page 42) — has log-linear complexity ($O(n \cdot log\, n)$).

To see how to minimize an acyclic automaton, we must first return to the notion of the right language. Its definition is repeated below for convenience.

$$\overrightarrow{\mathcal{L}}(q) = \{w \in \Sigma^* : \delta^*(q, w) \in F\}$$

Recall that two states are equivalent if they have the same right language. When this happens, it is possible to replace one of them with the other one. The result of performing such replacement on all equivalent pairs is the minimal automaton. It is useful to rewrite the definition of the right language in a different, recursive way:

$$\overrightarrow{\mathcal{L}}(q) = \{a\overrightarrow{\mathcal{L}}(r) : a \in \Sigma, r \in Q, \delta(q, a) = r\} \cup \begin{cases} \emptyset & q \notin F \\ \epsilon & q \in F \end{cases} \tag{3.1}$$

Let the states Q of an automaton be divided into two disjoint sets: a set of states R already evaluated by the minimization algorithm, and the remaining states $U = Q \setminus R$. The set R does not contain any pair of equivalent states. Let us compute $\overrightarrow{\mathcal{L}}(q), q \in U$ in such a way that:

$$\overrightarrow{\mathcal{L}}(q) = \{a\overrightarrow{\mathcal{L}}(r) : a \in \Sigma, r \in Q, \delta(q, a) = r, \mathbf{r} \in \mathbf{R}\}$$

One way to achieve that is to use the postorder method of visiting states of the trie. Suppose that *Oper* is the operation that is to be performed on each state. For each state q, triggered by POSTORDER(q_0), the following code is executed:

Algorithm 3.3.1: POSTORDER($A = (\Sigma, Q, q_0, F, \delta), q$)

for each $a \in \Sigma$ such that $\delta(q, a)$ is defined
  POSTORDER($\delta(q, a)$)
Oper(q)

Once the right language of a state q has been computed, we can look into set R for equivalent states. If such a state p can be found, it replaces q, i.e. state q is deleted, and all transitions that lead to q (in a tree, there is exactly one such transition), are redirected to p. If an equivalent state cannot be found, q is added to R.

Since q_0 — the root of the trie — is warranted to have a unique right language, the minimization of a trie can be written in the following way:

Algorithm 3.3.2: AcyclicMinim$(A = (\Sigma, Q, q_0, F, \delta), q)$

for each $a \in \Sigma : \delta(q, a) \in Q$
  $r \leftarrow \delta(q, a)$
  AcyclicMinim(A, r)
  if $\exists_{p \in R}\ p \equiv r$
    $Q \leftarrow Q \setminus \{r\}$
    $\delta(q, a) \leftarrow p$
  else $R \leftarrow R \cup \{r\}$

This code is executed exactly once for each internal state (non-leaf) of a trie. Each state of the trie except the root becomes the variable r in the **for each** loop exactly once. Deletion of a state can be done in constant time, as can changing the destination of a transition.

The only potentially expensive operation is the comparison $p \equiv q$, i.e. $\overrightarrow{\mathcal{L}}(p) = \overrightarrow{\mathcal{L}}(q)$. Instead of a direct application of the recursive formula 3.1, we can use the fact that the right languages of the states in R are unique. Then the equality $\delta(q, a) = \delta(p, a)$ for each $a \in \Sigma$ (including the case $\delta(q, a)$ undefined, which is treated as a special value $\delta(q, a) = \perp$) is equivalent to $p \equiv q$ provided p and q are both final or both non-final.

Operations on the set R (called the register) are implemented using a hash table. On average, they are executed in constant time. The finality of a state, labels and destinations of transitions are the key in the table — they are used for computing the hash function. So both the register search, and adding a state to the register, run in constant time. The whole minimization algorithm runs in linear — $O(|Q|)$ — time.

Incremental Construction of Minimal Dictionaries

Recall that our aim is not to minimize the trie once it is constructed, but rather to minimize it during its construction. Once a subtree has been minimized, it cannot be changed without additional effort and complication of the minimization algorithm. The key to success is detection of those subtrees that will no longer change.

Let us first assume that the lexicon being minimized is ordered lexicographically. Recall that a string $w = w_1 \dots w_n$ is *lexicographically smaller* than a string $w' = w'_1 \dots w'_m$ (written $w <_{\text{lex}} w'$) if and only if w is a prefix of w' or $w_t < w'_t$ for the first character t after the longest common prefix of w and w' (i.e. $w_1 \dots w_{t-1} = w'_1 \dots w'_{t-1}$).[3]

Now suppose a word w from the sorted dictionary has just been added to a trie. What states can change their right language when the next word w' is added? The answer is "all states along the path that recognizes w". Word w' must either follow the whole path of w, and then extend it with a suffix, or follow a part of it, and then create a new subtree starting with an extra transition added to some state on the path of w. Note that w' cannot follow any existing transition that does not belong to the path of w. Because the words are ordered lexicographically, so are labels on subsequent transitions of a state. If $\forall_{1 \le j \le i}\, w_j = w'_j, w_{i+1} \neq w'_{j+1}$, then $w_{i+1} < w'_{i+1}$. As w_{i+1} is lexicographically the last label on a transition, a new transition has to be created for w'_{i+1}. When we add w', the states $\delta(q_0, w_{1\dots k}), i + 1 \le k \le |w|$ can undergo local minimization.

With all necessary ingredients already prepared, it is easy to formulate the incremental construction algorithm.

[3] As the name suggests, the lexicographic order is the order in which words are typically arranged in dictionaries, *a* preceding *aardvark*, but *aardvark* coming before *and*. The ordering of characters may reflect their encoding — for example, their ASCII codes.

Algorithm 3.3.3: IncrementalConstruction($A = (\Sigma, Q, q_0, F, \delta), input$)

```
w' ← ε
while not Empty(input)
   w ← NextWord(input)
   q ← q0
   i ← 1
   while i ≤ |w| ∧ δ(q, wi) ∈ Q        comment: traverse the common prefix
      q ← δ(q, wi)
      i ← i + 1
   LocalMin(A, q, w'i+1...|w|)
   q ← AddBranch(q, wi ... w|w|)       comment: append the remainder of w.
   F ← F ∪ {q}
   w' ← w
LocalMin(A, q0, w')

procedure LocalMin(A, q, w)
 if |w| > 0
    r ← δ(A, q, w1)
    LocalMin(A, r, w2...|w|)
    if ∃p∈R p ≡ r
       Q ← Q \ {r}
       δ(q, w1) ← p
     else R ← R ∪ {r}
```

The algorithm runs in time proportional to the size of the input data measured in characters. Each input word w can be divided into two parts: a prefix $w_{1\ldots i}$ and a suffix $w_{i+1\ldots|w|}$. Either part can be empty. The first i characters of w are processed in the inner **while** loop — the loop runs i times, its body being executed in constant time. The remaining $|w| - i$ characters are first processed in the call to the procedure AddBranch() (see algorithm 3.2.3 on page 72). The suffix appended by AddBranch() is processed again in calls to LocalMin(). There may be as many as $|w| - i$ calls, but each state recognizing the suffix is processed only once.

Let us see how the algorithm works in practice. Suppose our dictionary comprizes the words *cat*, *chat*, *swat*, and *sweat*. We start with an automaton that contains only the initial non-final state q_0. We add *cat*. The inner loop is skipped as q_0 has no outgoing transitions. A call to LocalMin() does nothing, as there is no previous word, and $w' = \epsilon$. AddBranch() creates the chain of states q_1, q_2, q_3 and makes q_3 final. Then *chat* is added. In the inner **while** loop, the prefix c is traversed, and the current state becomes q_1. Procedure LocalMin() is called recursively as LocalMin(A, q_1, at), LocalMin(A, q_2, t), and LocalMin(A, q_3, ϵ). The last call immediately returns to the second one. Since R is empty, no state equivalent to q_3 can be found, and q_3 is added to R. Back in the first call, no equivalent to q_2 can be found in R, so q_2 is also added to R. AddBranch() creates the chain of states q_4, q_5, and q_6 and makes q_6 final (figure 3.17).

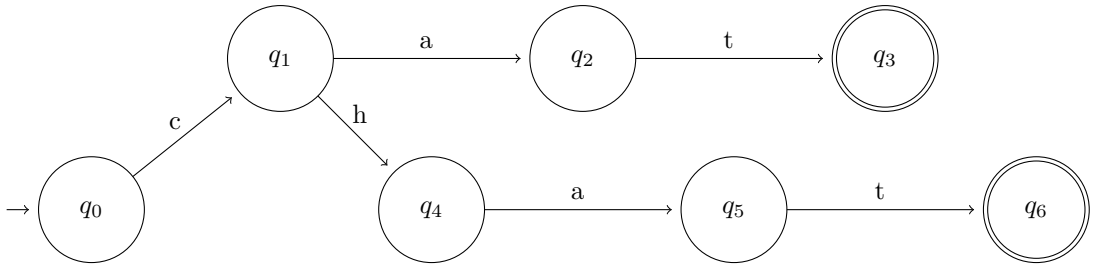


Figure 3.17: Incremental construction: automaton recognizing words *cat* and *chat*.

It is time to add *swat*. The inner loop is not entered as $\delta(q_0, s) \notin Q$. LocalMin() is

called recursively as LocalMin()$(A, q_0, chat)$, LocalMin()(A, q_1, hat), LocalMin()(A, q_4, at), LocalMin()(A, q_5, t), and LocalMin()(A, q_6, ϵ). The last call returns immediately. In the penultimate call, q_6 is found to be equivalent to $q_3 \in R$. So q_6 is deleted, and $\delta(q_5, t)$ is redirected to q_3. Back in the preceding call, q_5 is found equivalent to q_2, gets deleted, and $\delta(q_4, a)$ is redirected to q_2. Back one level, q_4 is found to be unique, so it is added to R. So is q_1 one level up. AddBranch() creates states q_7, q_8, q_9, and a final state q_{10}, as well as transitions $\delta(q_0, s) = q_7$, $\delta(q_7, w) = q_8$, $\delta(q_8, a) = q_9$, and $\delta(q_9, t) = q_{10}$. The result is shown in figure 3.18.

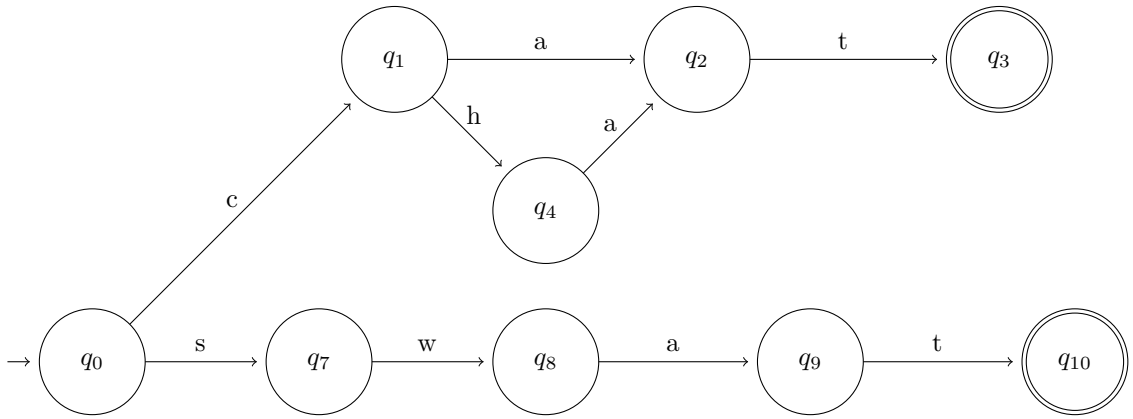


Figure 3.18: Incremental construction: automaton recognizing words *cat*, *chat*, and *swat*.

Now, we add *sweat*. The inner loop brings us to q_8. LocalMin(A, q_8, at), LocalMin(A, q_9, t), and LocalMin(A, q_{10}, ϵ) are called. The last call returns immediately, and in the preceding one, q_10 is found equivalent to q_3, so it gets deleted, and $\delta(q_9, t)$ is redirected to q_3. One level up, as q_9 is found equivalent to q_2, it gets deleted, and $\delta(q_8, a)$ is redirected to q_2. In AddBranch(), q_{11}, q_{12}, as well as $\delta(q_8, e) = q_{11}$, $\delta(q_{11}, a) = q_{12}$, and $\delta(q_{12}, t) = q_{13}$ are created. The result is shown in figure 3.19.

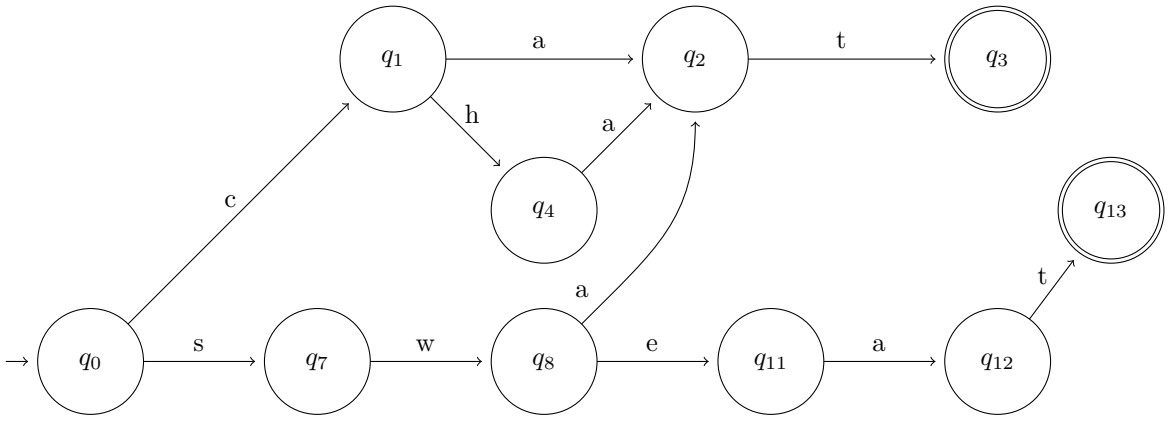


Figure 3.19: Incremental construction: automaton recognizing words *cat*, *chat*, *swat*, and *sweat*.

Finally, there are no more words to be added. Recursive calls LocalMin$(A, q_0, sweat)$, LocalMin$(A, q_7, weat)$, LocalMin(A, q_8, eat), ..., LocalMin(A, q_{13}, ϵ) result in the minimal automaton. The last call returns immediately, the penultimate one deletes q_{13} and replaces it with q_3, redirecting $\delta(q_{12}, t)$ to q_3. The preceding call deletes q_{11}, replaces it with q_2, and redirects $\delta(q_{11}, a)$ to q_2. Then q_{11} is deleted, and $\delta(q_8, e)$ is redirected to q_4. Finally, q_8 and q_7 are found to have unique right languages, so they are added to R. The result is shown in figure 3.20.

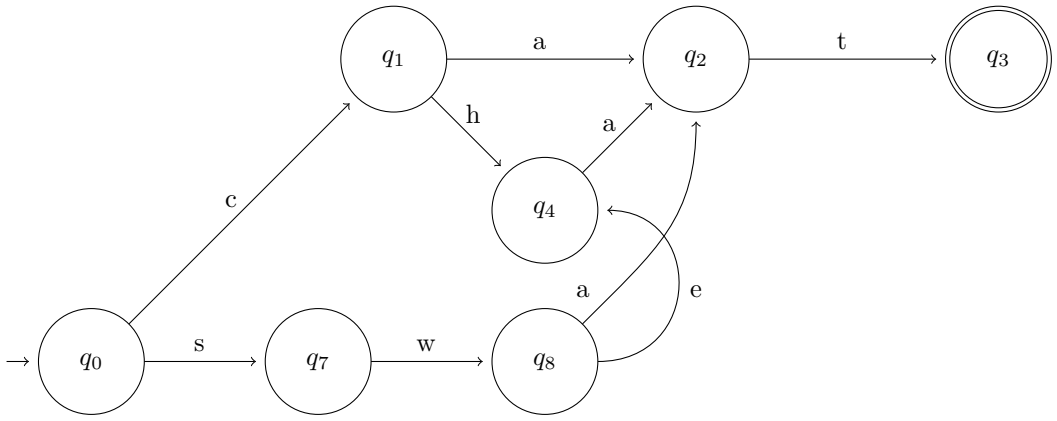


Figure 3.20: Incremental construction: the minimal automaton recognizing *cat*, *chat*, *swat*, and *sweat*.

Incremental Construction from Unordered Data

What happens when words come in arbitrary order? Since it is impossible to predict what the next word could be, there is no part of the automaton that is safe from future modification. The automaton has to be minimized entirely each time a new word has been added. Therefore, each time a new word is added, it is added to a minimal automaton.

When a word is being added to the language of an automaton, the first part of the word — the prefix — is already there. The remaining part must be used to create a path recognizing the suffix. However, there is a complication. A minimal automaton may contain states that have more than one incoming transition. They are called *confluence states*.

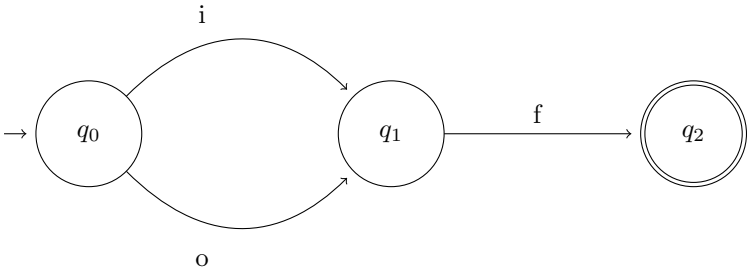


Figure 3.21: Automaton recognizing the words *if* and *of*.

Figure 3.21 shows the minimal DFSA recognizing *if* and *of*. Suppose we want to add *it*. The prefix *i* is already there. It leads to state q_1, so we add a transition labeled *t* to q_1 (figure 3.22).

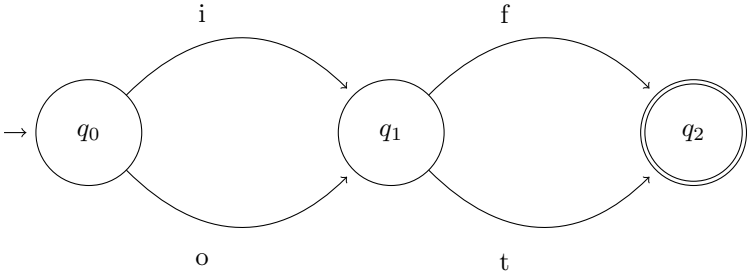


Figure 3.22: Automaton from figure 3.21 with naïvely added *it*. The FSA also recognizes *ot*.

Note that the above operation "overgenerates": it inserts not only *in*, but also *ot*. The latter is added because q_1 is reached not only via $\delta(q_0, i)$, but also via $\delta(q_0, o)$ — it is a confluence state.

Confluence states are the consequence of minimization. They represent several isomorphic subtrees merged into one. As only one of those subtrees is to be modified, it should first be extracted. Since only the path traversed during recognition of a prefix of the word to be added is to be modified, only that path should be extracted. It is done by cloning states in that path, starting from the first confluence state. Cloning means making an exact copy of a state, including its finality and the suite of outgoing transitions. Once the extraction is done, a path recognizing the suffix can be created. This procedure creates a new automaton that recognizes the language of the old automaton and the new word just added. However, the new automaton is not minimal.

Recall that the word is added to the minimal automaton. All states in the automaton are already in the register R. All states in the path of the prefix of the word just added change their right language. Theoretically, they all should be removed from R and reevaluated. Practically, this is not always necessary. Recall that if the proper order of evaluation is used, checking equivalence of two states involves only checking their finality and the set of outgoing transitions (including destination states). They form the key of the hash table implementing the register R. If those features do not change, there is no need to change the register.

The only state that needs to be removed from R is either the state preceding the first confluence state, or — if no confluence states are in the path — the last state in the path of the prefix. All states that follow it are not in R. In the second case, they are newly created states. In the first case, newly created states are preceded by clones of existing states. The state that needs to be removed from R changes its suite of outgoing transitions — either a new transition is added to it, or an existing transition is redirected to a clone of its original destination, or the state becomes final — when the word just added is a prefix of another word already in the language of the automaton. When the state is reevaluated, it can be replaced with another state. In that case, the suite of outgoing transitions of the preceding state changes, and that state must be removed from R. Changes can percolate to the initial state.

Algorithm 3.3.4: UPDATEAUTOMATON$(A = (\Sigma, Q, q_0, F, \delta), w)$

$i \leftarrow 0$
$q \leftarrow q_0$
while $i \leq |w| \wedge \delta(q, w_i) \in Q \wedge |\{(p, a) : \delta(p, a) = \delta(q, w_i)\}| = 1$ (1)
  $q \leftarrow \delta(q, w_i)$
  $i \leftarrow i + 1$
$u \leftarrow i$
$R \leftarrow R \setminus \{q\}$
while $i \leq |w| \wedge \delta(q, w_i) \in Q$ (2)
  $p \leftarrow$ CLONE$(\delta(q, w_i))$
  $Q \leftarrow Q \cup \{p\}$
  $\delta(q, w_i) \leftarrow p$
  $q \leftarrow p$
  $i \leftarrow i + 1$
$q \leftarrow$ ADDBRANCH$(q, w_i \ldots w_{|w|})$
$F \leftarrow F \cup \{q\}$
while $i \geq u$ (3)
  $r \leftarrow \delta(q_0, w_{1 \ldots i-1})$
  if $\exists_{p \in R}\ p \equiv q$
    $Q \leftarrow Q \setminus \{q\}$
    $\delta(r, w_i) \leftarrow p$
    if $i = u \wedge i > 0$
      $u \leftarrow u - 1$
      $R \leftarrow R \setminus \{r\}$
   else $R \leftarrow R \cup \{q\}$
  $i \leftarrow i - 1$
  $q \leftarrow r$

The **while** loop (1) traverses the part of the prefix free of confluence states. The next two lines remove a state from the register and record its location in variable u. The **while** loop (2) executes only if confluence states are found in the path. It clones them. Note that when a state is cloned, all targets of its transitions have incoming transitions from both the original and the clone. They become confluence states. Once a confluence state in the prefix path is found and cloned, all subsequent states in the path must also be cloned. The call to AddBranch() (cf. algorithm 3.2.3) creates a chain of states and transitions that recognize the suffix of the word being added. The last **while** loop (3) goes back from the end of the path recognizing the whole word just added. States are evaluated and either replaced or put into R. If the state is replaced, and it is the first state in the path not included in R, the previous state is removed from R and its position is recorded in variable u. The process stops when there are no new replacements.

The algorithm runs in time proportional to the length of the word. The bodies of all **while** loops can be executed in constant time. It is convenient to store states of the current path in a vector, so that $r \leftarrow \delta(q_0, w_{1\ldots i-1})$ becomes just vector indexing performed in constant time. The first two **while** loops execute $|w|$ times altogether, while AddBranch() and the last loop are executed in $|w|$ steps. Thus, the algorithm has the same asymptotic complexity as the version for sorted data. In practice, however, it runs more slowly, as certain states have to be reevaluated over and over again.

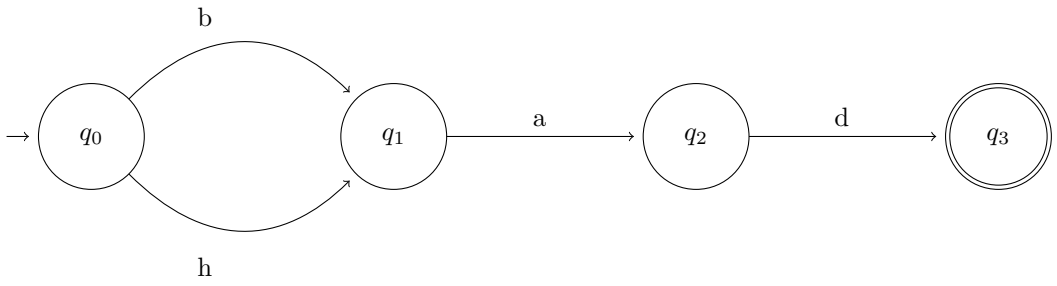


Figure 3.23: Incremental addition: the minimal automaton recognizing *bad* and *had*.

Let us take a look at an example. The automaton in figure 3.23 recognizes words *bad* and *had*. We add *bat*. The first **while** loop in algorithm 3.3.4 is skipped because q_1 is a confluence state. u is set to 0, and q_0 is removed from R. The first iteration of the second loop clones state q_1. The clone q_4 has the same transitions as q_1. Note that state q_2 automatically becomes a confluence state. The situation is shown in figure 3.24.

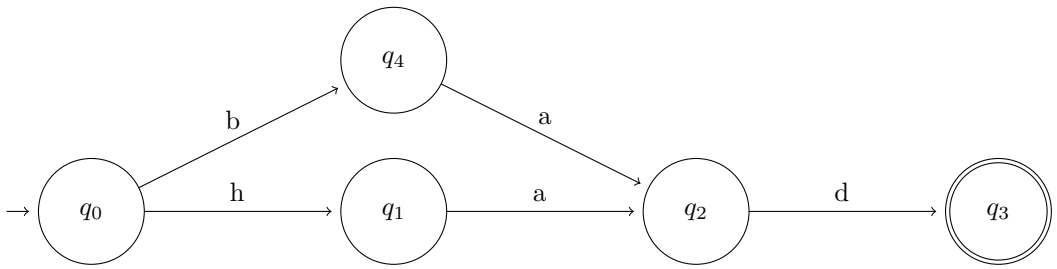


Figure 3.24: Incremental addition: in the minimal automaton from Figure 3.23, state q_1 was cloned.

The second iteration of the **while** loop (2) clones state q_2. The clone (q_4) has a transition labeled d leading to state q_3. The call to AddBranch() creates a transition to a new state q_6. The state is made final. In the first iteration of the **while** loop (3), q_6 is found equivalent to q_3.

Then q_6 is deleted, and $\delta(q_5, t)$ is redirected to q_3. Subsequent iterations of the loop put q_5, q_4 and q_0 into the register.

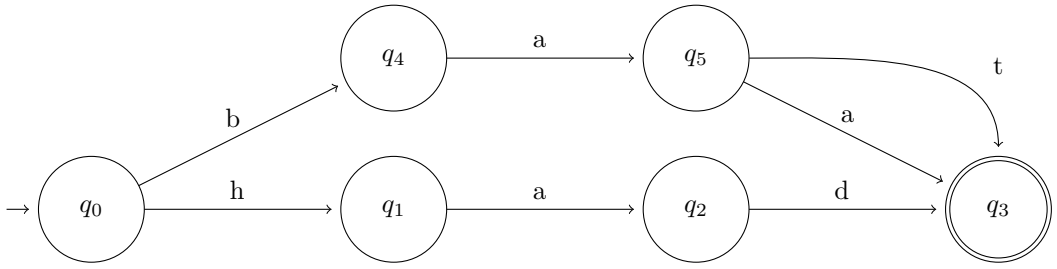


Figure 3.25: Incremental addition: the minimal automaton recognizing *bad*, *bat*, and *had*.

Now we add the word *hat*. The **while** loop (1) takes us to state q_2. It is removed from R. u is set to 3. The loop (2) has nothing to clone, while the next one creates a new state q_7, and a transition to it $\delta(q_2, t) = q_7$. i is incremented to 4. The state is made final. In the first iteration of the loop (3), it is found equivalent to q_3, so it is deleted, and $\delta(q_2, t)$ is redirected to q_3. i is set to 3. In the next iteration, q_2 is found to be equivalent to q_5. State q_2 is deleted, and $\delta(q_1, a)$ is redirected to q_5. Since $u = i$, and there was a replacement, u is decremented so that the next iteration is possible. At the same time, q_1 is removed from R. In that iteration, q_1 is found to be equivalent to q_4, so q_1 is deleted, and $\delta(q_0, h)$ is redirected to q_4. Again, u is decremented, and q_0 is removed from the register. In the last iteration, q_0 is put back to R. The minimal automaton is shown in figure 3.26.

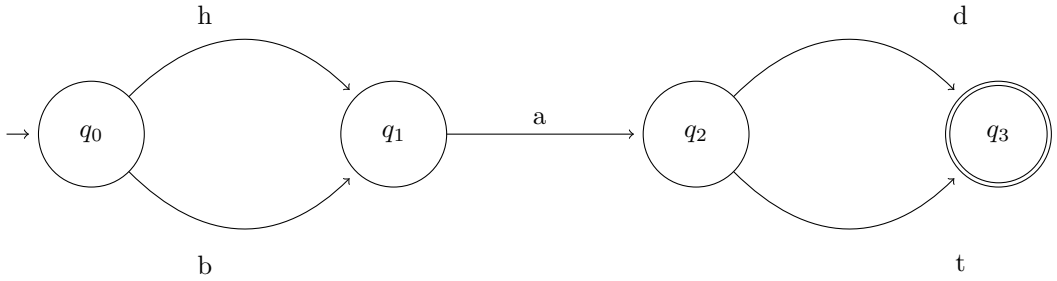


Figure 3.26: Incremental addition: the minimal automaton recognizing *bad*, *bat*, *had*, and *hat*.

3.3.4 FSA as an Associative Container

The dictionaries presented so far in this book were simple word lists. They are of limited interest to a linguist; it could be useful to associate some information with the words. One technique is to encode it at the end of the words. Suppose part-of-speech tags are needed for each word. We no longer put a pure word, like *beautiful* or *beautifully* into an automaton. Instead, we append a separator (We use a plus sign here) and the tag to the end of the word, and put that string into the FSA. As a result, *beautiful* becomes `beautiful+Adj`, and *beautifully* becomes `beautifully+Adv`. This encoding can be viewed as a generalization of the *final transitions* introduced in section 3.1.2.

The language of the automaton is no longer a list of words. To find a word w, one forms w' as a concatenation of w and the separator, and one checks whether $\delta(q_0, w')$ is defined. If it is indeed so, then the word is in the dictionary. In addition, $\overrightarrow{\mathcal{L}}\delta(q_0, w')$ is the set of tags associated with the word.

This method of associating extra information with lexical entries is effective as long as the information depends on their endings, as part-of-speech tags do. For example, English words ending in *-ing* are mostly gerunds (`+Ger`). A minimization algorithm will be able to factor out the common suffix `ing+Ger` into a single substructure in the lexicon, thus avoiding the multiple representation of the suffix for the different *-ing* verb forms.

More encoding methods for dictionaries are discussed in the literature. One common option is to use finite-state transducers, i.e. finite-state machines that can produce output strings (Karttunen 1994). The incremental construction of minimal transducers is discussed by Mihov and Maurel (2001) and Skut (2004).

Another technique of associating information with words is the *minimal perfect hashing*, which maps each word in a dictionary to a unique integer value called a *hash code* (Revuz 1991, Lucchiesi and Kowaltowski 1993).

3.4 Further Reading

There exists a rich literature on pattern matching with finite-state automata. Crochemore and Rytter (1994) provide a very good introduction this topic. A more in-depth treatment of the subject can be found in the books by Charras and Lecroq (2004) and Crochemore and Hancart (1997).

The use of tries and minimal automata as dictionaries was introduced by Liang (1983). Associating additional information with words in an automaton is discussed by Revuz (1991), as well as by Lucchiesi and Kowaltowski (1993) and Kowaltowski, Lucchesi and Stolfi (1998). Another method of providing that functionality is to use finite-state transducers, i.e. finite-state machines that can produce output strings (Karttunen 1994). Another technique of associating *any* information with words is *minimal perfect hashing* (Revuz 1991, Lucchiesi and Kowaltowski 1993). In general, the proceedings of a series of workshops Finite State Methods in Natural Language Processing held since 1998 can provide the readers with valuable material on many aspects of the use of finite-state devices in natural language processing.

Further NLP applications of finite-state machines range from text indexing (Silberztein 1999b) and spelling correction (Oflazer 1996) to morphology modeling (Koskenniemi 1983, Karttunen and Beesley 2003) and parsing (Abney 1991, Karlsson, Voutilainen, Heikkila and Anttila 1995, Grefenstette 1996, Roche 1997).

Chapter 4

Implementing Automata

This chapter discusses issues related to the implementation of finite-state automata, and tools for manipulating them. Section 4.1 introduces standard data structures for representing automata and outlines various compression and speed-up techniques. Subsequently, in section 4.2, we briefly introduce a number of software toolkits, which are freely available for research purposes, for building, combining and optimizing finite-state devices.

4.1 Data Structures for Representing Automata

One of the major issues in the implemention of finite-state devices is their computational representation. Since an FSA can be seen as a labeled directed graph, we can deploy standard data structures used to represent graphs, i.e. *adjacency matrices* and *adjacency lists*, with some slight enhancements. Selecting an appropriate storage model for an FSA in a given scenario requires consideration of the following questions:

- Is the FSA deterministic or non-deterministic?
- Is it static (it will be employed in read-only mode) or dynamic?
- What are the major operations which will be carried out on the automaton?

For example, consider a named entity extractor that locates occurrences of named entity patterns in texts. The patterns are formulated using regular expressions and compiled into a device according to the algorithm described in section 3.2. The extractor is constructed and applied in three steps. First, each of the the regular expressions is compiled into an NFSA. In the second step, the NFSAs are combined into a single automaton, which is determinized and minimized. Finally, the resulting DFSA is used to locate named entities in a text.

The types of operations performed at the three stages differ. The NFSAs constructed in steps one and two must support efficient dynamic insertion and deletion of transitions (for algorithms such as ϵ-removal, determinization and minimization). Fast *iteration* over sets of states and/or transitions is also desirable. On the other hand, the resulting DFSA is static: once constructed, it is never changed. The only operation required now is its fast application to strings, which entails efficient computation of the transition $\delta(q, s)$ for a given state q and symbol s.

As it happens, there is no data structure that would be optimal for both kinds of operations. In effect, different data structures are often used at different processing stages. In the remainder of this chapter, we investigate a number of data structures from the efficiency perspective. In order to simplify description, we focus on DFSAs, but most of the data structures can be extended to NFSAs in a trivial way.

4.1.1 Transition Matrix

The simplest way to represent a DFSA $A = (\Sigma, Q, q_0, F, \delta)$ is to use a $|Q| \times |\Sigma|$ matrix whose ij-th element contains the value of $\delta(q_i, a_j)$, where a_j is the j-th symbol in the alphabet Σ (we can easily define a mapping from alphabet symbols to integers). If the transition function is not defined for a given state-symbol combination then the corresponding matrix element contains a **null** value. The presented data structure is known as the *adjacency matrix*. For the sake of clarity in the context of automata we call such a matrix a *transition matrix*. An extension to NFSAs is straightforward, i.e., the ij-th element of the transition matrix contains a list of all target states of outgoing transitions from state q_i labeled with a_j. Figure 4.1 gives an example of a simple DFSA with the corresponding transition matrix depicted in table 4.1. For marking states as initial, accepting or rejecting, we can deploy a simple boolean-valued vector.

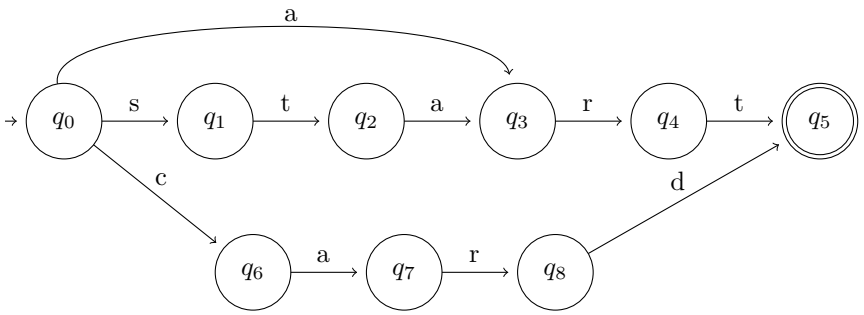


Figure 4.1: An DFSA accepting the language $\{start, card, art\}$.

Q/Σ	a	c	d	r	s	t
q_0	3	6			1	
q_1						2
q_2	3					
q_3				4		
q_4						5
q_5						
q_6	7					
q_7				8		
q_8			5			

Table 4.1: A transition matrix for the DFSA in figure 4.1.

What makes a transition matrix representation highly attractive is the quick access to the information on $\delta(q, a)$, which costs $O(1)$ time, and the facility to efficiently add and delete transitions, which can be similarilly done in constant time.

Unfortunately, the matrix representation has two major drawbacks. Firstly, the complexity of iterating over all transitions from a given state is proportional to the size of the alphabet since we must inspect every single element in the row corresponding to the current source state. Note that this is not a *worst-case* estimate (as the $O(2^n)$ complexity of determinization) — iteration *always* takes $|\Sigma|$ steps for each state, which is formally abbreviated as $\Theta(|\Sigma|)$ (see framed box on page 38).

The other major drawback of the matrix representation is its high space requirement, amounting to $\Theta(|Q| \cdot |\Sigma|)$. As we can see in the example in table 4.1 only a minor part of the transition table is filled with non-**null** values, i.e. the average number of outgoing transitions from a given state is relatively low. We call such automata *sparse*, whereas automata with a high average

number of outgoing transitions from a given state are called *dense*. Formally, the density of an DFSA $A = (\Sigma, Q, q_0, F, \delta)$ is defined as follows:

$$density(A) = \frac{|\delta|}{|Q| \times |\Sigma|} \tag{4.1}$$

The density of the automaton in figure 4.1 is $10/48 \approx 20.83\%$.

Obviously, the density of automata depends on the process or phenomenon we are modeling. In most NLP applications, however, we deal with very sparse automata. For example, the density of DFSAs implementing large morphological lexica oscillates around $1 - 2\%$.

Therefore, the matrix representation is used relatively rarely, most typically for dense and small automata, especially when there is a need to frequently modify transitions in constant time.

4.1.2 Transition Lists

An alternative way of representing an automaton is to use *adjacency lists*. For each state q in the automaton we define a list of all pairs (a, p) such that $\delta(q, a) = p$. In the world of finite-state devices we will call adjacency lists *transition lists*. This data structure is suitable for both static and dynamic automata, and for both deterministic and nondeterministic variants as well. Figure 4.2 shows an example of transition-list representation of the automaton in figure 4.1.

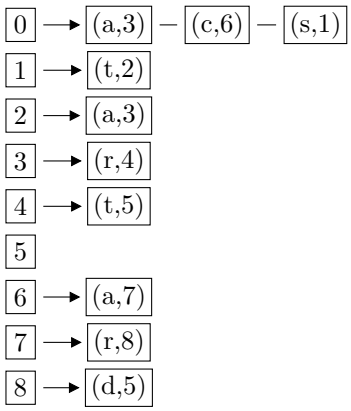


Figure 4.2: Transition-list representation of the automaton in figure 4.1.

The main advantage of using this data structure is its low space complexity. Its memory requirements are proportional to the size of the automaton and amount to $\Theta(|Q| + |\delta|)$. Having this nice feature is penalized by slower access to $\delta(q, a)$. In the worst case we have to traverse the whole list for a given state in order to locate a transition labeled with a given symbol. As a result, the overall complexity of this operation is $O(|\Sigma|)$ in the case of DFSAs and $O(|Q| \cdot |\Sigma|)$ for NFSAs. As discussed earlier, a majority of finite-state based NLP applications usually deploy sparse automata, which means that the runtime performance of the $\delta(q, a)$-look-up operation does not constitute a critical factor. Nevertheless, in the case of denser automata we could keep the pairs in the transition lists sorted by input symbol, which could reduce the time for accessing single transitions to $O(log|\Sigma|)$ via application of *binary search* (see frame on page 80, and note that in such a case lists have to be implemented as vectors in order to meet the random-access requirement).

Consequently, adding a new transition to the automaton would involve inserting it into the appropriate transition list at the right position. This could be done in $O(|\Sigma|)$ steps since in the worst case the whole list (represented as a vector) has to be reallocated. Fortunately, if addition

and deletion operations are highly relevant, we could find a better solution. For instance, instead of lists we could utilize balanced trees or even more fancy and efficient data structures for storing transitions, which allow for accessing, deleting and inserting elements in sorted sets of size n in $O(logn)$ time. It means that operations involving modification of transitions could be performed in $O(log|\Sigma|)$ time. Further details on efficient data structures implementing dynamic sets can be found in Cormen et al. (2001).

Finally, transition-list representation allows for iterating over all transitions from a given state q in $\Theta(|outdegree(q)|)$, where $outdegree(q)$ denotes the number of outgoing transitions from state q. In comparison to matrix represention there is still one more tiny advantage, namely: introducing new transitions labeled with symbols not covered by the current alphabet is somewhat easier than in the case of a transition matrix since an introduction of a new symbol to the alphabet in the latter case would result in expensive reorganization of the underlying data structure.

4.1.3 Compressed Transition Matrix

The representations introduced in the two previous sections have the advantage of either compact memory representation (transition lists) or fast transition access (transition matrix). However, they always sacrifice the other desirable property, which results in slower transition access for transition lists and high memory requirements for transition tables. Fortunately, there are data structures that combine both advantages.

Sparse transition matrices can be compressed in such a manner that the space requirement is nearly linear in the number of transitions, without sacrifying the constant time for accessing $\delta(q, a)$. The idea is to shift and overlap the rows of the transition matrix so that no two non-zero entries end up in the same position, and to store them in a one-dimensional array. This can be done in a greedy manner by successively placing the consecutive rows of the transition matrix into the array DELTA in such a way that collisions are avoided, i.e. a single element in the array DELTA may refer to at most one element in some row of the transition matrix. Additionally, we introduce an array ROW[1...|Q|] for storing for each state a pointer to the beginning of the corresponding transition row stored in DELTA. Figure 4.3 gives an example of shifting and overlapping the rows of the transition matrix presented in table 4.1. As we can see the rows 0,1,2,3 and 5 can share the same space in DELTA without performing any shifting operations, whereas other rows have to be displaced in order to avoid clashes.

For accessing the value of $\delta(q, a)$, we simply need to check the element in DELTA at position ROW[q]+INDEX[a], where INDEX maps each alphabet symbol to a unique integer. One question remains: namely, how can we guarantee that a non-empty value in DELTA at a given index encodes the target state of some outgoing transition from state q. In order to assure this, we still need another one-dimensional array OWNER of the same length as DELTA and assigns each element in DELTA an associated state, i.e. OWNER[i]=q means that the information stored in DELTA[i] refers to a transition for state q. If $\delta(q, a)$ is undefined and the corresponding element in DELTA is not utilized, we can assign the latter a separate value which stands for "undefined and unused" (denoted by a dash in our example). The pseudocode of the operation for accessing δ is given below.

Algorithm 4.1.1: GETDELTA(q, a)

```
if OWNER[ROW[q] + INDEX[a]] = q
   return (DELTA[ROW[q] + INDEX[a]])
 else
   return (nil)
```

Let us now briefly discuss the issue of packing a transition table into a one-dimensional array. The example given above represents just a single row displacement. Ideally, we would be interested in finding a set of row displacements that minimizes the size of DELTA. Although this task is NP-complete, there are many heuristics which yield nearly optimal compression rates, in particular

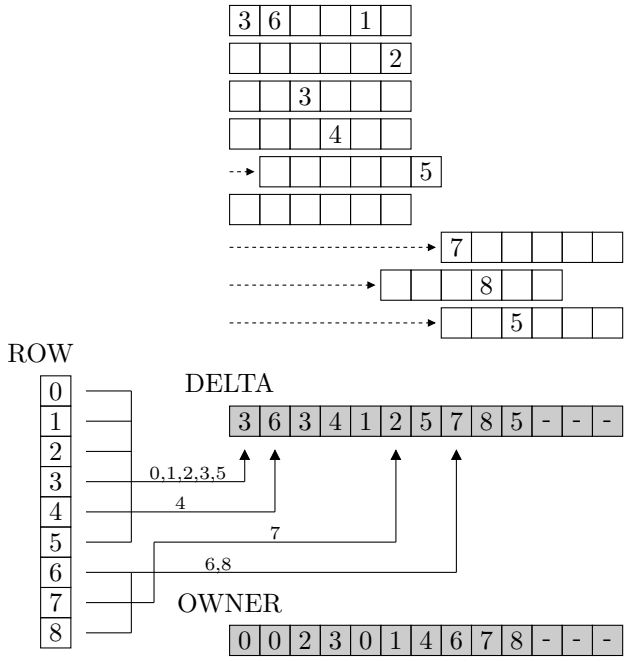


Figure 4.3: Compression of transition table presented in figure 4.1.

in the case of typical NLP applications. The simplest way is the so called 'first-fit' strategy. In the i-th step we try to shift the i-th row from left to right over the previously packed first $i-1$ rows already stored in DELTA, until a zero-collision overlap has been identified. Since the initial elements in DELTA will already be covered by the first couple of transition rows, a more efficient way is to start computing a collision-free overlap from the first non-occupied position in DELTA. Further improvements can be achieved by sorting all transition rows with respect to the number of transitions they encode, which imposes the order of packing the rows, i.e. rows containing more elements are packed before we proceed with packing sparse rows. Alternatively, we could partition the set of transition rows into dense and sparse based on some threshold, and subsequently pack dense rows in the aforementioned manner, then precompute free positions in DELTA and finally displace the sparse rows via utilization of precomputed free positions for speeding up the whole process.

The application of the above techniques may result in a fair compression rate. But for all that, compressed matrices are not suitable for representing dynamic automata since adding new transitions would lead to time-intensive recomputations. Therefore their usability is limited to static finite-state devices.

Table 4.2 summarizes the main features of the presented storage models for deterministic automata in terms of space and time complexity of relevant operations.

Data Structure	space	accessing $\delta(q,a)$	iteration	modification
transition matrix	$\Theta(\|Q\| \cdot \|\Sigma\|)$	$O(1)$	$O(\|\Sigma\|)$	$O(1)$
transition lists	$\Theta(\|Q\| + \|\delta\|)$	$O(outdeg(q))$	$\Theta(outdeg(q))$	$O(outdeg(q))$
sorted transition lists	$O(\|Q\| + \|\delta\|)$	$O(log(outdeg(q)))$	$\Theta(outdeg(q))$	$O(outdeg(q))$
compressed trans. matrix	$\Omega(\|Q\| + \|\delta\|)$ $O(\|Q\| \cdot \|\Sigma\|)$	$O(1)$	$\Theta(\|\Sigma\|)$	-

Table 4.2: Comparison of different data strctures for representing FSAs.

Please note that for NFSAs all the values in the table are identical except the space complexity of the transition matrix. In this case the space complexity is $\Theta(|Q| \cdot |\Sigma| + |\delta|)$ since the elements of the transition table are lists of total length $|\delta|$. Analogously, the space complexity of a compressed transition table is $O(|Q| \cdot |\Sigma| + |\delta|)$ ($\Omega(|Q| + |\delta|)$). Finally, bear in mind again that replacing lists with balanced trees (or other efficient data structures for implementing dynamic ordered sets) for keeping the transition sorted would yield an $O(log(outdeg(q)))$ complexity of the modification operation in the third row in the table 4.2.

4.2 Finite-State Toolkits

This book provides the reader with sufficient knowledge to build his or her own finite-state toolkit. However, the implementation of such a library is a time-consuming task, and a number of broadly-used and well-debugged toolkits are freely available for research purposes.

Two such toolkits are already mentioned in the introduction. Xerox finite-state tools (*xfst*, *twolc*, *lexc*) were developed by some of the best-known researchers in the domain. The tools are very sophisticated, efficient, and with a sound theoretical background. They offer a complete system. More information can be found in a book by Karttunen and Beesley (2003), and on the website http://www.xrce.xerox.com/competencies/content-analysis/fst/home.en.html.

The second toolkit mentioned in the introduction is INTEX, which is described in Silberztein (1999a). It is the result of decades of research by French scientists. While Xerox puts more stress on various finite-state operators, INTEX has a more sophisticated user interface, and also features push-down automata. More information on INTEX can be found at http://msh.univ-fcomte.fr/intex/. Unfortunately, recent versions of INTEX run only on Windows. UNITEX is a very similar system, based on the same concepts; it works in Unicode, and is written in Java, so it can be used on a variety of platforms. More information about UNITEX can be found at http://www-igm.univ-mlv.fr/~unitex/.

The AT&T FSM library is a collection of about 30 programs that manipulate automata and transducers with special focus on weighted finite-state machines. They also offer some of the best visualization tools for graphs (the graphical representation for finite-state machines). More information about the library can be found at http://www.research.att.com/sw/tools/fsm/ or in the article Mohri, Pereira and Riley (1998).

Gertjan van Noord's `fsa` program is written in Prolog. This makes it slow for large-scale NLP systems, but at the same time highly flexible. It is a program of choice for theoretical research. More information can be found at http://odur.let.rug.nl/~vannoord/Fsa/fsa.html. It is possible to translate operators from the AT&T toolkit and from Xerox tools to the form used by `fsa` – details can be found at http://cs.haifa.ac.il/~shuly/teaching/03/lab/fst.html or http://www.sfs.uni-tuebingen.de/~nvail/ss04/synopsis-of-software-tools.html.

Many other toolkits are available. There is a large variety of software, some of it written by the authors of this book. Sheng Yu maintains a list of pointers to finite-state software packages at http://www.csd.uwo.ca/research/grail/links.html. A more linguistically oriented database can be found at http://tangra.si.umich.edu/clair/universe-rk/html/u/db/acl/.

4.3 Further Reading

Modeling compact and efficient data structures for representing automata is a well-studied area. To take a deeper insight into general compression techniques of automata we recommend the articles Kiraz (1999), Daciuk (2000) and Beijer, Watson and Kourie (2003) for further reading. Tarjan and Yao (1979) gives an extensive overview of state-of-the-art methods for storing sparse tables which can be utilized in the context of implementing automata. For those considering implementing their own finite-state tools we recommend studying the articles Mohri et al. (1998) and Kanthak and Ney (2004), which handle crucial implementational issues in more detail.

Bibliography

Abney, S.(1991), Parsing by chunks, *in* R. Berwick, S. Abney and C. Tenny (eds), *Principle-Based Parsing*, Kluwer Academic Press, Dordrecht.

Aho, A. V., Hopcroft, J. E. and Ullman, J. D.(1974), *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company.

Aho, A. V., Sethi, R. and Ullman, J. D.(1988), *Compilers. Principles, Techniques, and Tools*, Addison-Wesley.

Beijer, N. D., Watson, B. W. and Kourie, D. G.(2003), Stretching and jamming of automata, *SAICSIT 2003: Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, South African Institute for Computer Scientists and Information Technologists, , Republic of South Africa, pp. 198–207.

Charras, C. and Lecroq, T.(2004), *Handbook of Exact String Matching Algorithms*, King's College London Publications.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C.(2001), *Introduction to Algorithms, Second Edition*, MIT Press.

Crochemore, M. and Hancart, C.(1997), Automata for matching patterns, *in* G. Rozenberg and A. Salomaa (eds), *Handbook of Formal Languages*, Vol. 2, Springer-Verlag, pp. 399–462.

Crochemore, M. and Rytter, W.(1994), *Text Algorithms*, Oxford University Press, New York.

Daciuk, J.(2000), Experiments with automata compression., *Proceedings of CIAA - Implementation and Application of Automata*, London, Ontario, Canada, pp. 105–112.

Glushkov, V. M.(1961), The abstract theory of automata, *Russian Mathematical Surveys* **16**, 1–53.

Grefenstette, G.(1996), Light parsing as finite-state filtering, *EACI 1996 Workshop Extended Finite-State Models of Language*, Budapest.

Hopcroft, J. E., Motwani, R. and Ullman, J. D.(2001), *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley.

Kanthak, S. and Ney, H.(2004), Fsa: An efficient and flexible c++ toolkit for finite state automata using on-demand computation, *In Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL 2004)*, Barcelona, Spain, pp. 510–517.

Karlsson, F., Voutilainen, A., Heikkila, J. and Anttila, A.(1995), *Constraint Grammar, A Language-independent System for Parsing Unrestricted Text*, Mouton de Gruyter.

Karttunen, L.(1994), Constructing lexical transducers, *COLING-94*, Kyoto, Japan.

Karttunen, L. and Beesley, K.(2003), *Finite-State Morphology*, Chicago University Press.

Kiraz, G. A.(1999), Compressed storage of sparse finite-state transducers, *Proceedings of WIA 1999*, Potsdam, Germany, pp. 109–121.

Koskenniemi, K.(1983), Two-level model for morphological analysis, *IJCAI-83*, Karlsruhe, Germany, pp. 683–685.

Kowaltowski, T., Lucchesi, C. L. and Stolfi, J.(1998), Finite automata and efficient lexicon implementation, *Technical Report IC-98-02*.

Leslie, T.(1995), *Efficient approaches to subset construction*, Master's thesis, Computer Science, University of Waterloo.

Liang, F. M.(1983), *Word Hy-phen-a-tion by Comp-uter*, PhD thesis, Stanford University.

Lucchiesi, C. and Kowaltowski, T.(1993), Applications of finite automata representing large vocabularies, *Software Practice and Experience* **23**(1), 15–30.

McNaughton, R. and Yamada, H.(1960), Regular expressions and state graphs for automata, *IEEE Transactions on Electronic Computers* **9**, 39–47.

Mihov, S. and Maurel, D.(2001), Direct construction of minimal acyclic subsequential transducers, *in* S. Yu (ed.), *Implementation and Application of Automata*, Vol. 2088 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 217–229.

Mohri, M.(1997), Finite-state transducers in language and speech processing, *Computational Linguistics* **23**(2), 269–311.

Mohri, M., Pereira, F. C. N. and Riley, M.(1998), A rational design for a weighted finite-state transducer library, *WIA '97: Revised Papers from the Second International Workshop on Implementing Automata*, Springer-Verlag, London, UK, pp. 144–158.

Oflazer, K.(1996), Error-tolerant finite state recognition with applications to morphological analysis and spelling correction, *Computational Linguistics* **22**(1), 73–89.

Revuz, D.(1991), *Dictionnaires et lexiques, méthodes et algorithmes*, PhD thesis, Université Paris 7.

Roche, E.(1997), Parsing with finite-state transducers, *in* E. Roche and Y. Schabes (eds), *Finite-State Language Processing*, MIT Press, Cambridge, pp. 241–281.

Roche, E. and Schabes, Y. (eds)(1997), *Finite-State Language Processing*, Bradford Book, MIT Press, Cambridge, Massachusetts, USA.

Silberztein, M.(1999a), INTEX tutorial notes, *in* D. Wood and D. Maurel (eds), *Workshop on Implementing Automata WIA99 – Pre-Proceedings*, Springer-Verlag, Potsdam, Germany, pp. XIX–1 – XIX–31.

Silberztein, M.(1999b), Text indexation with intex, *Computers and the Humanities* **33**(3), 265–280.

Skut, W.(2004), Incremental construction of minimal acyclic sequential transducers from unsorted data, *Proceedings of COLING 2004*, Geneva, Switzerland.

Tarjan, R. E. and Yao, A. C.-C.(1979), Storing a sparse table, *Commun. ACM* **22**(11), 606–611.

Index