Finite-State Transducers and Speech Recognition

School of Computer Science
McGill University,Montreal

A Thesis submitted to the faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science

Patrick Cardinal

March 2003

Your file Votre référence
ISBN: 0-612-88170-9
Our file Notre référence
ISBN: 0-612-88170-9

Abstract

Finite-state automata and finite-state transducers have been extensively studied over the years. Recently, the theory of transducers has been generalized by Mohri for the weighted case. This generalization has allowed the use of finite-state transducers in a large variety of applications such as speech recognition. In this work, most of the algorithms for performing operations on weighted finite-state transducers are described in detail and analyzed. Then, an example of their use is given via a description of a speech recognition system based on them.

Acknowledgment

I would like to thank my supervisor Gerald Ratzer for his guidance and encouragement throughout my stay at McGill University. This has been greatly appreciated.

I also truly thank Pierre Dumouchel who has accepted to co-supervise this work and allowed me to work on its relevant topic. The CRIM has provided me with a wonderful research environment.

I thank all members of the speech recognition group at CRIM. More particularly, Michel Comeau, Gilles Boulianne and Pierre Ouellet for their comments and help in the writing of this thesis; with Gilles I have especially had various constructive discussions about this work and other topics.

I thank also Marc Boulé for his ecouragement and help on various topics.

Résumé

La théorie des machines à états finis et des transducteurs à états finis est un sujet qui a été étudié en détail depuis plusieurs années. Récemment, la théorie des transducteurs a été généralisée par Mohri au cas des transducteurs pondérés. Cette généralisation a permis l'utilisation des transducteurs à états finis dans une grande variété d'applications comme, par exemple, la reconnaissance automatique de la parole. Dans ce travail, plusieurs algorithmes permettant la manipulation des transducteurs à états finis pondérés sont décrits et analysés en détail. Ensuite, un exemple de leur utilisation est donné en présentant un sytème de reconnaissance de la parole basé sur les transducteurs.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Finite-state automata have been extensively studied over the years. Originally, automata theory had been proposed to model brain functions [24]. This model is very useful for many other purposes and is now used in many important software such as compilers, speech recognition systems and bioinformatics.

Finite-state transducers are a generalization of the theory of automata. An automaton can be seen as a binary relation mapping a sequence of symbols to a binary value representing its acceptation value. Finite-state transducers generalize this behaviour by producing a sequence of symbols instead of a single binary value. These symbols are combined together according to their nature. Thus, finite-state transducers describe also a binary relation mapping a sequence of symbols to another sequence of symbols.

The use of finite-state machines is motivated by their computational efficiency. The time efficiency is achieved by using deterministic automata. In such machines, the generation of the output depends only on the length of the input sequence. From this point of view, sequential machines are considered optimal. The space efficiency is achieved with the classical minimization algorithm [1]. This algorithm ensures that the size of the automaton is minimal according to the language described. The efficiency of such automata has been proven in applications such as compiler design [2].

Several operations can be done on finite-state transducers. Some of them are borrowed from graph theory such as the shortest-path algorithm and depth-first search-based algorithms. Other operations are based on the more classic operations of automata theory. These operations have been generalized for weighted string-to-string transducers by Mohri. For example, the composition of transducers is a generalization of the intersection of automata. Several operations are fully described in Chapters 3 and 4. The running time of these algorithms is also analyzed.

Automata theory is widely used in traditional speech recognition since they represent efficient models for expressing language phenomena such as lexical rules [4, 25, 17]. The recent generalization of transducers to the weighted case by Mohri allowed the use of them to build a speech recognition system. The main advantage of this system over the traditional one is that all speech knowledge is expressed using the same transducer representation, allowing to make changes in the network without modifying the decoder [13]. Chapter 5 discusses how such recognition systems can be implemented.

To begin with, the mathematical foundations related to finite-state machines are introduced in the following chapter. This chapter gives the formal definition of many of the subjects related to transducers theory. Particularly, the formal definitions of the four types of transducers considered in this work are given. Some of the operations described in the following chapters are also introduced in this chapter.

Chapter 2

Basics of Finite-State Transducers

This chapter presents a brief introduction to the world of finite-state automata and finite-state transducers. Far from being exhaustive, the intended aim is to lay down the basic concepts and to introduce the notation used throughout. The uninitiated in the field can, as a complement to this chapter, consult the excellent books of Hopcroft [24] or Sipser [32] who present a more thorough introduction to the subjects of automata and language processing.

The first part of this chapter sketches the mathematical foundations of automata theory including language theory, the concepts of semirings and formal power series, and finally the formal description of automata. This part also introduces the reader to the terminology and notation used throughout this work.

The second part presents automata operations such as the union of two automata and their composition. Each operation will be briefly described; this will serve as a prelude to the next chapter in which algorithms performing these operations are presented and analysed.

Let us begin with the basic notion of automata: languages.

2.1 Alphabets, Strings and Languages

An alphabet is a finite and non-empty set of symbols generally denoted by the Greek letter Σ. Here are some examples of common alphabets:

$$\begin{aligned}\Sigma_1 &= \{0,1\}\\ \Sigma_2 &= \{a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z\}\end{aligned}$$

A string (or word) w is a finite sequence of symbols from a specified alphabet. For example, 010101 is a string over Σ_1 while *cat* is a string over Σ_2.

The length of a string, written $|w|$, is the number of symbols that it contains. An empty string has a length of 0 and is generally denoted as ϵ.

The set of all strings of length k over an alphabet Σ can be expressed using the exponential notation Σ^k. For example, if $\Sigma = \{0,1\}$ then $\Sigma^2 = \{00, 01, 10, 11\}$ while $\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$. Σ^* denotes the infinite set of all possible strings over Σ and is called the Kleene closure of Σ.

The binary operator $\cdot$ denotes the concatenation of two strings s_1 and s_2 performed by appending s_2 to s_1. For example, if $s_1 = \sigma_1\sigma_2...\sigma_n$ and $s_2 = \omega_1\omega_2...\omega_m$ then $s_1 \cdot s_2 = \sigma_1\sigma_2...\sigma_n\omega_1\omega_2...\omega_m$. Since concatenation is a kind of multiplication for strings, the n^{th} power of a string w, written w^n, is obtained by concatenating w with itself n times.

A *language* L on Σ is a set of strings chosen in Σ^* for the specified alphabet Σ.

2.2 Semiring

A semiring is an algebraic structure that can be used as an abstraction in the description of algorithms. Let us start by defining a smaller structure called a monoid. The semiring definition is based on monoids. A monoid M is set together with a binary operation and a neutral element.

Definition 2.1.
More formally, a monoid is a system $\mathcal{M} = (\mathbf{M}, \otimes, \bar{1})$ where :

- $\mathbf{M}$ is a set,
- $\otimes$ is an associative binary operator: $x \otimes (y \otimes z) = (x \otimes y) \otimes z, \forall x, y, z \in \mathbf{M}$,
- $\bar{1}$ is an identity element over $\otimes$: $x \otimes \bar{1} = \bar{1} \otimes x = x, \forall x \in \mathbf{M}$.

A monoid $(\mathbf{M}, \otimes, \bar{1})$ is said to be commutative if $x \otimes y = y \otimes x$ for all x and y in $\mathbf{M}$. A monoid can be designated only by $\mathbf{M}$ when the binary operation and the identity element are known.

An important monoid is the *free monoid* $(\Sigma^*, \cdot, \epsilon)$ where Σ^* is generated over a set Σ, $\cdot$ is the concatenation operator and ϵ is the empty string.

A semiring $\mathcal{K}$ contains two binary operators associated with a set $\mathbf{K}$ and two constant elements from $\mathbf{K}$ having some particular properties.

Definition 2.2.
A semiring $\mathcal{K} = (\mathbf{K}, \oplus, \otimes, \bar{0}, \bar{1})$ consists of two monoids such that:

- $(\mathbf{K}, \oplus, \bar{0})$ is a commutative monoid,
- $(\mathbf{K}, \otimes, \bar{1})$ is a monoid
- $\bar{0}$ is an annihilator: $x \otimes \bar{0} = \bar{0} \otimes x = \bar{0}, \forall x \in \mathbf{K}$.
- $\otimes$ distributes over $\oplus$ on the right: $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$ and on the left $(y \oplus z) \otimes x = (y \otimes x) \oplus (z \otimes x), \forall x, y, z \in \mathbf{K}$

As is the case for monoids, a semiring is called a *commutative* semiring if $x \otimes y = y \otimes x$ for every x and y in $\mathbf{K}$. A semiring is said to be *idempotent* if $x \oplus x = x$ for every x in $\mathbf{K}$.

There are three important semirings used in automata and transducers theory: the Boolean, the tropical and the string semiring. They are described as follows:

Boolean semiring

The Boolean semiring has a set with only two elements: true or false. The Boolean semiring is defined by $\mathcal{B} = (\{0,1\}, \vee, \wedge, 0, 1)$ where $\vee$ denotes the "or" operation and $\wedge$ denotes the "and" operation.

tropical semiring

The tropical semiring, presented in [29], is also called the min-plus semiring and is defined by $\mathcal{T} = (\mathbb{R}_+ \cup \{\infty\}, min, +, \infty, 0)$ where min denotes the classical minimum function and $+$ denotes the usual addition over real numbers.

string semiring

The string semiring defines operations on strings. The string semiring is defined as $\mathcal{S} = (\Sigma^* \cup \{\infty\}, \wedge, \cdot, \infty, \epsilon)$ where $a \wedge b$ denotes the longest common prefix of a and b, $\cdot$ is the concatenation of two strings, ∞ is a new element not in Σ such that the semiring properties are maintained and ϵ denotes the empty string.

The cross-product of two semirings is also a semiring. Given two semirings $\mathcal{K}_1 = (K_1, \oplus_1, \otimes_1, \bar{0}_1, \bar{1}_1)$ and $\mathcal{K}_2 = (K_2, \oplus_2, \otimes_2, \bar{0}_2, \bar{1}_2)$, their cross-product is defined as:

$$\mathcal{K}_1 \times \mathcal{K}_2 = (K_1 \times K_2, (\oplus_1, \oplus_2), (\otimes_1, \otimes_2), (\bar{0}_1, \bar{0}_2), (\bar{1}_1, \bar{1}_2))$$

2.3 Formal Power Series

Consider the mapping function $\alpha : \Sigma^* \longrightarrow K$ where Σ^* is a monoid and K is a semiring. This function is called a *formal power series* and is denoted by:

$$\alpha = \sum_{w \in \Sigma^*} \alpha(w) w$$

where $\alpha(w)$ is called the *coefficient* of w in α and $w \in \Sigma^*$ is the (noncommuting) variable. The set of all power series is denoted $K\langle\langle\Sigma^*\rangle\rangle$.

The *support* of α is the language defined by $supp(\alpha) = \{w \in \Sigma^* \mid \alpha(w) \neq \bar{0}\}$. Since the concept of support is a language, it brings a natural interconnection between the theory of formal power series and the theory of languages and thus, with automata.

2.4 Automata

Automata are a way to describe a set of strings and thus, represent a language. A language is called a *regular language* if and only if it can be represented by a finite automaton. Figure 2.1 depicts a simple automaton.

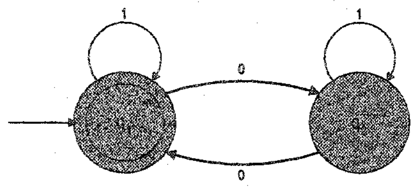


Figure 2.1: *Finite automaton with two states*

This automaton has two states labelled q_1 and q_2; the *initial state* is characterized by an arrow pointed to it from nowhere; the *final state*, also called *accepting state*, is represented by a double circle; the labelled arrows connecting two states are called *transitions*. In this example, q1 is both the initial and the final state.

An automaton processes an input string such as 1010 by following transitions from an initial state, depending on the symbols in the input string. Each symbol of the input string is consumed by the automaton from left to right. The output of the automaton is either to accept or to reject the input string. The string is accepted if after having processed all symbols of the input string, the automaton is in an accepting state. If not, the string is rejected by the automaton.

Thus, in the example of figure 2.1, the state sequence for the input string 1010 will be q_1, q_1, q_2, q_2, q_1. Since the last state q_1 is a final state, the string is accepted by this automaton.

Another interpretation of an automaton is to view it as a generator, rather than a consumer, of symbols: Starting from the initial state and following transitions produces a sequence of symbols, thus a string. The string is valid if the last state visited is a final state.

In the example of figure 2.1, the automaton accepts all strings that have an even number of 0's. Thus, the language is the set:

$$L(\mathcal{A}_1) = \{w \mid w \text{ is the empty string } \epsilon \text{ or has an even number of } 0's\}$$

Definition 2.3.
More formally, a finite automaton $\mathcal{A}$ is a 5-tuple (Q, i, F, Σ, E), where:

- Q is a set of states,
- $i \in Q$ is the initial state,
- $F \subseteq Q$ is the set of final states,
- Σ is the alphabet of $\mathcal{A}$,
- $E \subseteq Q \times \Sigma \times Q$ is the set of transitions.

Instead of a set of transitions, it is common to have a transition function mapping a state q and a symbol a to a destination state. More formally, this function is defined as $\delta : Q \times \Sigma \longrightarrow Q$. This function can be extended to $Q \times \Sigma^*$ using the following recurrence relation [17]:

$$\delta^*(q, wa) = \delta(\delta(q, w), a) \qquad \forall q \in Q, \forall w \in \Sigma^*, \forall a \in \Sigma \tag{2.1}$$

Thus, a string w is accepted by $\mathcal{A}$ if and only if $\delta^*(i, w) \in F$.

Path in Automata

A path π, also denoted $q_1 \rightsquigarrow q_2$ is a sequence of consecutive transitions from state q_1 to q_2. The length $|\pi|$ of the path π is the number of transition making up this path.

Definition 2.4.
More formally, a path is a sequence of transitions $\pi = (q_1, \sigma, q_1')...(q_{|\pi|-1}, \sigma, q_{|\pi|})$ such that $q_i' = q_{i+1}, i = 1, ..., |\pi| - 1$.

2.4.1 Weighted Automata

Weighted automata, also called weighted acceptors, output a weight depending on the input string and not simply a reject/accept value. The weight carried by transitions along the symbols are $\oplus$-additionned according to a given weight semiring such as the tropical semiring or the log semiring. The choice of the semiring should reflect the intended interpretation of the weights. Figure 2.2 shows a weighted acceptor.

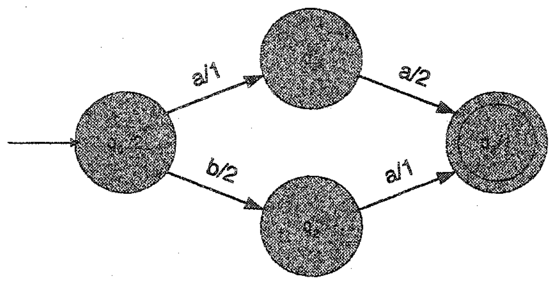


Figure 2.2: *Example of a string-to-weight transducer*

The weight associated with a string takes into account the output weights of transition but also a weight associated with the initial state and another weight associated with the final state.

Definition 2.5.
More formally, a weighted acceptor $\mathcal{A}$ over a semiring K is a 7-tuple $(Q, i, F, \Sigma, E, \lambda, \rho)$, where:

- Q is the set of states,
- $i \in Q$ is the initial state,
- $F \subseteq Q$ is the set of final states,
- Σ is the alphabet of the automaton,
- $E \subseteq Q \times \Sigma \times K \times Q$ is the set of transitions,
- $\lambda : i \longrightarrow K$ is the initial weight function,
- $\rho : F \longrightarrow K$ is the final weight function.

The set of transitions can be replaced by a transition function, as is the case for non-weighted automata, and by an output function mapping a state q and a symbol a to a weight semiring. More formally, the output function is defined as $\sigma : Q \times \Sigma \longrightarrow K$. As is the case for the transition function, the function can be extended to $Q \times \Sigma^*$ using the following recurrence equation [17]:

$$\sigma^*(q, wa) = \sigma(q, w) \cdot \sigma^*(\delta(q, w), a) \qquad \forall q \in Q, \forall w \in \Sigma^*, \forall a \in \Sigma \tag{2.2}$$

Thus, if the string w is accepted by $\mathcal{A}$, its output will be $\sigma(i, w)$.

2.4.2 Epsilon Transitions

An epsilon or null transition is one that does not consume any input symbol. In the graph representation, the epsilon is denoted by the Greek symbol ϵ. Figure 2.3 shows an example of an automaton with ϵ-transitions.

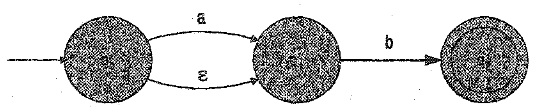


Figure 2.3: *Automaton with ϵ-transitions*

The language accepted by this automaton is $\{ab, b\}$. Since no input symbols are consumed when an ϵ-transition is taken, the language accepted by the automaton is not influenced by it. However, the creation of automata is often simplified by using epsilons.

2.4.3 Determinism

A finite-state automaton is called deterministic (DFA) if and only if for any input string w, the sequence of states is unique. Figure 2.4a shows a non-deterministic finite-state automaton (NDFA) since there are two transitions with the symbol a going out of state q_0. Figure 2.4b shows a deterministic automaton accepting the same language as the automaton of Figure 2.4a.

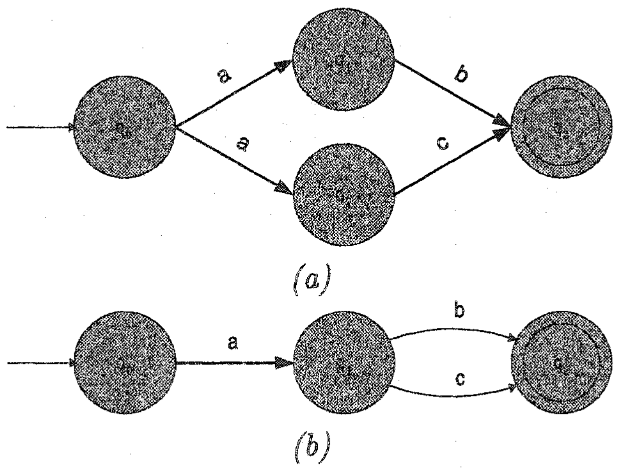


Figure 2.4: *Non-deterministic and deterministic automata*

Definition 2.6.
More formally, an automaton $(Q, i, F, \Sigma, \delta)$ is deterministic if:

$$|\delta^*(q, w)| \leq 1 \qquad \forall q \in Q, \forall w \in \Sigma^*$$

Every language that can be described by a NDFA can also be described by a DFA [24, 32]. This property helps with the design of automata since it is often easier to construct a new automaton as NDFA and then to transform it to a DFA. Since DFAs are computationally more efficient, this operation is very useful.

2.4.4 Equivalence of Automata

For a given language there exists an infinite number of ways to construct automata representing this language. These automata are said to be equivalent.

Definition 2.7.
More formally, two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are equivalent if and only if $L(\mathcal{A}_1) = L(\mathcal{A}_2)$.

2.5 Finite-State Transducers

Transduction is the process which maps an input string w_i over the alphabet Σ_i to an output string w_o over the alphabet Σ_o.

Definition 2.8.
A transduction is a mapping function defined as $\mathcal{T} : \Sigma_i^* \longrightarrow \Sigma_o^*$ where Σ_i^* is the set of input strings and Σ_o^* is the set of output strings.

Definition 2.9.
A weighted transduction is a mapping function defined as $\mathcal{T} : \Sigma_i^* \longrightarrow \Sigma_o^* \times K$ where Σ_i^* is the set of input strings, Σ_o^* is the set of output strings and K is a weight semiring.

Transducers are a type of automaton whose transitions carry an output symbol in addition to the input symbol. Thus, the output of a transducer is a string over a given alphabet and not just a weight or a reject/accept value as with automata.

2.5.1 String-To-String Transducers

A string-to-string transducer represents the function $T : \Sigma_i^* \longrightarrow \Sigma_o^*$ where Σ_i^* and Σ_o^* are the sets of input and output strings. Figure 2.5 shows an example of a string-to-string transducer. In this example, the string *aa* is mapped to the string *cd* while the string *ba* is mapped to the string *ec*. All other strings are rejected by the transducer.

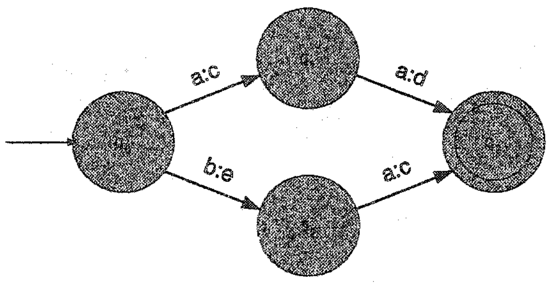


Figure 2.5: *Example of a string-to-string transducer*

Definition 2.10.
More formally, a string-to-string transducer $\mathcal{T}$ over a semiring K is a 6-tuple $(Q, i, F, \Sigma_i, \Sigma_o, E)$, where:

- Q is the set of states,
- $i \in Q$ is the initial state,
- $F \subseteq Q$ is the set of final states,
- Σ_i is the input alphabet of the automaton,
- Σ_o is the output alphabet of the automaton,
- $E \subseteq Q \times \Sigma_i \times \Sigma_o \times K \times Q$ is the set of transitions.

As is the case for acceptors, the set of transitions can be replaced by a transition function and an ouput function. The transition function is the same as for acceptors while the ouput function becomes $\sigma : Q \times \Sigma_i \longrightarrow \Sigma_o$. Both functions can be extended using the recurrence relations expressed in equations 2.1 and 2.2.

2.5.2 Weighted String-To-String Transducers

This kind of transducer is the most general finite-state automaton discussed in this work. It maps a pair consisting of an output string and a weight.

More formally, the mapping function of a weighted string-to-string transducer is $T : \Sigma_i^* \longrightarrow \Sigma_o^* \times K$ where Σ_i^* and Σ_o^* are the sets of input and output strings respectively and K is a weight semiring. Figure 2.6 shows a weighted string-to-string transducer.

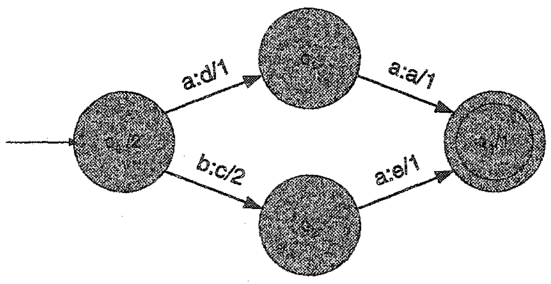


Figure 2.6: *Example of a weighted string-to-string transducer*

As is the case for weighted acceptors, a weighted string-to-string transducer also provides an initial and a final weight.

Definition 2.11.
A weighted string-to-string transducer $\mathcal{T}$ over a semiring K is a 8-tuple $(Q, i, F, \Sigma_i, \Sigma_o, E, \lambda, \rho)$, where:

- Q is a set of states,
- $i \in Q$ is the initial state,
- $F \subseteq Q$ is the set of final states,
- Σ_i is the input alphabet of the automaton,
- Σ_o is the output alphabet of the automaton,
- $E \subset Q \times \Sigma_i \times \Sigma_o \times K \times Q$ is the set of transitions,
- $\lambda : i \longrightarrow K$ is the initial weight function,
- $\rho : F \longrightarrow K$ is the final weight function.

As is the case for string-to-string transducers, the set of transitions can be replaced by a transition function and an output function. The transition function is identical to that of the string-to-string transducer and the output function becomes $\sigma : Q \times \Sigma_i \longrightarrow \Sigma_o \times K$. Both functions can be extended using the recurrence relations expressed in equations 2.1 and 2.2.

2.5.3 Epsilon Symbols in String-To-String Transducers

As is the case for automata, epsilon symbols are allowed in string-to-string transducers both for input and output symbols. An input string and its corresponding output string do not necessarily have the same length. Thus, epsilons are used to fill the "blanks".

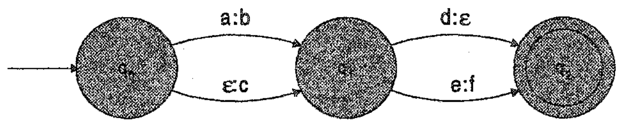


Figure 2.7: *Example of a transducer using epsilons.*

Figure 2.7 shows a transducer using epsilons to map strings of different length. In a transducer, ϵ-transitions are represented by a transition with an input and output epsilon.

2.5.4 Sequential Transducers

A transducer is called sequential if it is deterministic from the point of view of its input. Figure 2.8a shows a non-sequential transducer since there are two transitions with the symbol a outgoing from state q_0. Figure 2.8b shows a sequential transducer.

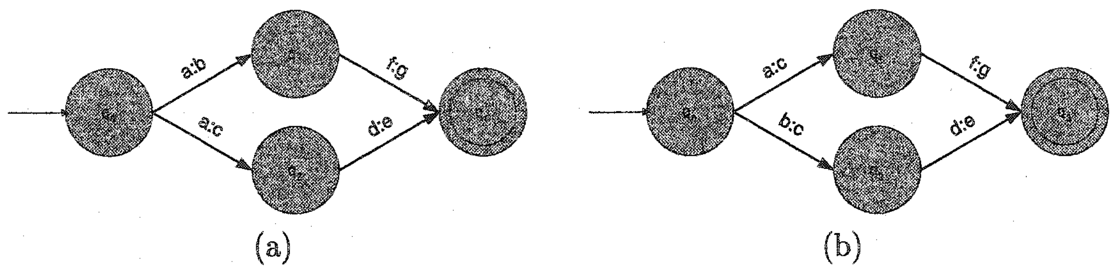


Figure 2.8: *A non-sequential and a sequential transducer*

The empty string, namely ϵ, is not allowed as an input symbol in a sequential transducer. Sequential transducers are computationally efficient since the time requirements depend only on the size of the input string and not on the size of the transducer. This efficiency comes from the fact that for a given input string, the output string is written by following the only corresponding path.

2.6 Operations on transducers

As is the case for automata, many operations are available for working with transducers. This section will briefly describe these operations.

2.6.1 Union

Union is a basic operation in automata theory. The union of two languages L_1 and L_2 is the set of strings that are in either L_1, L_2 or both. More formally, the union $L_1 \cup L_2 = \{x | x \in L_1 \text{ or } x \in L_2\}$.

For transducers, this operation is done by combining the initial states of both transducers. Figure 2.9 shows an example of the union of two simple transducers over a semiring K.

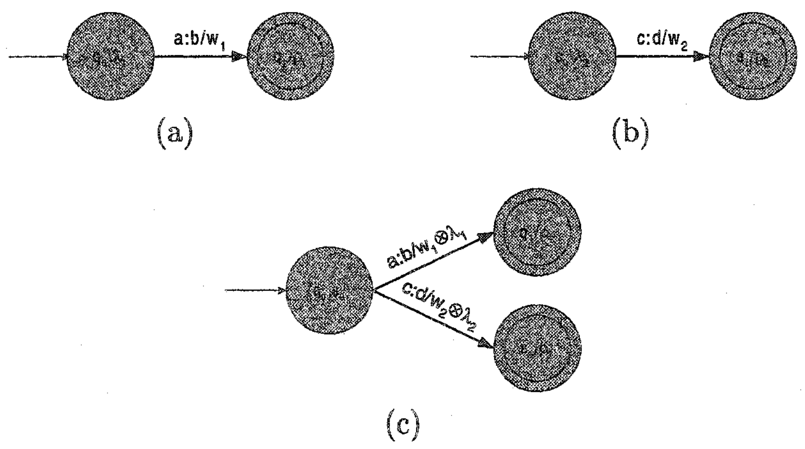


Figure 2.9: *Example of weighted transducer union*

To be consistent with the weighting of original transducers, the initial weights of both transducers have been moved to outgoing transitions of the initial state using the $\otimes$-product.

2.6.2 Concatenation

The concatenation of two languages L_1 and L_2 is the set of strings formed by concatenation of all strings in L_1 with strings of L_2. More formally, the concatenation $L_1 \cdot L_2 = \{x \cdot y | x \in L_1 \text{ and } y \in L_2\}$.

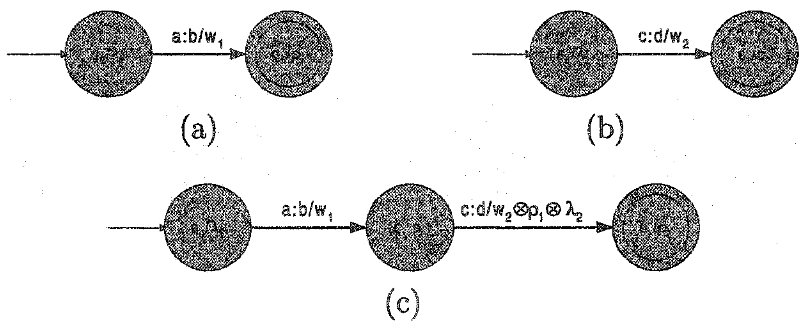


Figure 2.10: *Example of weighted transducer concatenation*

From the point of view of transducers, the concatenation of two transducers T_1 and T_2 is obtained by appending T_2 to the end of T_1 by merging the final state(s) of T_1 with the initial state of T_2. Figure 2.10 shows an example of the concatenation of two simple transducers over a semiring $\mathbb{K}$.

To be consistent, the final weight of the first transducer and the initial weight of the second transducer are moved to transitions going out of the states merged during the operation. These weights are combined with transition weights using the $\otimes$-product.

2.6.3 Connection

This operation removes from a given transducer all unconnected states. A state q is *accessible* if there exists a path from the initial state to q and is *coaccessible* if there exists a path from q to a final state. A state is said to be connected if it is both accessible and coaccessible. Figure 2.11a shows a transducer with a non-coaccessible (q_1) and a non-accessible state (q_4). Figure 2.11b shows the same transducer without these useless states.

In the removal process, all transitions going out from and going into an unconnected state are also deleted. This operation is often used to clean up the result of other operations such as composition which leaves some unconnected states.

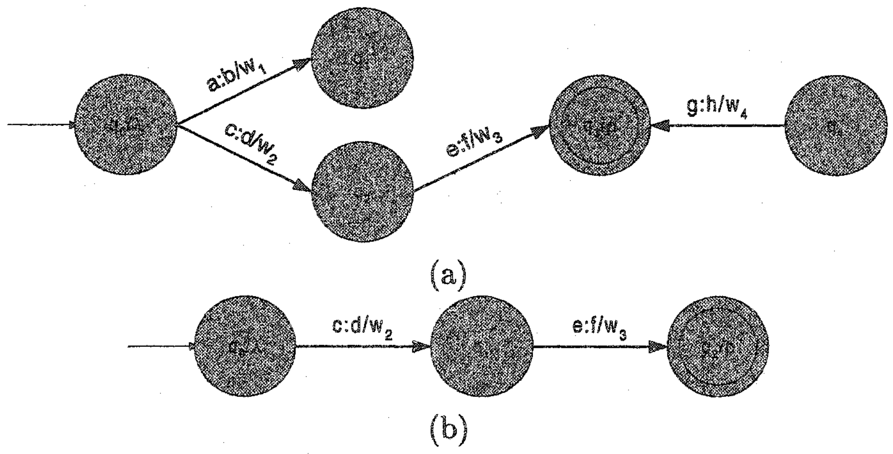


Figure 2.11: *Example of trimming*

2.6.4 Reverse

This operation consists of reversing all transitions of the given transducer. The operation also transform final states into an initial state and the initial state into a final state. The reverse operation is denoted by $T_{res} = T_{in}^{r}$. Figure 2.12b shows the reverse of transducer of figure 2.12a.

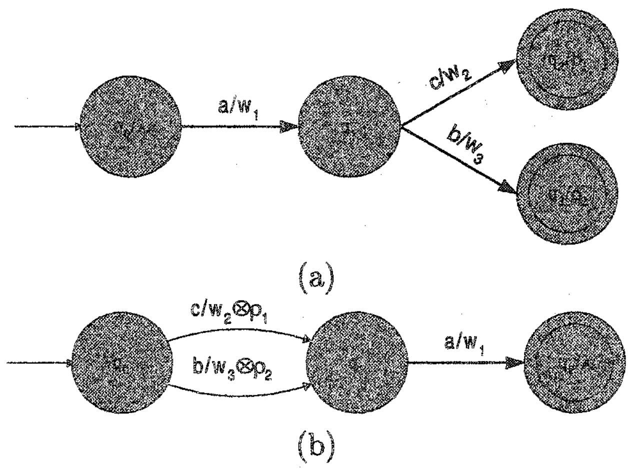
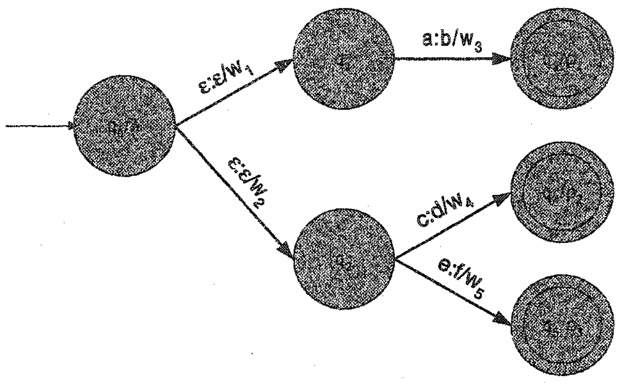


Figure 2.12: *Example of transducer reversal*

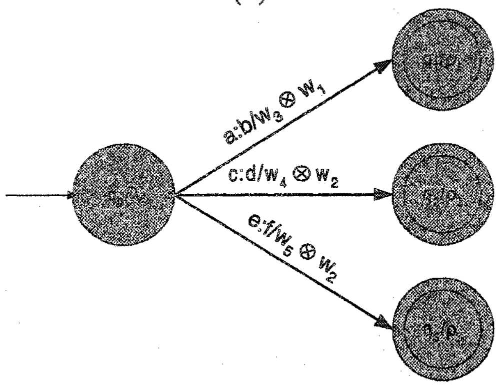
Note that applying the reversal operation twice on a transducer T produce a new transducer equivalent to T in which there is only one final state *i.e.* $|F| = 1$.

2.6.5 Removing Epsilons

Transducers are often constructed with ϵ-transitions. Unfortunately, these transitions decrease the computational efficiency of FST since they make them non-deterministic. This operation of epsilon removal produces an equivalent transducer with no ϵ-transitions. Figure 2.13 shows an example of this operation on a transducer.



(a)



(b)

Figure 2.13: *Example of removing epsilons on a transducer*

The determinization operation, which transform a non-deterministic transducer into a deterministic one, generally considers epsilons as an ordinary symbol and thus, determinization keeps ϵ-transitions. For this reason, it is common to remove epsilons before applying determinization to obtain a deterministic automaton or a sequential transducer.

2.6.6 Composition

Composition is a generalization of the intersection operation for automata. This operation is very useful since it allows the construction of complex transducers from simpler ones. Figure 2.14 shows a cascade of two transducers.

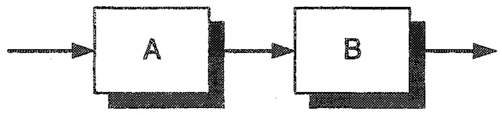


Figure 2.14: *A cascade of two transducers*

The transducer A maps Σ_i^* to Δ^*. Thus, the set Δ^* becomes the input of transducer B that maps Δ^* to Σ_o^*. Therefore, the general behaviour of the cascade is: $A \circ B = \Sigma_i^* \longrightarrow \Sigma_o^*$. The composition creates the transducer equivalent to this cascade.

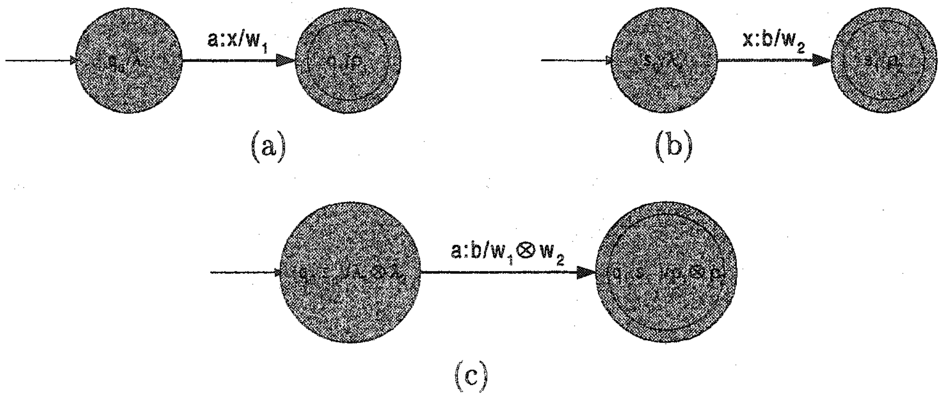


Figure 2.15: *Example of transducer composition*

Given a transducer A in which there is a path mapping sequence x to sequence y and a transducer B in which there is a path mapping sequence y to sequence z, the composition $A \circ B$ has a path mapping x to z. The weight of this path is the $\otimes$-product of the weights of the corresponding path in A and B [19]. Figure 2.15 shows two simple transducers and the result of their composition.

The composition is a key operation in transducer based application since it is used to construct complex transducers representing complex functions. For example, in the case of speech recognition, the composition is used to construct the knowledge network needed by the recognition system. This network is constructed by the composition of different level of representation for which transducers are associated. The construction of this network will be described in detail later.

2.6.7 Determinization

Deterministic automata and sequential transducers have already been defined. Any non-deterministic automaton has an equivalent deterministic one. Determinization is the process which takes a non-deterministic automata as input and produces a deterministic one as output. Figure 2.16b shows a deterministic automaton constructed from the automaton of figure 2.16a.

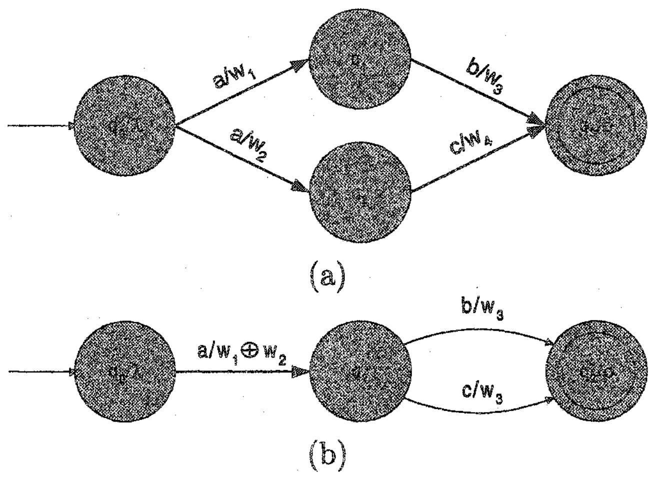


Figure 2.16: *Example of transducer determinization*

Deterministic automata are computationally more efficient but in practice, the number of states involved is often greater than the equivalent non-deterministic counterpart. In the worst case, the smallest deterministic automaton can have 2^n states while the smallest non-deterministic automaton describing the same language has n states.

The same operation can be applied to non-sequential transducers to obtain sequential ones. Unfortunately, this process does not terminate for all transducers. This point will be discussed in the next chapter.

2.6.8 Minimization

Given the complete set of equivalent deterministic automata, there exists a unique automaton which has a minimum number of transitions and (arbitrarily labelled) states with respect to the implied language. Figure 2.17b shows the minimized version of the automaton of figure 2.16a.

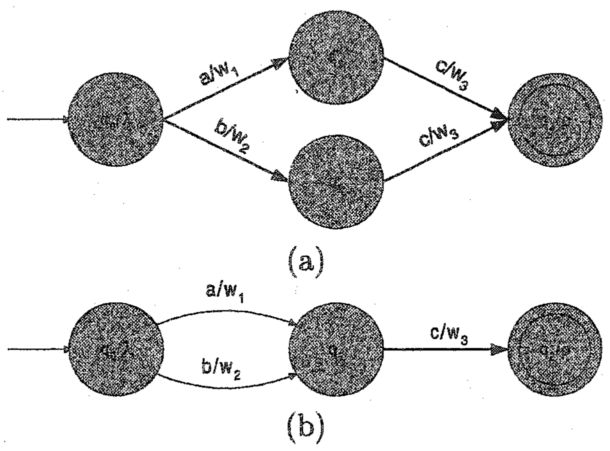


Figure 2.17: *Example of transducer minimization*

The minimization for weighted transducers requires two steps. The first step is a reweighting operation called pushing. A transducer can be reweighted in an infinite number of ways. The *pushing* operation moves the weights toward the initial state. The result is a transducer as seen on figure 2.17b, which contains some transitions having the same symbol and the same weight. The second step is the classical minimization process that considers the symbol and the weight as a single symbol. Since the pushed transducer has some transitions with identical (symbol,weight) pair, it can be minimized.

This procedure can be applied to string-to-string transducers but does not necessarily yields the minimum transducer. However, the method can be useful for reducing the transducer's size and is called *compaction* in this case.

The minimization of transducers is performed by consecutively applying the determinization, reverse, determinization and reverse operations. Mohri has proved the optimality of this algorithm in [17]. Unfortunately, not all transducers can be determinized. In that case, the only solution is to use compaction since there does not exist a minimization algorithm that can be applied to non-determinizable transducers.

Therefore, the minimization of transducers is based on the classical minimization algorithm presented in [1]. This algorithm will not be described in this work since it is a classical one. For any reader interested in it, the algorithm is presented in [1] and [24]. Moreover, the compaction has been fully studied by Zhang [35].

2.6.9 Other Operations

The major FST operations have been presented but there exists some other useful manipulations that can be done on a FST, and are briefly described here:

Inversion

Invert the transducer by swapping the input and output symbols on transitions.

Arithmetic

Apply some arithmetic operation (addition or multiplication) on weights of weighted FSM.

Projection

Convert a transducer to an acceptor by keeping either only the input or only the output symbol.

Best paths

Find the k paths of lowest weight from the initial state to a final state in a weighted FSM.

Topological sort

This operation numbers states such that for any transition from a state numbered i to a state numbered j, the condition $i \leq j$ is respected.

Algorithms for performing these operations will also be given in the next chapter.

2.7 Summary

This chapter has introduced some theorical aspects of weighted finite-state transducers in order to present the basic concepts and to introduce the notation used in the algorithm descriptions presented in the next chapters. The important points discussed in this chapter are:

- The mathematical foundations of automata theory including language theory, semiring and formal power series.
- The formal definitions of the different kinds of transducers for which the algorithms in the next chapters can be applied.
- A brief overview of operations described and analyzed in greater detail in the next chapters.

Chapter 3

Basic Algorithms

This chapter presents some basic algorithms applicable to finite-state transducers. It is divided into five sections.

The first section presents the union operation, which is a fundamental operation in automata theory. In the second section will be presented another important operation in automata theory: the concatenation of two finite-state transducers. Both of these operations are discussed in all introductory books to the automata theory.

The third section will explore algorithms based on the depth-first search method, first introduced by Tarjan [33]. In particular, this section describes the topological sort algorithm, which sorts the states in a left to right order and the connection algorithm, which removes the unconnected states of a transducer.

The fourth section concerns the shortest-path problem. A generic shortest-distance algorithm will be presented. This algorithm is generic in the sense that it can be implemented with a large variety of semirings and queue disciplines. In the second part of the section, the classical shortest-path algorithm introduced by Dijkstra [8] will be presented in the context of transducers. A generalization of this algorithm, resolving the k-shortest-path problem, will also be described.

The last section describes a pushing algorithm used to move weights along the paths toward the initial state.

3.1 Union

The union of two languages L_1 and L_2 is a new language obtained by combining all words of L_1 and $L2$ in a new set L_3 denoted by $L_1 \cup L_2$.

Definition 3.1.
More formally, the union of two languages L_1 and L_2 is defined as:

$$L_1 \cup L_2 = \{x \mid x \in L_1 \vee x \in L_2\}$$

Regular languages are closed under the union operation [32]. That means that if L_1 and L_2 are regular languages, then the union $L_1 \cup L_2$ is also a regular language. This property implies that the union can be applied to transducers which represent regular languages.

Formally speaking, the union of transducers is obtained by combining their initial states. In practice, however, ϵ-transitions carrying initial weights are used to "merge" together both transducers as illustrated in Figure 3.1. Algorithm 1 shows the pseudocode of a procedure which makes the union of the two input transducers $T_1 = (Q_1, i_1, F_1, \Sigma_{i1}, \Sigma_{o1}, E_1, \lambda_1, \rho_1)$ and $T_2 = (Q_2, i_2, F_2, \Sigma_{i2}, \Sigma_{o2}, E_2, \lambda_2, \rho_2)$.

Algorithm 1 FST Union

```
FSTUnion($T_1, T_2$)
1: $q_o \leftarrow newstate$
2: $i_o \leftarrow q_o$
3: $Q_o \leftarrow \{q_o\} \cup Q_1 \cup Q_2$
4: $Eo \leftarrow \{(q_o, \epsilon, \epsilon, \lambda_1(i_1), i_1), (q_o, \epsilon, \epsilon, \lambda_2(i_2), i_2)\} \cup E_1 \cup E_2$
5: $F_o \leftarrow F_1 \cup F_2$
6: **return** $(Q_o, i_1, F_o, \Sigma_{i1} \cup \Sigma_{i2}, \Sigma_{o1} \cup \Sigma_{o2}, \lambda_o, \rho_o)$
```

The algorithm works as follows. At lines 1-2, the initial state q_o of the output transducer T_o is created. Then, the set of states Q_1,Q_2 and the new initial state are merged to create Q_o, the set of states of the output transducer. Line 4 creates transitions of T_o from transitions of both input transducers. Moreover, two new ϵ-transitions, which connect q_o to both initial states i_1 and i_2, are also added to T_o. Note that since these new transitions do not carry symbols, the languages described by T_1 and T_2 are still correctly represented by T_o. Finally, line 5 combines the accepting states of both input transducers and line 6 returns the new transducer created.

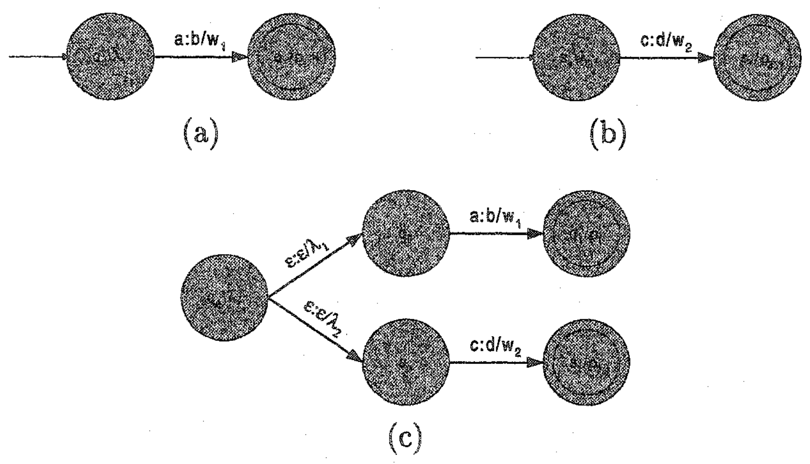


Figure 3.1: *Result of the union of two transducer by FSTUnion.*

Note that the result is not exactly the same that shown in section 2.6.1. This is because the algorithm described here uses ϵ-transitions to simplify the construction of the resulting transducer. On the other hand, the result shown in Figure 2.9 can be obtained by removing ϵ-transitions of the transducer created by FSTUnion. The algorithm performing this operation will be presented in the next chapter.

Running Time Analysis

The running time of this algorithm depends on how both state and transition sets are implemented. Indeed, using linked lists for these sets, the algorithm can run in $\mathcal{O}(1)$ since, in this case, the union of sets corresponds to the concatenation of the lists.

Other implementations use arrays to represent sets. In this case, the union of sets implies a loop which will pass through all the elements of both sets. Considering that, the union operation of line 3 will take $\mathcal{O}(|Q_1| + |Q_2|)$ time and the union of line 4 will take $\mathcal{O}(|E_1| + |E_2|)$. Therefore, the running time of this algorithm is linear:

$$\mathcal{O}(|Q| + |E|)$$

where $|Q| = |Q_1| + |Q_2| + 1$ and $|E| = |E_1| + |E_2| + 2$ are respectively the number of states and transitions in the resulting transducer.

3.2 Concatenation

Recall that the concatenation of two words w_1 and w_2 is obtained by appending w_2 at the end of w_1. The concatenation of two languages L_1 and L_2 is a new language obtained by appending every word of L_2 at the end of every word of L_1. This operation is usually denoted by a dot.

Definition 3.2.
More formally, the concatenation of two languages L_1 and L_2 is defined as:

$$L_1 \cdot L_2 = \{x \cdot y \mid x \in L_1 \wedge y \in L_2\}$$

As is the case for the union operation, the regular languages are closed under the concatenation operation[32]. Therefore, the concatenation of two regular languages results in a third language which is also a regular language. It follows that the concatenation of two regular languages can also be represented by a transducer.

From the perspective of transducers, the concatenation of two transducers T_1 and T_2 is obtained by merging the initial state of T_2 with all the final states of T_1. In practice, it is easier to use ϵ-transitions to "connect" both transducers together as shown in Figure 3.2. Algorithm 2 shows the pseudocode performing the concatenation of transducers $T_1 = (Q_1, i_1, F_1, \Sigma_{i1}, \Sigma_{o1}, E_1, \lambda_1, \rho_1)$ and $T_2 = (Q_2, i_2, F_2, \Sigma_{i2}, \Sigma_{o2}, E_2, \lambda_2, \rho_2)$ which are the input of the procedure.

Algorithm 2 FST Concatenation

```
FSTConcatenation($T_1, T_2$)
1: $Q_o \leftarrow Q_1 \cup Q_2$
2: $E_o \leftarrow E_1 \cup E_2$
3: for each $q \in F_1$ do
4:     $E_o[q] \leftarrow E_o[q] \cup \{(q, \epsilon, \epsilon, \rho_1(q) \otimes \lambda_2(i_1), i_1)\}$
5: return $(Q_o, i_1, F_2, \Sigma_{i1} \cup \Sigma_{i2}, \Sigma_{o1} \cup \Sigma_{o2}, \lambda_1, \rho_2)$
```

Lines 1-2 initialize the set of states and the set of transitions of the new transducer. The loop at lines 3-4 add a new ϵ-transition from every final state $q \in F_1$ to the initial state of T_2. The weight of every added transition corresponds to the $\otimes$-product of the acceptation cost of the final state from which the transition going out and the initial cost of T_2. Line 5 returns the new transducer created.

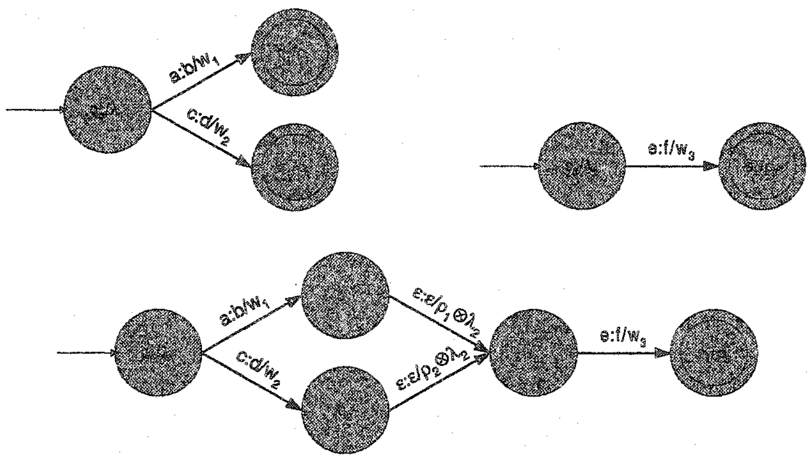


Figure 3.2: *Result of the concatenation of two transducer by FSTConcatenation.*

Figure 3.2 shows two transducers and their concatenation as computed by Algorithm 2. As is the case for the union operation previously described, the ϵ-transitions can be removed to obtain the same result shown in figure 2.10.

Running Time Analysis

The running time of this algorithm depends on how the sets are implemented. Indeed, using a linked list for the sets of states and transitions allows an implementation of the concatenation operation in $\mathcal{O}(1)$ time. Therefore, lines 1-2 run in $\mathcal{O}(1)$. The loop at lines 3-4 pass through all final states. In the worst case, all states of the transducer are final and thus, the running time of this loop is $\mathcal{O}(|Q|)$. The total running time is therefore $\mathcal{O}(|Q|)$.

In the case where the sets are implemented with data structure such as arrays, lines 1-2 pass through each state and transition to copy them. Thus, the running time is $\mathcal{O}(|Q_1| + |Q_2|$ for the union of state sets (line 1) and $\mathcal{O}(|E_1| + |E_2|)$ for the union of transitions set (line 2). The running time of the loop at lines 3-4 is not affected by the sets' implementations. Therefore, the total running time is linear:

$$\mathcal{O}(|Q| + |E|)$$

where $|Q| = |Q_1| + |Q_2|$ and $|E| = |E_1| + |E_2| + |F_1|$ are, respectively, the number of states and transitions in the resulting transducer.

3.3 Depth-First Search Algorithms

Depth-First Search (DFS) is a simple algorithm for searching in a FST and is used as a base for many other algorithms. The algorithm is similar to what is used in graph theory [33, 7, 30]. The strategy is, as its name implies, to explore the transducer "deeper" whenever it is possible. The search begins in state s which is marked as visited. Then, the search is recursively applied to all adjacent states to s. The process continues until all states in the transducer have been visited. Figure 3.3 shows in which order states are visited by the DFS.

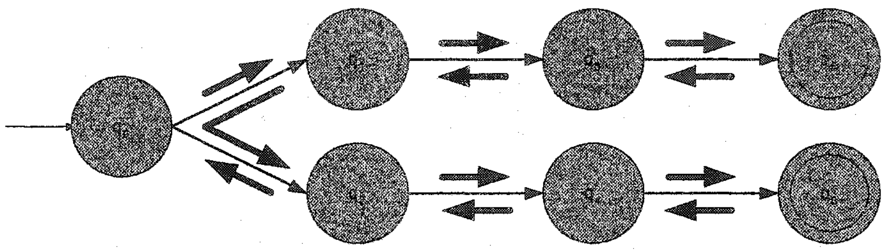


Figure 3.3: *Example of DFS execution*

The coloring method is used to mark states. White denotes a state which has not yet been visited. Grey denotes a state for which exploring adjacent states is in progress and black denotes a state for which all adjacent states have been visited.

Algorithm 3 Depth-First Search

```
DFS(T)
1: for all $q \in Q$ do
2:     color[q] ← white
3: for $q \in Q$ do
4:     if color[q] is white then
5:         DFS-Visit$(T, q)$

DFS-Visit(T,q)
1: $color[q] \leftarrow Grey$
2: for each $(q, \sigma_i, \sigma_o, w, q') \in E[q]$ do
3:     if color$[q']$ is white then
4:         DFS-Visit$(T, q')$
5: $color[q] \leftarrow Black$
```

Algorithm 3 depicts the pseudocode of the depth-first search algorithm in two procedures. The input of this algorithm is a transducer $T = (Q, i, F, \Sigma_i, \Sigma_o, E, \lambda, \rho)$.

The algorithm works as follows. Lines 1-2 initialize all states by painting them white. Lines 3-5 visit all white states using DFS-Visit(). In each call to DFS-Visit, the state q is initially white. Line 1 paints it grey. Lines 2-4 recursively explore each white state adjacent to q. Finally, when all adjacent states have been explored, line 5 paints it black.

Running Time Analysis

The running time of loops on lines 1-2 and 3-5 of DFS depends on the number of states in T, thus DFS is $\mathcal{O}(|Q|)$ when the call to DFS-Visit is not taken into account.

The DFS-Visit procedure is called exactly once for each state since the procedure is called only on white states and painting it grey is the first thing that DFS-Visit does. The loop on lines 2-4 of DFS-Visit is executed $|E[q]|$ times, thus:

$$\mathcal{O}(\sum_{q \in Q} |E[q]|) = \mathcal{O}(|E|).$$

Therefore, the total running time of DFS is linear : $\mathcal{O}(|Q| + |E|)$.

3.3.1 Topological Sort

A topological sort of a transducer T is a linear ordering of all its states such that for every transition $(q, \sigma_i, \sigma_o, w, q')$, q is smaller than q' *i.e.* q appears before q'. By definition, a cyclic transducer cannot be topologically sorted since in a cycle, there is always a transition such that $q > q'$.

Algorithm 4 shows the pseudocode for the topological sort. The algorithm input is a transducer $T = (Q, i, F, \Sigma_i, \Sigma_o, E, \lambda, \rho)$. The output is a list of topologically ordered states.

Two changes have been made in the original DFS procedure to construct the TopologicalSort procedure. Firstly, a FIFO (first in, first out) list is initialized at line 1. Line 4 ensures that the ordering process will begin at the initial state so that the first state in the list is the initial state.

Algorithm 4 Topological sort

```
TopologicalSort(T)
1: LIST $\leftarrow \emptyset$
2: for all $q \in Q$ do
3:    color[q] $\leftarrow$ white
4: DFS-Visit($T, T.InitialState$)
5: for $q \in Q$ do
6:    if color[q] is white then
7:       DFS-Visit($T, q$)
8: return LIST

DFS-Visit(T,q)
1: $color[q] \leftarrow Grey$
2: LIST $\leftarrow$ q
3: for each $(q, \sigma_i, \sigma_o, w, q') \in E[q]$ do
4:    if color[$q'$] is white then
5:       DFS-Visit($T, q'$)
6: $color[q] \leftarrow Black$
```

In the DFS-Visit procedure, the only change appears at line 2 and consists in inserting the state q in the list. Note that if line 2 is moved after the loop of lines 3-5, the inverse topological order will be obtained.

Note that in practice, the output list will be used to create a new transducer in which states will be inserted in the data structure representing the topologically ordered set of states.

Running Time Analysis

The running time of the TopologicalSort procedure, excluding the call to DFS-Visit, is the same as for the DFS procedure previously analyzed since the call to DFS-Visit appearing before the loop of lines 5-7 does not change the fact that DFS-Visit is called once per state. Thus, its complexity is also $\mathcal{O}(|Q|)$.

In the case of the DFS-Visit procedure, the inserting operation has to be taken into account. Since the list is a FIFO data structure, a new element is inserted in constant time; insertion is thus $\mathcal{O}(1)$. Since the procedure differs from the original only by this operation, its running time is also $\mathcal{O}(|E|)$. Therefore, the topological sort runs in linear time : $\mathcal{O}(|Q| + |E|)$.

3.3.2 Connection (Trimming)

The aim of this algorithm is to remove all unconnected states from a transducer. This operation is often used to clean up the result of other operations that yield unconnected states. Recall that a state q is connected if

1. it is accessible: there exists a path from the initial state to q,
2. it is coaccessible:there exists a path from q to a final state.

Figure 3.4 shows a transducer with accessible and coaccessible states. In this figure, "a" denotes an accessible state and "c" a coaccessible state.

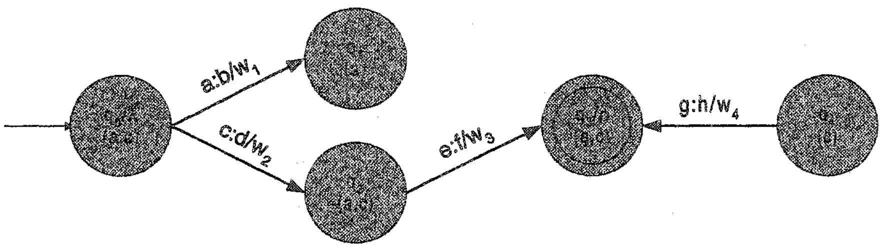


Figure 3.4: *Transducer with accessible and coaccessible states.*

The intuitive way to implement this operation takes three steps. The first step consists in performing a depth first-search from the initial state and marking all reachable states as accessible. The second step is to perform another depth-first search from the final states and marking all reachable states as coaccessible. Finally, the last step consists in removing all states that are not simultaneously accessible and coaccessible. Note that transitions going out from and going into an unconnected state are also removed.

In fact, the entire process can be done in one depth-first search as shown in Algorithm 5. The input of this algorithm is a transducer $T = (Q, i, F, \Sigma_i, \Sigma_o, E, \lambda, \rho)$ with output T trimmed of all unconnected states.

The algorithm works as follows. Lines 1-3 paint all states white and label them as unconnected. Line 4 starts the depth-first search at the initial state to ensure that all states which will be reached fulfill the first condition of a connected state. Lines 5-7 remove all unconnected states from Q.

Algorithm 5 Connection

```
Connection(T)
1: for each $q \in Q$ do
2:    color[q] $\leftarrow$ white
3:    connected[q] $\leftarrow$ false
4: DFS-Visit($T, T.InitialState$)
5: for each $q \in Q$ do
6:    if connected[q] = false then
7:       Remove q from Q
8: return $T$

DFS-Visit(T,q)
1: $color[q] \leftarrow Grey$
2: for each $(q, \sigma_i, \sigma_o, w, q') \in E[q]$ do
3:    if color[$q'$] is white then
4:       DFS-Visit($T, q'$)
5:       $connected[q] \leftarrow connected[q] \wedge connected[q']$
6: if $q \in F$ then
7:    connected[q] $\leftarrow$ true
8: $color[q] \leftarrow Black$
```

The DFS-Visit procedure is used to find which states are connected. The algorithm first searches final states and marks them as connected (lines 6-7) since they are reachable from the initial state. Each time an adjacent state of q has been explored (line 4), its connection property is propagated (lines 5) to the state itself. Indeed, the adjacent state has been marked connected only if a final state has been reached from it and thus, there exists a path from q to a final state (condition 2) and since q has been reached from the initial state (condition 1), q is connected.

Applying this algorithm to the transducer of Figure 3.4 will produce the transducer shown in Figure 3.5.

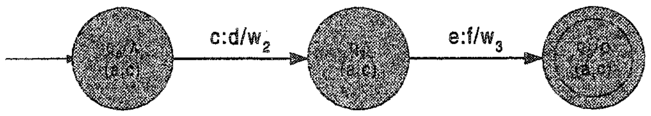


Figure 3.5: *Transducer without unconnected states.*

Unfortunately, this algorithm does not work in the case of cyclic transducers. The problem stems from the fact that in a cyclic transducer a state can be connected but the algorithm will label it as unconnected. Figure 3.6 shows an example of such a transducer in which a connected state will be erroneously removed. Indeed, the state q_2 will be labelled unconnected even if it is connected since the algorithm will not find the path (q_2, q_0, q_1) which leads to the final state.

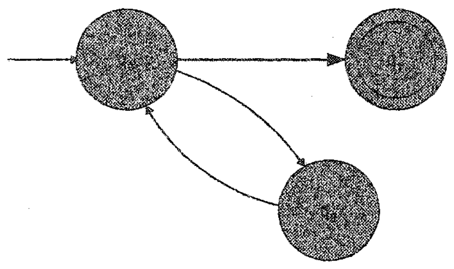


Figure 3.6: *Example of cyclic transducer for which Algoritm 5 fails.*

For resolving this problem, the concept of strongly connected component will now be introduced.

Definition 3.3.
A strongly connected component (SCC) in a transducer $T = (Q, i, F, \Sigma_i, \Sigma_o, E, \lambda, \rho)$ is a set of states $Q_{scc} \subseteq Q$ for which every state $q_v \in Q_{scc}$ can be reached from every state $q_u \in Q_{scc}$.

In Figure 3.6, the set $\{q_0, q_2\}$ is a strongly connected component since q_2 is reachable from q_0 and vice-versa. Strongly connected components of a transducer can be found using the Tarjan algorithm [33] which uses depth-first search algorithm. The following theorem uses the SCC concept to solve the connection problem.

Theorem 3.1.
If a state $q \in Q_{scc}$ is connected, then all states in Q_{scc} are also connected.

Proof. Let $q \in Q_{scc}$ be a connected state. By Definition 3.3, there exists a path from q to every state $q_v \in Q_{scc}$. Since q is connected, there exists a path from the initial state to q and thus, from the initial state to every $q_v \in Q_{scc}$ (condition 1).

The initial statement that q is a connected state implies a path from it to a final state. Since Definition 3.3 states that q can be reached from every state $q_v \in Q_{scc}$, there exists a path from $q_v \in Q_{scc}$ to a final state (condition 2). Therefore, since both conditions are fulfilled for every state $q_v \in Q_{scc}$, they are connected. □

Algorithm 6 Revisited Connection

```
Connection(T)
 1: for each $q \in Q$ do
 2:    orderNum[q] $\leftarrow$ 0
 3:    connected[q] $\leftarrow$ false
 4: visitCount $\leftarrow$ 0
 5: DFS-Visit($T, T.InitialState$)
 6: for each $q \in Q$ do
 7:    if connected[q] = false then
 8:       Remove q from Q
 9: return $T$

DFS-Visit(T,q)
 1: $visitCount \leftarrow visitCount + 1$
 2: $orderNum[q] \leftarrow visitCount$
 3: $oldestState \leftarrow visitCount$
 4: $STACK \leftarrow q$
 5:
 6: for each $(q, \sigma_i, \sigma_o, w, q') \in E[q]$ do
 7:    if orderNum[$q'$] = 0 then
 8:       old $\leftarrow$ DFS-Visit($T, q'$)
 9:       if old < oldestState then
10:          oldestState = old;
11:    else if orderNum[$q'$] < oldestState then
12:       oldestState $\leftarrow$ orderNum[$q'$]
13:    $connected[q] \leftarrow connected[q] \wedge connected[q']$
14:
15: if $q \in F$ then
16:    connected[q] $\leftarrow$ true
17:
18: if oldestState = orderNum[q] then
19:    repeat
20:       $s \leftarrow STACK$
21:       connected[s] $\leftarrow$ connected[q]
22:    until $s = q$
23: return oldestState
```

Algorithm 6 is the Tarjan's algorithm, described in [33], for strongly connected components with lines 13, 15, 16 and 21 added to find connected states. Note that in this algorithm, states do not have any color associated with them but rather numbers describing the order in which they have been discovered during the search process. This number is also referenced to be the ancientness of the state. The main goal of the algorithm is to find the root of strongly connected components.

Definition 3.4.
The root of a strongly connected component is the first state reached from the initial state in a depth-first search process.

The algorithm takes as input a transducer $T = (Q, i, F, \Sigma_i, \Sigma_o, E, \lambda, \rho)$. The Connection procedure works as follows. Lines 1-4 initialize all states as not connected and their ancientness number as not visited. Line 5 initiates the depth-first search process at the initial state. When the search process is completed, all unconnected states are removed from the transducer (lines 6-8).

Lines 1-4 of the DFS-Visit procedure assign the ordering number to the state which is put in the stack. When q is reached, it is guessed as the root of its component. Note that states in the stack are ordered according their ancientness. When an adjacent state is processed (at lines 7-8), new root candidates are obtained. If a candidate is older than the current root, it becomes the new root of the component containing q (lines 9-12). Line 13 propagates the connection attribute from adjacent states. When all adjacent states have been processed, the ancientness of q is equal to the ancientness of the root of its component if and only if q is a root. In this case, all states of components *i.e.* all states having a greater ordering number, are removed from the stack (lines 18-22) and form a strongly connected component. If the root state is connected, then all states of the component must be set to connected (line 21). The correctness of this operation is based on the following theorem:

Theorem 3.2.
The root state of a strongly connected component can be used to determine if the component is connected or not.

Proof. There are two cases to take into account. The first case occurs when the root is connected which should imply that the strongly connected component is also connected. This case is directly proved by Theorem 3.1.

The second case arises when the root state is unconnected. In this case, it must be proven that it is impossible to set as connected a state in the component and then setting the root as unconnected in Algorithm 6. The proof is by contradiction. Suppose that the root state q_r is not connected and a state $q_{scc} \in Q_{scc}$ is. Since state q_r is the first component state discovered, the path from it to q_{scc} is known, which implies that the connection attribute of q_{scc} will be propagated to q_r. Therefore q_r is also set as connected, contradicting the initial assumption. □

Running Time Analysis
The running time of the Connection procedure, without considering the call to the DFS-Visit procedure, depends on the number of states $|Q|$ in the input transducer since loops of lines 1-3 and 6-8 pass through all states. Thus, the running time of this procedure is $\mathcal{O}(|Q|)$.

The DFS-Visit procedure is called at most one time per state $q \in Q$ since the function is called only when its ancientness is 0 and a new ancientness, different from 0, is determined at the beginning of the procedure. In one execution pass of DFS-Visit, the loop of lines 6-13 has the same complexity as for a simple depth-first search and is thus $\mathcal{O}(|E|)$. The loop of lines 18-22 is executed at most $|Q|$ times since each state can be pushed on the stack only once. Operations on the stack can be done in $\mathcal{O}(1)$ time. Therefore, the running time of the connection algorithm is linear: $\mathcal{O}(|Q|+|E|)$.

3.4 Shortest-Paths Algorithms

Finding the shortest-path is a classic problem in graph theory and network programming and has been extensively studied over the years. The problem consists in finding in a given transducer, the successful path yielding the smallest cost. Recall that a successful path is a collection of consecutive transitions beginning at the initial state and ending at a final state. Typically, the cost of a path is the sum of transition weights making up this path.

Two categories of problems will be discussed in this section. The first one concerns the classic shortest-distance problem presented in the general case of semirings. The second problem consists in finding the shortest-path and its generalization which consists in finding the k shortest-paths in transducers.

3.4.1 Shortest-Distance Algorithms

The shortest-distance algorithm presented here is a generalization of the classical shortest-distance algorithms described in many computer science books. The classical algorithms cannot be used with non-idempotent semirings since they produce a wrong result. The generalization proposed by Mohri [18, 22] resolves this problem by allowing the use of non-idempotent semirings. Recall that a semiring $(\mathcal{K}, \oplus, \otimes, \bar{0}, \bar{1})$ is idempotent if and only if $x \oplus x = x, \forall x \in \mathcal{K}$.

Recall that in the general case of semirings, the cost $w[\pi]$ of a path $\pi = e_1, ..., e_n$ is the $\otimes$-product of transition weights making up this path:

$$w[\pi] = \bigotimes_{(q,\sigma_i,\sigma_o,w,q')\in\pi} w$$

and the shortest-distance from a state $s \in Q$ to a final state $q \in F$, denoted $d[s]$, is defined as

$$d[s] = \bigoplus_{\pi\in\Pi(q)} w[\pi]$$

where $\Pi(q) = \{\pi_1, ..., \pi_n\}$ is the set of paths from q to F. When the tropical semiring $\mathcal{S} = (\mathbb{R}_+ \cup \{\infty\}, min, +, \infty, 0)$ is used, this definition of the shortest path coincides with the classical definition presented in books since the $\otimes$-product of transitions weights becomes the usual addition of costs and the $\oplus$-addition calculates the minimum cost of all paths $s \rightsquigarrow q$.

The algorithm is also general in the sense that any queue discipline, such as priority queue or DFS, can be used. Selected combinations of a semiring and of a queue discipline produce algorithms equivalent to those presented in the classic literature. For example, using a priority queue in conjunction with the tropical semiring in the implementation of the generic algorithm coincides with the classical Dijktra's algorithm.

Algorithm 8 shows the pseudocode of the generic single-source shortest-distance algorithm presented by Mohri in [18] and [22]. The algorithm computes the shortest distance from the source state s to a final state in the input transducer $T = (Q, i, F, \Sigma_i, \Sigma_o, E, \lambda, \rho)$.

Algorithm 7 Generic single-source shortest path

```
SingleSourceShortestDistance(T,s)
 1: for each $q \in Q$ do
 2:     $d[q] \leftarrow \bar{0}$
 3:     $r[q] \leftarrow \bar{0}$
 4: $d[s] \leftarrow \bar{1}$
 5: $r[s] \leftarrow \bar{1}$
 6: $S \leftarrow \{s\}$
 7: while $S \neq \emptyset$ do
 8:     $q \leftarrow head(S)$
 9:     $Dequeue(S)$
10:     $r \leftarrow r[q]$
11:     $r[q] \leftarrow \bar{0}$
12:     for each $(q, \sigma_i, \sigma_o, w, q') \in E[q]$ do
13:         if $d[q'] \neq d[q'] \oplus (r \otimes w)$ then
14:             $d[q'] \leftarrow d[q'] \oplus (r \otimes w)$
15:             $r[q'] \leftarrow r[q'] \oplus (r \otimes w)$
16:             if $q' \notin S$ then
17:                 $S \leftarrow S \cup \{q'\}$
18: return $\bigoplus_{q \in F} d[q]$
```

The algorithm works as follows. For each state $q \in Q$, the algorithm uses two attributes: an estimate of the shortest path from s to q, maintained in $d[q] \in \mathcal{K}$; and $r[q] \in \mathcal{K}$, the total weight $\oplus$-added to $d[q]$ since the last time q has been extracted from the queue. Both attributes are initialized at lines 1-5.

At line 7, the queue is initialized with s, the state from where the search begins. This queue is used to maintain the set of states to be explored. At each pass through the loop of lines 7-17, a state q is extracted from the queue (lines 9-10). The r attribute of state q is stored and then reset to $\bar{0}$ (lines 10-11). The loop of lines 12-17 explores each transition of state q and updates the attributes of the destination state q' if it can be improved *i.e.* if the cost $d[q']$ is different than $d[q]$ $\otimes$-multiplied by the weight of the transition. In the specific case of the tropical semiring, that means that $d[q']$ will be updated if the new path, passing by q, has a smaller cost than the previous estimated shortest-path. In the literature, this step is often referred to as the relaxation of the transition $(q, \sigma_i, \sigma_o, w, q')$. If the transition has been relaxed and if the destination state q' is not in the queue, q' is inserted in.

Finally, line 18 returns the $\oplus$-added cost of all the final states in T. In the case of the tropical semiring, it is clear that each estimated path from s to q is, when the algorithm terminates, the shortest one since q is updated each time that a new shortest-distance is discovered.

Running Time Analysis

The running time of the algorithm depends on the semiring and the queue discipline considered. T_i will denote the worst cost for inserting a state q in S, T_e the worst cost for extracting q from S, N_q the number of times that q has been inserted in S, $T_\oplus$ the time of a $\oplus$-addition, $T_\otimes$ the time of a $\otimes$-product and T_a the time of an assignment.

The loop of lines 1-3 passes through all states and thus, runs in $\mathcal{O}(|Q|)$. The second loop depends of the number of times that states are inserted in the queue. Thus, its running time is

$$\mathcal{O}((T_i + T_e)\sum_{q \in Q} N_q).$$

The loop of lines 12-17 passes through all transitions and performs some semiring operations and assignments. For each state, the loop is executed $|E[q]| \cdot N_q$ times. Thus, the running time of this loop is

$$\mathcal{O}((T_\oplus + T_\otimes + T_a) \cdot \sum_{q \in Q}(|E[q]| \cdot N_q)).$$

Therefore, the total running time, in general, for this algorithm is

$$\mathcal{O}(|Q| + (T_\oplus + T_\otimes + T_a) \cdot \sum_{q \in Q}(|E[q]| \cdot N_q) + (T_i + T_e)\sum_{q \in Q} N_q).$$

The Generic Algorithm and Dijkstra's Algorithm

Used with the tropical semiring $\mathcal{S} = (\mathbb{R}_+ \cup \{\infty\}, min, +, \infty, 0)$ and a priority queue, the generic algorithm coincides with Dijkstra's algorithm, a classic in the graph literature [7, 30].

Operations of the tropical semiring are quite simple; both semiring operations are done in $\mathcal{O}(1)$ time. The cost of priority queue operations depends on its implementation. Using a Fibonacci heap, insertion takes O(1) time and the extraction of the smaller element takes $\mathcal{O}(\log n)$ time. Finally, the priority queue ensures, by the optimality principle [15], that each state will be inserted in the queue at most once. Therefore, the running time is

$$\mathcal{O}(|E| + |Q| \log |Q|).$$

Note that even though the Fibonacci heap is the most used implementation of priority queues in shortest paths problems, the complexity can be further improved using a RAM priority queue [34].

In practice, if the algorithm is used only in the case of the tropical semiring and with a priority queue, the execution speed of the algorithm will be improved by stopping the search when a final state is reached. This optimization works since the priority queue, used in conjunction with the tropical semiring, ensures that there is no path from s to q with a smaller cost [7]. Therefore, when a final state is reached, the path from s to $q \in F$ is the smaller one. This improvement does not change the complexity but, in practice, often leads to a faster search.

K-Shortest-Distances Problem

It can also be useful to obtain the k shortest-distances from a state q to a final state. Fortunately, this problem can be solved using the generic algorithm presented here, implemented with the k-tropical semiring.

Definition 3.5.
The k-tropical semiring is a semiring defined as $(\mathbf{K}, \oplus, \otimes, \bar{0}, \bar{1})$ where

- $\mathbf{K} = (\mathbb{R}_+ \cup \{\infty\})^k$,
- $\bar{0} = (\infty, ..., \infty)$,
- $\bar{1} = (0, ..., 0)$,
- $(x_1, ..., x_k) \oplus (y_1, ..., y_k) = \underset{1<i,j<k}{kmin}(x_i, y_i)$,
- $(x_1, ..., x_k) \otimes (y_1, ..., y_k) = \underset{1<i,j<k}{kmin}(x_i + y_i)$.

Using this semiring, the algorithm calculates, at each state, the k-shortest distances. Thus, at the end, the algorithm outputs the k-shortest distances from the source state to every state in the transducer.

The complexity of the algorithm with the k-tropical semiring and using a best-first search queue discipline such as a priority queue is thus:

$$\mathcal{O}(k \cdot |E| + k \cdot |Q| \log |Q|).$$

All-Pairs Shortest-Distance

The all-pairs shortest-distance problem consists in finding the weight of the shortest-path between all pairs of states in a transducer. A small modification of Algorithm 7 allows to use it to compute the all-pairs shortest-distance. This modification consists in returning d, the array containing the weights of shortest-paths between s and all other states, instead of only the shortest distance. Applying this new algorithm to every state in T will compute the all-pairs shortest-distance.

Since Algorithm 7 is executed $|Q|$ times, the running time of the all-pairs shortest-distance is

$$\mathcal{O}(|Q|^2 + |Q| \cdot (T_\oplus + T_\otimes + T_a) \cdot \sum_{q \in Q}(|E[q]| \cdot N_q) + |Q| \cdot (T_i + T_e) \sum_{q \in Q} N_q).$$

In the case where the algorithm is implemented in the tropical semiring and with a priority queue, the running time is

$$\mathcal{O}(|Q| \cdot |E| + |Q|^2 \log |Q|).$$

3.4.2 Shortest-Paths Algorithms

The shortest-path algorithm described here is the famous Disjktra's shortest-path algorithm described in most computer science books, including [7] and [30]. This algorithm has been originally designed to find the shortest-path in a weighted graph. However, transducers can be considered as directed graphs since the additional symbols carried by transitions in them are not taken into account in the shortest path computation. Hence, the algorithm can be directly applied to them.

Algorithm 8 shows the pseudocode of Dijkstra's algorithm for transducers. A heap is used to maintain the set of states to be explored. The algorithm has two inputs: a transducer $T = (Q, i, F, \Sigma_i, \Sigma_o, E, \lambda, \rho)$ and s, the state from which the search begins. Typically, this state is the initial state of T.

Algorithm 8 Dijkstra's single-source shortest path

```
SingleSourceShortestPath(T,s)
 1: for each $q \in Q$ do
 2:    $d[q] \leftarrow \infty$
 3:    $\pi[q] \leftarrow NIL$
 4: $d[s] \leftarrow 0$
 5: $Heap.Insert(s)$
 6: while $S \neq \emptyset$ do
 7:    $q \leftarrow Heap.Head()$
 8:    if $q \in F$ then
 9:       return $\pi[q]$
10:    for each $(q, \sigma_i, \sigma_o, w, q') \in E[q]$ do
11:       if $d[q'] > d[q] + w$ then
12:          $d[q'] \leftarrow d[q] + w$
13:          $\pi[q'] \leftarrow q$
14:          if $q' \notin S$ then
15:             $S \leftarrow S \cup \{q'\}$
16: return $NIL$
```

The algorithm works in the same way as when Algorithm 7 is implemented with the tropical semiring and priority queue. In this algorithm, $d[q]$ is an estimation of the shortest-distance from s to q and $\pi[q]$ denotes the predecessor state of q used to save the shortest-path associated with q.

The first loop of the algorithm (lines 1-3) initializes both $d[q]$ and $\pi[q]$. At line 5, the state from which the search begins is inserted in the heap. Line 7 extracts, from the heap, the state with the smallest $d[q]$ value. The loop of lines 10-16 performs the relaxation step. This means that for each transition $e = ((q, \sigma_i, \sigma_o, w, q') \in E[q]$, the path from s to q' is updated if the weight of the new estimate is smaller than the previous one (lines 11-12). Lines 14-15 insert q' in the heap if it is not already present.

Recall that the optimality principle, which is ensured by the priority queue and the relaxation step [7], states that when a state q is extracted from the heap, the weight of the path $s \rightsquigarrow q$ is the shortest of all existing paths between s and q. Hence, if a final state is reached, the shortest-path $s \rightsquigarrow F$ is found and the algorithm can be halted. This optimization is applied at lines 8-9. Finally, the algorithm returns NIL if no final state has been reached during the search (line 16).

Running Time Analysis

The first loop of the algorithm passes through all states, thus takes $\mathcal{O}(|Q|)$ time. In the loop of lines 6-15, a state q is extracted from the heap and each transition is explored. The extraction is done in $\mathcal{O}(\log |Q|)$ using a classical heap implementation [7]. The loop of lines 10-15 is executed $|E[q]|$ times and, in the worst case, the insertion of $\mathcal{O}(\log |Q|)$ is done each time. Thus, the inside of the loop of lines 6-15 takes $\mathcal{O}(|E[q]| \cdot \log |Q|)$ time.

By the optimality principle, this loop is executed at most $|Q|$ times. Therefore, the total running time of this algorithm is

$$\mathcal{O}((|Q| + |E|) \cdot \log |Q|)$$

The algorithm can be improved by using a Fibonacci heap. This implementation of heap has an amortized cost of $\mathcal{O}(1)$ for insertion of an element. Therefore, the running time becomes

$$\mathcal{O}(|E| + |Q| \cdot \log |Q|)$$

The algorithm can yet be improved to $\mathcal{O}(|E| + |Q| \cdot \log \log |Q|)$ by using a RAM priority queue [34].

K Shortest-Paths Problem

The problem of finding the shortest, the second, the third,..., the K^{th} shortest path, for $K \geq 1$, instead of finding only the shortest one is another well studied problem in computer science.

The algorithm presented here is an extension of Dijkstra's shortest-path algorithm described earlier. In the original algorithm, two attributes are maintained for each state q: the weight of the shortest path from the source state to q and its predecessor. The K-shortest paths problem can be resolved by maintaining, at each state, a totally ordered set of (p, w) pairs, where w is the weight of one of the k shortest-paths from s and its predecessor p. In this case, the core of the algorithm is still the same and thus, works in the same way.

Algorithm 9 shows the pseudocode of Dijkstra's algorithm extended to determine the k-shortest paths in a transducer. Note that it is assumed that the transducer contains only one final state. However, in the general case of $|F| > 1$, the algorithm is not affected since ϵ-transitions can be implicitly added in that case. The algorithm needs as its input the transducer $T = (Q, i, F, \Sigma_i, \Sigma_o, E, \lambda, \rho)$ in which the search will be done, the source state s and the number K of paths to find.

Algorithm 9 A generalization of Dijkstra's algorithm of the k-shortest paths

```
SingleSourceKShortestPath(T,s,K)
1: for each $q \in Q$ do
2:    $\pi[q] \leftarrow \emptyset$
3:    $c[q] \leftarrow 0$
4: $\pi[s] \leftarrow (NIL, 0)$
5: $Heap.Insert(s)$
6: while $S \neq \emptyset$ do
7:    $q \leftarrow Heap.Head()$
8:    $(p, \omega) \leftarrow \min \pi[q]$
9:    $c[q] \leftarrow c[q] + 1$
10:   if $q \in F$ and $c[q] = K$ then
11:      return $\pi[q]$
12:   if $c[q] \leq K$ then
13:      for each $(q, \sigma_i, \sigma_o, w, q') \in E[q]$ do
14:         $\pi[q'] \leftarrow \pi[q'] \cup \{(q, \omega + w)\}$
15:         if $q' \notin S$ then
16:            $S \leftarrow S \cup \{q'\}$
17:
18:   $\pi[q] \leftarrow \pi[q] - \min \pi[q]$
19:   if $|d[q]| \geq 1$ then
20:      $S \leftarrow S \cup \{q\}$
21: return $NIL$
```

In this algorithm, $\pi[q]$ denotes for the state q, the set of pairs (p, ω) of a predecessor state p which describes a path and its associated weight ω. The min operation over the set $\pi[q]$ returns the pair (p, ω) such that ω is the smallest. The algorithm also maintains the attribute $c[q]$ which contains the number of times that q has been extracted from the priority queue. These attributes are initialized at lines 1-5. The priority queue ordering is based on the smallest weight in $\pi[q]$. When a state q is extracted, its associated pair containing the smallest weight is removed from $\pi[q]$ and it is inserted in the queue according the new smallest weight (lines 18-20).

Each time through the loop 8-18, a state is extracted from the priority queue (line 7). For each transition $e \in E[q]$, a new pair (p, ω) is added to the set of the destination state q' (lines 14). Note that the size of $\pi[q]$ can be limited to K since q will be taken into account at most K times (line 12). At lines 17-18, q' is added to the priority queue if it is not already present.

Since no more than K paths can pass through any state q, the search is limited to K extractions from the priority queue (line 12) and the algorithm terminates when the final state has been extracted from the priority queue K times (lines 10-11).

Running Time Analysis

The loop at lines 1-3 passes through all states and thus runs in $\mathcal{O}(|Q|)$ time. The loop of lines 6-20 will be executed while the priority queue is not empty. However, the number of extractions per state is limited to K (by the optimality principle) and thus, the loop will be executed $\mathcal{O}(K \cdot |Q|)$ times. At each loop iteration a state is extracted from the priority queue, which takes $\mathcal{O}(\log n)$ time when a heap is used. The loop of lines 13-16 passes through all transitions exiting q and is therefore executed $\mathcal{O}(|E[q]|)$ times. In this loop, the insertion in the priority queue (line 16) takes $\mathcal{O}(\log |Q|)$ time. Thus, the running time of this loop is $\mathcal{O}(|E[q]| \cdot \log |Q|)$ and is executed $\mathcal{O}(K \cdot |Q|)$ times. Therefore, the total running of this algorithm is

$$\mathcal{O}(K \cdot (|Q| + |E|) \cdot \log |Q|)$$

Using a Fibonacci heap, the insertion is done in constant time. Therefore, the running time of the loop at lines 13-16 is $\mathcal{O}(|E[q]|)$. The total running time becomes

$$\mathcal{O}(K \cdot |E| + K \cdot |Q| \cdot \log |Q|)$$

3.5 Weight Pushing

It is known that a weighted transducer can be reweighted in a infinite number of ways. This means that an equivalent transducer can be obtained from an input one by modifying the weight distribution along the transitions without altering the described language. The weight pushing is a special case of reweighting which consists in pushing the weights toward the initial state.

The weight distribution can improve some algorithms such as the Viterbi search used in speech recognition. To be efficient in a large vocabulary context, the Viterbi algorithm employs pruning based on the combined weight from different probabilities involved in speech recognition. Thus, the weight distribution may have an impact on the execution speed of the speech recognition system. Weight pushing is also used in the minimization of weighted automata.

This section presents an algorithm performing weight pushing on any weighted transducer. This algorithm can be used with any weight semiring. In the case of speech recognition, the use of the log semiring considerably increases the recognizer's speed [22]. However, the tropical semiring is used in the weighted version of the minimization algorithm.

To see how weight pushing works, let us introduce a new function $V : Q \to \mathcal{K}$ called the potential function of states. This function maps a state $q \in Q$ to a weight $w \in \mathcal{K}$ where $\mathcal{K}$ is a weight semiring. In the case presented here, where the weights have to be pushed toward the initial state, the potential $V(q)$ is defined as the shortest distance from q to a final state $q_f \in F$. The potential function is used to update the initial weight, transition weights and the accepting costs as follows [22]:

$$\begin{aligned}\lambda &\leftarrow \lambda \otimes V(i)\\ w &\leftarrow V(q)^{-1} \otimes w \otimes V(q'), \forall (q, \sigma_i, \sigma_o, w, q') \in E\\ \rho(q) &\leftarrow V(q)^{-1} \otimes \rho(q), \forall q_f \in F\end{aligned}$$

where $V(q)^{-1}$ should be interpreted as $-V(q)$. Note that the potentials along any successful path, namely, paths from the initial state to a final state, are added and then substracted. Hence, the weights associated to input strings are not affected by the reweighting.

Algorithm 10 shows the pseudocode of a weight pushing algorithm based on the reweighting rules previously presented. It uses the shortest distance algorithm, as mentioned before, to compute the potential function of states. The input is a transducer $T = (Q, i, F, \Sigma_i, \Sigma_o, E, \lambda, \rho)$ in which the weights will be pushed toward the initial state according to the implemented semiring.

Algorithm 10 Weight-Pushing

```
PushWeight(T)
1: $T^R \leftarrow Reverse(T)$
2: $V \leftarrow shortestDistance(T^R, T^R.InitialState)$
3: $\lambda(i) \leftarrow \lambda(i) \otimes V(i)$
4: for each $q \in Q$ do
5:   if $q \in F$ then
6:     $\rho(q) \leftarrow V[q]^{-1} \otimes \rho(q)$
7:   for each $(q, \sigma_i, \sigma_o, w, q') \in E[q]$ do
8:     $w \leftarrow V[q]^{-1} \otimes w \otimes V[q']$
```

The first step of the algorithm consists in computing the shortest distance from every state $q \in Q$ to a final state $q_f \in F$. An efficient way to perform that consists in applying the shortest distance algorithm presented in Section 3.4 on the reverse of the transducer. The result is the shortest distance from every state to the final state under consideration in T. The loop of lines 3-7 applies the update rules presented before on each final state and on every transition of the input transducer.

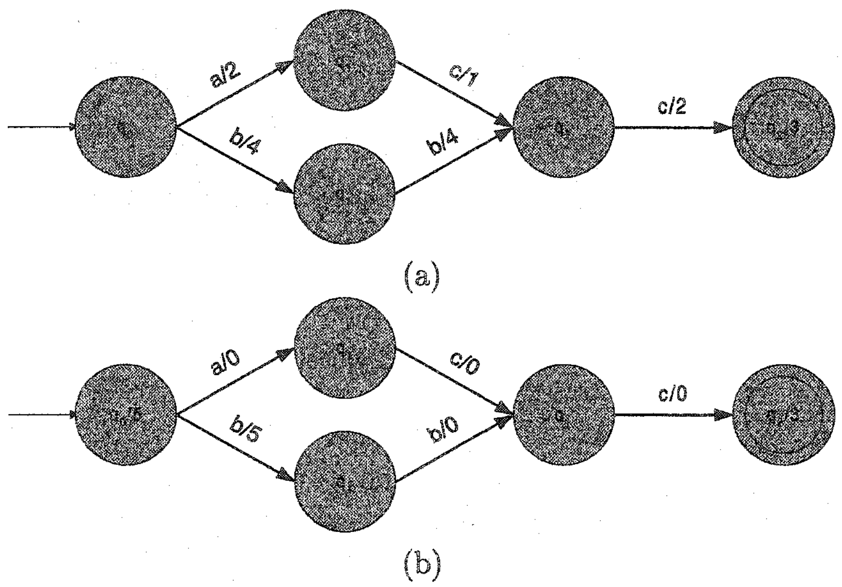


Figure 3.7: *Example of weight pushing*

Figure 3.7(a) shows a weighted transducer and Figure 3.7(b) shows the transducer obtained by pushing weights in the tropical semiring. This figure clearly shows that the weights have been moved along the paths toward the initial state. Note that the weights associated to the original input string have not been altered by the algorithm.

Running Time Analysis

The algorithm will be divided into three parts for the complexity analysis. The first part is the computation of the reverse of the input transducer. The corresponding algorithm will not be described in detail since it is straightforward. Basically, it consists in passing through all transitions of the transducer and reversing it by swapping their origin and destination states. This computation involves a loop passing through all transitions of all states; the running time is linear : $O(|Q| + |E|)$.

The second section concerns the shortest distance algorithm. This algorithm has been presented and analysed in section 3.4. The third section applies the reweighting rules on all transitions and all final states according to the potential function previously computed. This calculation of new weights implies a loop that passes through all states and all transitions. The calculated weights depend on the complexity of the semiring operations. Therefore, the running time of this loop is $O((|Q| + |E|) \cdot T_{\otimes}$.

It is clear that the computation of the shortest distance dominates the running time of the other sections of the algorithm. Therefore, the complexity of this algorithm is:

$$O(|Q| + (T_{\oplus} + T_{\otimes} + T_a) \cdot \sum_{q \in Q} (|E[q]| \cdot N_q) + (T_i + T_e) \sum_{q \in Q} N_q).$$

3.6 Summary

This chapter has presented some basic algorithms applicable to finite-state transducers. These algorithms have been described in detail and their complexity analyzed. The algorithms presented in this chapter are:

- The union operation which creates a new transducer representing the union of two languages.
- The concatenation operation which creates a new transducer representing the concatenation of two languages.
- Algorithms based on Depth-First Search techniques, including topological sort and connection algorithms.
- Shortest-Path algorithms allowing to find the path(s) of minimal cost in a transducer.
- The weight pushing algorithm which shifts weights carried by transitions toward the initial state.

These algorithms are fairly straightforward; presenting them has been a good introduction for the more complex algorithms presented in the next chapter.

Chapter 4

Advanced Algorithms

This chapter introduces algorithms for three important operations to work with finite-state transducers.

The first algorithm implements the ϵ-removal operation. This operation removes from a transducer all transitions for which the input and output symbols are ϵ. The resulting transducer describes the same language but does not contain any of these transitions; this increases the computational efficiency of the transducer.

The second algorithm concerns the determinization algorithm. This algorithm transforms a non-sequential transducer to its equivalent deterministic counterpart. Unfortunately, not all transducers admit a deterministic representation. This point is also discussed in this section.

Finally, the composition algorithm is described. As noted in Chapter 2, a transducer represents a binary relation between sequences of symbols. Thus, the composition of two transducers implements their relational composition.

4.1 Epsilon Removal

Transducers produced by several applications are often the result of various complex operations introducing ϵ-transitions in order to simplify them. Unfortunately, these transitions increase the computational load of transducers since they make them non-deterministic and in general, induce a delay in their use. Thus, the goal of this operation is to remove the ϵ-transitions of a given transducer.

Definition 4.1.
An ϵ-transition is a transition $e = (q, \sigma_i, \sigma_o, w, q')$ for which both σ_i and σ_o are the empty string ϵ, as typified in Figure 4.1.

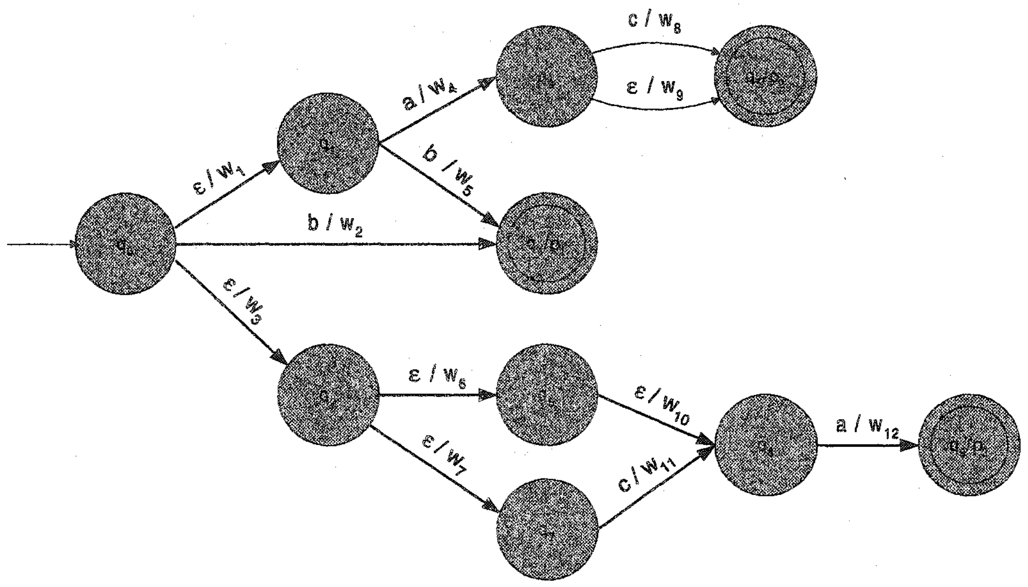


Figure 4.1: Example of a transducer with ϵ-transitions

The ϵ-removal algorithm runs in two steps. The first step consists in computing the ϵ-closure of the input transducer. The second step performs the ϵ-removal itself. The description of the ϵ-closure algorithm will first be described.

4.1.1 Epsilon-Closure

The first part of the ϵ-removal algorithm is the computation of the ϵ-closure of the input transducer. The ϵ-closure of a transducer T is another transducer T_c, containing only ϵ-transitions such that for all ϵ-paths $q \rightsquigarrow q'$ in T, there exists a transition $(q, \epsilon, \epsilon, w_\epsilon, q') \in E_c$.

Figure 4.2 shows the ϵ-closure of the transducer of Figure 4.1. In this figure, transitions already present in the input transducer are denoted by plain arrows and transitions added by the ϵ-closure are denoted by dashed arrows.

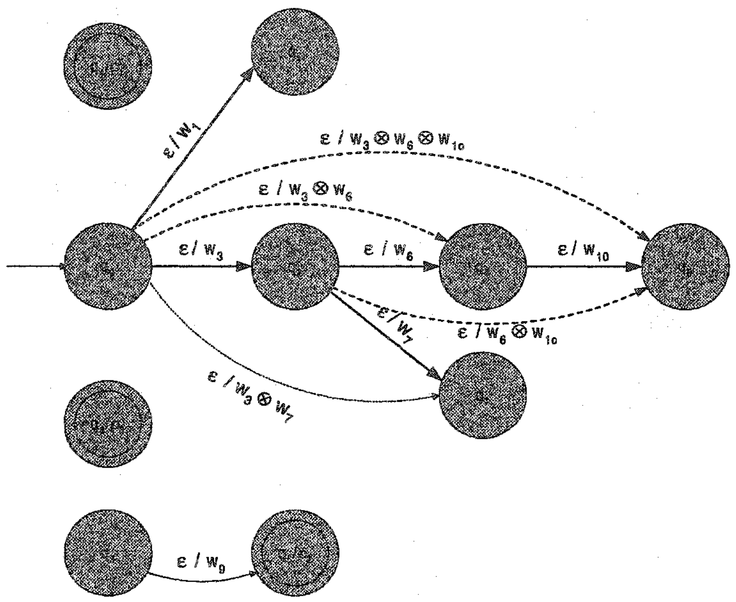


Figure 4.2: Epsilon-Closure of transducer shown by Figure 4.1.

Computing the ϵ-closure is equivalent to computing the all-pairs shortest-distances over the semiring $\mathcal{K}$ in T_ϵ [18] where, T_ϵ denotes the transducer T in which all non-ϵ-transitions have been removed . Thus, the algorithm involves the use of the shortest-distances algorithm described in Section 3.4. Algorithm 11 shows the pseudocode computing the ϵ-closure of an input transducer $T = (Q, i, F, \Sigma_i, \Sigma_o, E, \lambda, \rho)$.

Algorithm 11 Epsilon-Closure

```
Epsilon-Closure(T)
1: $T_\epsilon \leftarrow (Q, i, F, \Sigma_i, \Sigma_o, \{(q, \sigma_i, \sigma_o, w, q') \in E \mid \sigma_i = \epsilon \wedge \sigma_o = \epsilon\}, \lambda, \rho)$
2: $E_c \leftarrow \emptyset$
3: for each $q \in Q$ do
4:    $d \leftarrow SingleSourceShortestDistance(T_\epsilon, q)$
5:    for each $q' \in Q$ do
6:       if $d[q'] \neq \infty$ then
7:          $E_c \leftarrow E_c \cup \{(q, \epsilon, \epsilon, d[q'], q')\}$
8: return $(Q, i, F, \Sigma_i, \Sigma_o, E_c, \lambda, \rho)$
```

The algorithm works as follows. The transducer T_ϵ containing only ϵ-transitions of the input transducer is created at line 1. For each state $q \in Q$, the shortest-distance between q and all $q' \in Q$ is computed at line 4. If there is no path $q \rightsquigarrow q'$, the distance is set to infinity. Therefore, since d contains weights for all accessible states q' from q, creating transitions according to these values is equivalent to the ϵ-closure of state q. This operation is done in the loop of lines 5-7. The complete ϵ-closure for the transducer is accomplished by repeating this procedure for every state in Q (lines 3-7).

Running Time Analysis

It is already known that the running time for the shortest-distances algorithm is

$$\mathcal{O}(|Q| + (T_\oplus + T_\otimes + T_a) \cdot \sum_{q \in Q}(|E[q]| \cdot N_q) + (T_i + T_e) \sum_{q \in Q} N_q)$$

(see section 3.4 for more details). This algorithm is executed $|Q|$ times. The running time of the loop at lines 5-7 is also $\mathcal{O}(|Q|)$ since, in the worst case, the loop passes through all states in T. Therefore, the running time of this algorithm is:

$$\mathcal{O}(|Q^2| + |Q| \cdot (T_\oplus + T_\otimes + T_a) \cdot \sum_{q \in Q}(|E[q]| \cdot N_q) + |Q| \cdot (T_i + T_e) \cdot \sum_{q \in Q} N_q)$$

Using the tropical semiring and the Fibonacci heap in the shortest-distances algorithm, the running time is:

$$\mathcal{O}(|Q| \cdot |E| + |Q|^2 \cdot \log |Q|)$$

4.1.2 Epsilon-Closure in Acyclic Case

In the case where the transducer T does not contain any ϵ-cycle, *i.e.* T_ϵ is acyclic, the running time of the ϵ-closure algorithm can be significantly improved by visiting states in reverse topological order.

This improvement is obtained by using the property of topologically ordered transducers, which states that for any transition $(q, \sigma_i, \sigma_o, w, q')$, q appears before q'. Since the transducer is acyclic, the ϵ-closure of a state q depends only on ϵ-closure of its adjacent states. The ϵ-closure of adjacent states is already computed since states are visited in reverse order. Therefore, it is possible to compute the ϵ-closure by visiting each state once, thus in linear time.

Algorithm 12 shows the pseudocode of a procedure computing the ϵ-closure of a transducer $T = (Q, i, F, \Sigma_i, \Sigma_o, E, \lambda, \rho)$ in which there is no ϵ-cycle.

Algorithm 12 Epsilon-Closure-Acyclic

```
Epsilon-Closure-Acyclic(T)
 1: $T_\epsilon \leftarrow (Q, i, F, \Sigma_i, \Sigma_o, \{(q, \sigma_i, \sigma_o, w, q') \in E \mid \sigma_i = \epsilon \wedge \sigma_o = \epsilon\}, \lambda, \rho)$
 2: $S \leftarrow TopologicalSort(T_\epsilon)$
 3: $E_c \leftarrow \{(q, \sigma_i, \sigma_o, w, q') \in E \mid \sigma_i = \epsilon \wedge \sigma_o = \epsilon\}$
 4: while $S \neq \emptyset$ do
 5:    $q \leftarrow tail(S)$
 6:    $S \leftarrow S - \{q\}$
 7:    for each $(q, \epsilon, \epsilon, w_1, q') \in E[q]$ do
 8:       for each $(q', \epsilon, \epsilon, w_2, q'') \in E[q']$ do
 9:          $E_c \leftarrow E_c \cup \{(q, \epsilon, \epsilon, w_1 \otimes w_2, q'')\}$
10: return $(Q, i, F, \Sigma_i, \Sigma_o, E_c, \lambda, \rho)$
```

The first step consists in stripping the transducer T of its non-ϵ-transitions to produce T_ϵ (line 1). Line 2 initializes the list S by filling it with the topologically ordered set of states Q while line 3 initializes the set of transitions E_c with the ϵ-transitions of T. As noted before, the ϵ-closure computation of q depends only on adjacent states for which the ϵ-closure has already been computed. Thus, the ϵ-closure of q is computed by creating a transition t from q to every state reachable by its adjacent states (lines 7-9). Since this operation always implies two transitions, the weight carried by t is the $\otimes$-product of the two transition weights involved (line 9). The loop of lines 4-9 repeats the procedure for every state $q \in Q$ in reverse topological order.

Running Time Analysis

It is already known that the topological sort runs in linear time, *i.e.* $\mathcal{O}(|Q| + |E|)$ (see section 3.3 for more details). The main loop passes through all states $q \in Q$. For every state, transitions of q and those of its adjacent states are considered. Thus, the running time of this loop, in the worst case, is $\mathcal{O}(|Q| + 2 \cdot |E|) = \mathcal{O}(|Q| + |E|)$. Therefore, the running time of the algorithm is linear : $\mathcal{O}(|Q| + |E|)$.

4.1.3 Epsilon-Removal Algorithm

This section will describe the second part of the ϵ-removal algorithm which consists in creating new non-ϵ-transitions from pairs made of a non-ϵ and an ϵ-transition. The resulting transducer will contain only non-ϵ-transitions and will be equivalent to the original one.

Algorithm 13 shows the pseudocode of the ϵ-removal algorithm for weighted transducers. Its input is a transducer $T = (Q, i, F, \Sigma_i, \Sigma_o, E, \lambda, \rho)$ and its output is T without ϵ-transitions. Figure 4.3 shows the resulting transducer when the algorithm is applied to the transducer of Figure 4.1.

Algorithm 13 Epsilon Removal

```
RemoveEpsilon(T)
 1: $T_c \leftarrow \epsilon$-closure(T)
 2: $E \leftarrow \{(q, \sigma_i, \sigma_o, w_o, q') \in E \mid \sigma_i \neq \epsilon \vee \sigma_o \neq \epsilon\}$
 3: for each $q \in Q$ do
 4:   for each $(q, \epsilon, \epsilon, w_\epsilon, q') \in E_c[q]$ do
 5:     if $q' \in F$ then
 6:       $\rho_o[q] \leftarrow \rho_o[q] \oplus \rho[q'] \otimes w_\epsilon$
 7:     for each $(q', \sigma_i, \sigma_o, w, q'') \in E[q']$ do
 8:       if $\exists(q, \sigma_i, \sigma_o, w_o, q'') \in E$ then
 9:         $w_o \leftarrow w_o \oplus w_\epsilon \otimes w$
10:       else
11:         $E \leftarrow E \cup \{(q, \sigma_i, \sigma_o, w_\epsilon \otimes w, q'')\}$
12: return $T_o$
```

This algorithm works in two steps. The first step consists in computing the ϵ-closure of the input transducer (line 1), as previously described. The second step, which consists in removing ϵ-transitions, works as follows. The algorithm considers pairs of transitions (lines 3-11). A pair is made up of two transitions (t_1, t_2) where t_1 is an ϵ-transition $(q, \epsilon, \epsilon, w_\epsilon, q') \in E_c[q]$ and t_2 is a non-ϵ-transition $(q', \sigma_i, \sigma_o, w, q'') \in E[q']$. From every pair, a new transition $(q, \sigma_i, \sigma_o, w \oplus w_\epsilon, q'')$ is created. If the transition already exists in T, the weights are combined with the $\otimes$-product, otherwise the transition is inserted in T (lines 8-11). Lines 5-6 ensure that the final states are correctly handled. In the case where an ϵ-transition leads to a final state $q' \in F$, the originating state becomes also a final state for which the acceptation cost is the $\otimes$-product of the ϵ-transition weight and the acceptation cost of q'. The $\oplus$-addition is used to take into account the case where the originating state was already a final state.

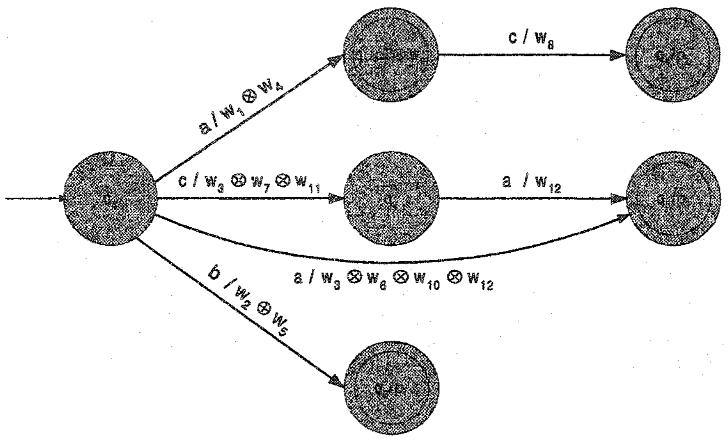


Figure 4.3: Transducer of figure 4.1 for which ϵ-transitions have been removed.

Running Time Analysis

Let q be the state considered and q' be a state belonging to the ϵ-closure of q. The loop at lines 7-11 considers all transitions in $E[q']$ and therefore is executed $\mathcal{O}(\mid E[q] \mid)$ times. In the worst case, every state q' belongs to the ϵ-closure of q. Hence, the running time of the nested loops of lines 4-11 is $\mathcal{O}(|Q| + |E|)$. These nested loops are executed $|Q|$ times (lines 3-11). Therefore, the running time of the second part of this algorithm (lines 2-11) is

$$\mathcal{O}(|Q|^2 + |Q| \cdot |E|).$$

In the case where the algorithm is applied to an ϵ-cyclic transducer, the total running time of the algorithm is dominated by the ϵ-closure, hence is

$$\mathcal{O}(|Q| \cdot |E| + |Q|^2 \log |Q|)$$

when the ϵ-closure is computed over the tropical semiring with a Fibonacci heap. In the case of an ϵ-acyclic transducer, the running time of the algorithm is dominated by the ϵ-removal section. Therefore, the running time is:

$$\mathcal{O}(|Q|^2 + |Q| \cdot |E|)$$

4.1.4 Improvements

In practice, the algorithm can be improved by using a heuristic to reduce, in many cases, the number of ϵ-transitions considered by the ϵ-removal algorithm. Consider the transducer shown in Figure 4.4. In this figure, dashed transitions represent ϵ-transitions created by the ϵ-closure algorithm and plain transitions are the original ones.

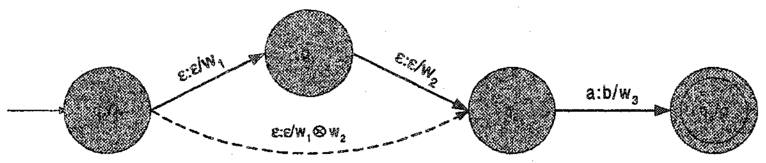


Figure 4.4: *epsilon*-closure with useless transitions

Let $e_{ij} \in E$ be a transition from state q_i to state q_j. At state 0, the algorithm will create a new transition e_{03} using the transitions-pair (e_{02}, e_{23}). Then, the algorithm will explore the state q_1 using the ϵ-transition e_{01}. However, this exploration is useless since there is no non-ϵ-transition going out from q_1. In general, ϵ-transitions going to a state without non-ϵ-transitions can be ignored by the algorithm since they are useless.

Now, consider the case of state q_1. Note that this state does not have ingoing non-ϵ-transitions. Therefore, this state will be unconnected since it cannot be reached in the resulting transducer. However, a new transition e_{13} will be created from the transitions-pair (e_{12}, e_{23}). This leads to useless computation since the state will be unconnected and thus, will be eliminated. In general, a state without ingoing non-ϵ-transitions can also be ignored by the algorithm.

This heuristic does not change the complexity of the algorithm since in the worst case, all ϵ-transitions are useful to obtain the good result. However, in practice, the implementation of this heuristic can lead to a 20% improvement in the speed of the algorithm.

4.1.5 Remarks

At the end of the process, some states may become inaccessible as previously mentioned . These states can be removed in linear time using the connection algorithm presented in section 3.3.

4.2 Determinization

This section describes a determinization algorithm. This algorithm can be used to obtain a deterministic automaton from a non-deterministic one or a sequential transducer from a non-sequential one.

An automaton is deterministic if and only if for any input string w, the sequence of states is unique. A transducer $T = (Q, i, F, \Sigma_i, \Sigma_o, \delta, \sigma, \lambda, \rho)$, where δ and σ are respectively the transition function and the output function such as defined in chapter 2, is said to be sequential if it is deterministic from its input point of view. More formally, T is sequential if and only if

$$|\delta(q, a)| \leq 1, \forall q \in Q, \forall a \in \Sigma_i$$

where $|\delta(q, a)|$ is the number of transitions leaving the state $q \in Q$ with the input label $a \in \Sigma_i$. Figure 4.5 shows an example of sequential transducer.

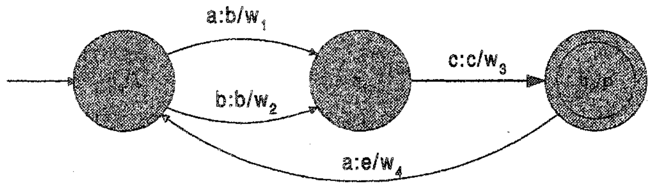


Figure 4.5: *Example of a sequential weighted transducer.*

Since in such transducers there is at most one transition labelled with any symbol of the input alphabet, sequential transducers are computationally very interesting. Indeed, using this kind of transducers to perform a transduction implies that each input string follows a unique path. Hence, the computation of the transduction depends only on the length of the input string and not on the number of states and transitions in the transducer.

The definition of sequential transducers can be generalized by introducing the possibility of generating an output string in final states. This final string works in the same way as the accepting cost previously defined. Hence, the final output string is $\oplus$-added to the usual output string of the transducer. Usually, the $\oplus$-addition refers to the concatenation of strings. This kind of transducer is called a subsequential transducers. An example of such a transducer is shown in Figure 4.6.

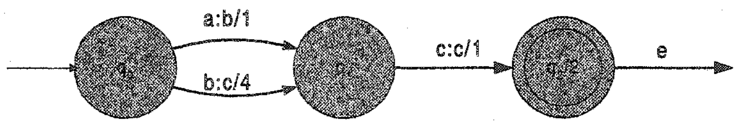


Figure 4.6: *Example of a subsequential transducer.*

A subsequential transducer with more than one final output string in a same final state is called p-subsequential, where p refers to the maximum number of final output strings in any final state. Hence, a p-subsequential transducer allows several output strings for a given input string.

Note that a p-sequential transducer can be easily converted into a transducer without output strings. Indeed, a final output string s can be represented by a sequence of $|s|$ consecutive transitions $(q_f, \epsilon, \sigma_1, \rho(q_1), q_2), (q_2, \epsilon, \sigma_2, 0, q_3), \cdots, (q_{|s|-1}, \epsilon, \sigma_{|S|}, 0, q_{|s|})$ where $q_{|s|}$ is the new final state and q_f is the old final state transformed in a non-final one. This new transducer is sequential up to the new transitions added in the conversion process. Figure 4.7 shows the subsequential transducer of Figure 4.6 converted into a transducer without final string output.

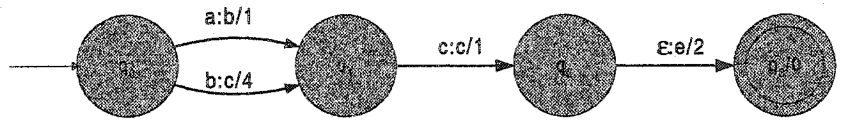


Figure 4.7: *Transducer without final output string equivalent to the one shown in Figure 4.6.*

Note that only one final state has to be added for all converted final output strings since all sequences of transitions can reach the same final state.

4.2.1 Determinization Algorithm

It is well known that any language described by a non-deterministic automaton can also be described by a deterministic one. Hence, any automaton admits an equivalent deterministic automaton. The procedure used to construct the deterministic automaton equivalent to a non-deterministic one is based on the subset construction method. This method constructs the set of states of the deterministic automaton with the power set of states of the input automaton. Then, transitions leaving these states are computed. Consider the automaton $A = (Q, i, F, \Sigma, \delta)$ shown in Figure 4.8.

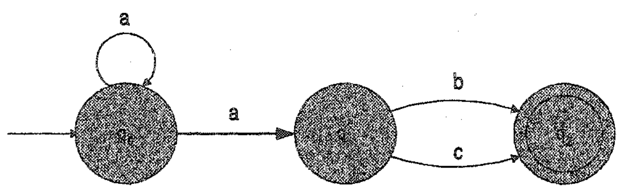


Figure 4.8: *Non-deterministic automaton A*

The first step in constructing the deterministic automaton $A_D = (Q', i', F', \Sigma, \delta')$ equivalent to A is the construction of the set of states Q', which is the power set of Q, denoted $P(Q)$.

$$Q' = P(Q) = \{\emptyset, \{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_1, q_2, q_3\}\}$$

Then, the initial state and the set of final states are defined. The initial state is the set of all states reachable by ϵ-transitions from the initial state of the non-deterministic automaton. In the example, the new initial state is $i' = \{q_0\}$. The set of final states is defined as the set of all subsets containing at least one final state in A, thus $F = \{\{q_2\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_1, q_2, q_3\}\}$ in the example.

Finally, the transition function δ' (or the set of transitions) is computed by calculating for each set $S \subseteq Q$ and for each symbol $a \in \Sigma$ the transition function:

$$\delta'(S, a) = \bigcup_{q \in S} \delta(q, a).$$

The resulting automaton, without unconnected states, is shown in Figure 4.9. This automaton describes the same language than A, *i.e.* the set of strings beginning by a finite number of as and ending by b or c.

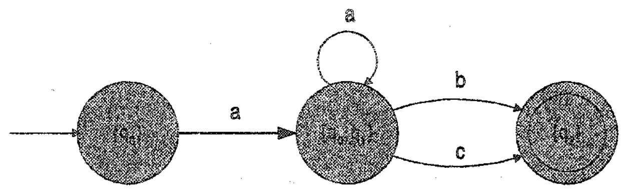


Figure 4.9: *Deterministic automaton equivalent to A*

This construction is used to prove that any automaton admits an equivalent deterministic one. A formal proof based on it is given in [24].

In contrast to unweighted automata, not all transducers (including acceptors) admit an equivalent sequential transducer. Indeed, a sequential transduction does not allow unbounded delay [17]. For example, consider the transducer function $f(w)$ which outputs $a^{|w|}$ when $|w|$ is even and $b^{|w|}$ otherwise. It is impossible to begin to write the output string associated to the input one since its length is known only after the input string has been processed.

Algorithm 14 shows the pseudocode of the determinization algorithm presented by Mohri in [17]. This algorithm is a generalization of the power set construction described before. In the classic algorithm, states are defined as a subset of states of the input automaton. In the case of transducers (and acceptors), the subsets contain pairs (q, x) where q is a state of the original transducer and x is the residual output associated with q.

The algorithm is presented in the general case of semirings, applicable to many types of transducers. For simplicity, the algorithm will be described in the case where the output symbol carried by transitions is a single element such as string-to-weight transducers. Thus, the algorithm takes as its input a transducer $T = (Q, i, F, \Sigma, \delta, \sigma, \lambda, \rho)$ and produces its sequential equivalent.

Algorithm 14 Determinization

```
Determinization(T)
 1: $F_o \leftarrow \emptyset$
 2: $\lambda_o \leftarrow \lambda$
 3: $i_o \leftarrow (i, 0)$
 4: $Queue \leftarrow \{i_o\}$
 5: while $Queue \neq \emptyset$ do
 6:   $q_o \leftarrow head[Queue]$
 7:   if $\exists(q, x) \in q_o$ such that $q \in F$ then
 8:     $F_o \leftarrow F_o \cup \{q_o\}$
 9:     $\rho(q_o) \leftarrow \bigoplus_{q \in F, (q,x) \in q_o} x \otimes \rho(q)$
10:   for each $a$ such that $\Gamma(q_o, a) \neq \emptyset$ do
11:     $\sigma_o(q_o, a) \leftarrow \bigoplus_{(q,x) \in \Gamma(q_o,a)} [x \otimes \bigoplus_{t=(q,a,\sigma,q')} \sigma]$
12:     $\delta_o(q_o, a) \leftarrow \bigcup_{q' \in \nu(q_o,a)} \{(q', \bigoplus_{(q,x,t) \in \gamma(q_o,a), n(t)=q'} [\sigma_o(q_o, a)]^{-1} \otimes x \otimes \sigma(t))\}$
13:     if $\delta_o(q_o, a)$ is a new state then
14:       $Queue \leftarrow Queue \cup \{\delta_o(q_o, a)\}$
```

The notation used in this algorithm as described in [17], will now be presented. Given a transition $t = (q, a, \sigma, q')$, $\sigma(t)$ denotes the output label/weight carried by t and n[t] denotes the destination state q' of the transition. Such as previously noted, a state q_o is made of a subset of pairs (q,x). The set of pairs $(q, x) \in q_o$ having transitions carrying the input label a is denoted by $\Gamma(q_o, a)$. The set of triples (q, x, t) where (q, x) is a pair in q_o for which q admits a transition t with the input label a is denoted by $\gamma(q_2, a)$. And finally, the set of states q' that can be reached by transitions carrying the input label a leaving states q_o states subset is denoted by $\nu(q_o, a)$. More formally, these sets are defined as

$$\begin{aligned}
\Gamma(q_o, a) &= \{(q, x) \in q_o \mid \exists t = (q, a, \sigma, q') \in E\}, \\
\gamma(q_o, a) &= \{(q, x, t) \in q_o \times E \mid t = (q, a, \sigma, q') \in E\}, \\
\nu(q_o, a) &= \{q' \mid \exists (q, x) \in q_o, \exists t = (q, a, \sigma, q') \in E\}.
\end{aligned}$$

The algorithm constructs the sequential transducer $T_o = (Q_o, i_o, F_o, \Sigma_o, \delta_o, \sigma_o, \lambda_o, \rho_o)$ as follows. The initial weight λ_o is the initial weight of T and the initial state i_o is a subset of one pair {(i,0)} (lines 1-2). A queue is used to maintain the set of subsets q_o waiting to be examined. This queue is initialized with the initial subset at line 3.

Recall that states of the resulting transducer are the subsets q_o. A subset q_o is a final state in the resulting transducer if q_o contains at least one pair (q, x) such that q is a final state in T, *i.e.* $q \in F$. The accepting weight of q_o is the $\oplus$-addition of all accepting weights of all final states in q_o (lines 8-9).

Then, for each symbol $a \in \Sigma$ such that there exists at least one state q of the subset q_o from which an outgoing transition carries the input symbol a, a new transition t_o leaving q_o and carrying the input symbol a is created (lines 10-14). The output symbol carried by the transition is computed as follows. For each transition $t = (q, a, \sigma, q')$, the $\otimes$-product of σ and the residual output associated with q is calculated. These results are $\oplus$-added to form the output symbol carried by the new transition.

The destination state of t_o is a subset made of pairs (q', x') where q' is a state of q_o that can be reached by transitions carrying the input label a and x' is the residual symbol associated with q'. The value of the residual symbol is the $\oplus$-addition of all output symbols carried by transitions reaching q' from states in q_o and carrying the input symbol a when they are combined by the $\otimes$-product. This operation is made at line 12. Finally, the newly created subset is inserted in the queue if it is a new state in Q_o (lines 13-14).

Figure 4.10a shows an example of non-sequential string-to-weight transducer admitting an equivalent sequential one. This transducer is defined over the tropical semiring thus, the $\oplus$-addition and the $\otimes$-product are respectively replaced in the pseudocode of Algorithm 14 by the *min* operation and by the usual addition of real numbers.

The resulting transducer is shown in Figure 4.10b. Note that the algorithm has produced a transducer accepting the same input strings accepted by the original one. However, only the smallest of weights associated with a given input string is produced by the sequential transducer. This is because the algorithm has removed the redundancies by combining the weights associated to the same input string. How weights are combined depends on over which semiring the transducer is defined. In the case of the tropical semiring, only the smallest one is considered.

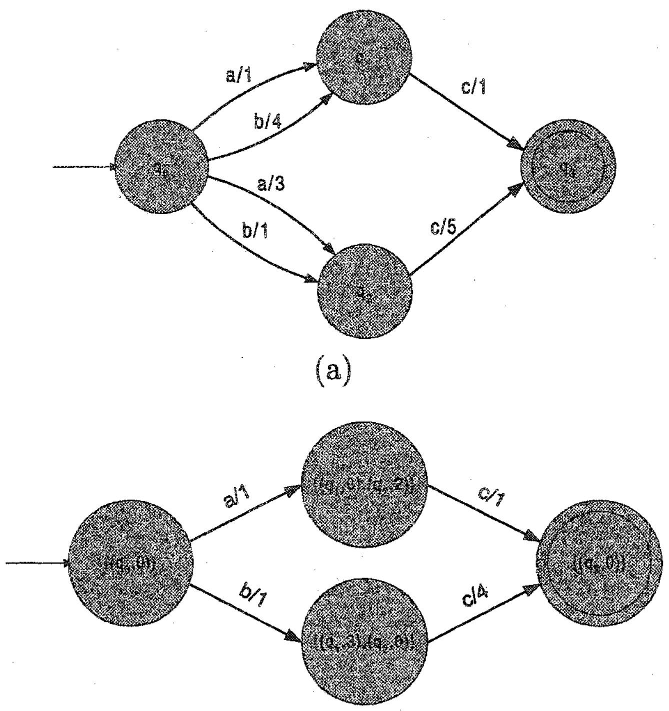


Figure 4.10: *A non-sequential transducer (a) and its sequential equivalent (b)*

Running Time Analysis

The main loop of this algorithm (lines 5-14) is executed once for each state q_o of the output transducer. Recall that q_o is a subset of states of the original transducer. In the worst case, the output transducer will contain all possible subsets of Q, namely the power set of Q. Therefore, the loop will be executed, in the worst case, $2^{|Q|}$ times. Hence, the running time of this algorithm is exponential to the number of states in the original transducer, therefore $\mathcal{O}(2^{|Q|})$.

4.2.2 Determinization of String-to-String Transducers

As noted before, Algorithm 14 has been presented in the general case of semirings; therefore it can be applied on transducers mapping strings to another type of output symbols. In particular, it can be used to determinize weighted string-to-string transducers defined over the cross-product of a weight semiring and the string semiring. Recall that the cross-product of two semirings is also a semiring. Figure 4.11a shows a non-sequential weighted string-to-string transducer defined over the cross-product of the tropical semiring and the string semiring.

Since such transducers output pairs of string and weight, subsets are made up of triplets (q, w, x) where $q \in Q$ is a state in the original transducer, $w \in \Sigma_o^*$ is the residual string and $x \in \mathcal{K}$ is the residual weight. This situation is illustrated in Figure 4.11b.

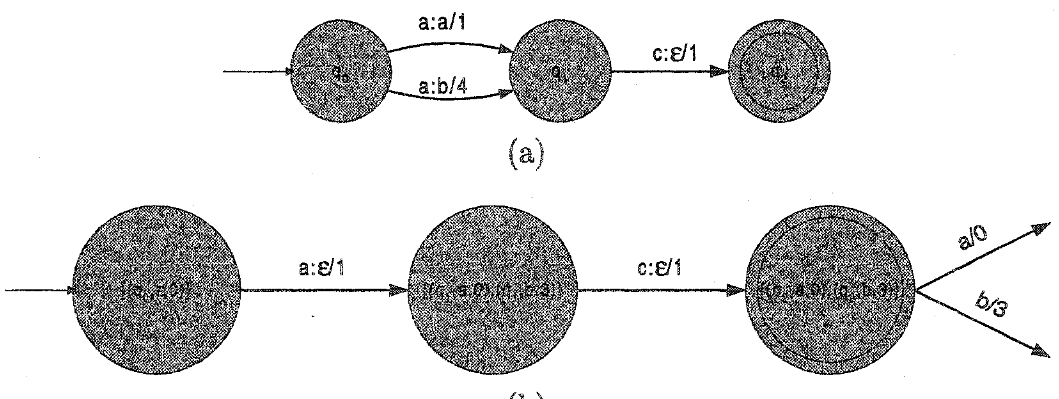


Figure 4.11: *A non-sequential transducer (a) and its 2-subsequential equivalent (b).*

4.2.3 Notes on Implementation

To implement this algorithm efficiently, there are two critical points to take into account.

The first one is the loop beginning at line 10. This loop considers every input label a such that there exists at least one state q in the subset q_o from which there is an outgoing transition labelled with a. This implies that the program has to search in transition sets of all states in q_2 to find both the next label to consider and all transitions labelled with it. This can be done by merging all transitions in the same set and then sorting it with respect to the input label. Thus, the problem is reduced to passing through the transitions composing this unique set. However, sorting an array takes $\mathcal{O}(n \log n)$ time, where n is the number of elements. In the worst case, a subset is made of all states of the original transducer; sorting this set takes $\mathcal{O}(|E| \log |E|)$ time. This procedure has to be repeated for all states of the output transducer, *i.e.* $\mathcal{O}(2^{|Q|})$ in the worst case.

A more efficient way consists in sorting all transition sets with respect to the input label, before performing the determinization algorithm. Since all transition sets are sorted, they can be merged in $\mathcal{O}(n)$ time, where n is the number of transitions in all states of the subset. Moreover, it is not necessary to perform the merge explicitly since passing through transitions in the same way as does the merging procedure leads to an efficient way of implementing the search of transitions carrying the input symbol a.

The second critical point that must be taken into account occurs at lines 13-14, in which the new subset is inserted in the queue if it has not already been created. To ascertain that, all subsets created so far have to be maintained in a list. A naive way to implement this is to use a data structure such as a linked list. However, to confirm that the subset does not exist, the new subset has to be compared to all other subsets in the list. A more efficient structure for this problem is a hashtable. Indeed, the hashing function will spread out the subsets over the buckets of the hashtable, which ensures that a manageable number of subsets will be compared to the new candidate. An efficient hashing function will take into account all triples in the subset and all elements of these triples.

4.2.4 Lazy Implementation

This algorithm allows a lazy implementation, also called an on-demand implementation. In the context of transducers, lazy implementation means that transitions of a state in the resulting transducer are computed only when required. A lazy implementation of the determinization algorithm is possible since the creation of transitions depends only on the subset from which transitions leaves. Indeed, transitions are created considering only with respect to transitions of states q belonging to the subset. Lazy implementation is very advantageous when a large transducer is constructed but only a small part of it has to be considered [27].

For example, consider the k-shortest paths algorithm presented in section 3.4. This algorithm outputs k-paths having the shortest distance from the initial state to a final one. However, it is possible that some input strings associated with these paths are the same. In many applications such as speech recognition, it is interesting to obtain the k-unique-shortest paths, namely the k-paths having the shortest distance and describing a unique input string. This version of the k-shortest paths can be implemented using the lazy implementation of the determinization algorithm. Indeed, since a sequential transducer does not have, by definition, two transitions sharing the same input label at the same state, applying the k-shortest path algorithms on a determinized transducer will produce a set of k shortest-paths having a one-to-one correspondence to the set of distinct input strings.

As noted before, not all transducers can be determinized and in that case, the determinization algorithm does not terminate and thus cannot be used as a pre-processing step. However, the shortest-paths algorithm explores only a finite part of the determinized transducer and thus the lazy implementation can be used to expand those states that are needed to compute the k unique shortest-paths. More details about this approach can be found in [23].

Another advantage of the lazy implementation is it can require less memory. Indeed, transitions are computed only when they are needed by the operation that requires them. Hence, when the operation does not use them any more, they can be deleted and re-computed if necessary. Since transitions are a big part of the memory space used by the transducer, the economy of memory can be substantial.

4.3 Composition

A transducer represents a binary relation between sequences of symbols (Chapter 2) thus, the composition of two transducers computes their relational composition. Let $T_1 : \Sigma_i^* \longrightarrow \Delta^*$ and $T_2 : \Delta^* \longrightarrow \Sigma_o^*$ be two relations represented by transducers in cascade as shown in Figure 4.12.

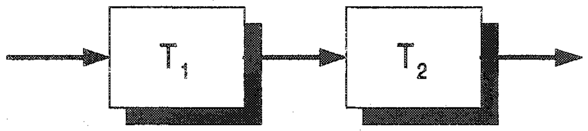


Figure 4.12: *A cascade of two transducers*

This cascade can be interpreted as follows. The transducer T_1 maps Σ_i^* to Δ^*. Thus, the set Δ^* becomes the input of transducer T_2 which itself maps Δ^* to Σ_o^*. Hence, the general behaviour of the cascade is a new binary relation: $T_1 \bullet T_2 : \Sigma_i^* \longrightarrow \Sigma_o^*$.

In general, given a transducer T_1 in which there is a path mapping sequence x to sequence y and a transducer T_2 in which there is a path mapping sequence y to sequence z, the composition $A \bullet B$ has a path mapping x to z. The weight of this path is the $\otimes$-product of the weights of the corresponding paths in T_1 and T_2 [19].

The composition is a key operation in transducer-based applications. It is used to construct complex transducers representing complex functions. For example, in the case of speech recognition, the composition is used to construct the recognition network needed by the recognition system. This network is constructed by the composition of different levels of representation with which transducers are associated.

4.3.1 Composition Algorithm

The composition algorithm is a generalization of the classical construction of pairs of states computing the automata intersection [24]. Recall that the intersection of two languages L_1 and L_2 is defined as:

$$L_1 \cap L_2 = \{w \mid w \in L_1 \land w \in L_2\}$$

Thus, the intersection of two automata $A_1 \cap A_2$ is a new automaton accepting any string accepted by both the original automata.

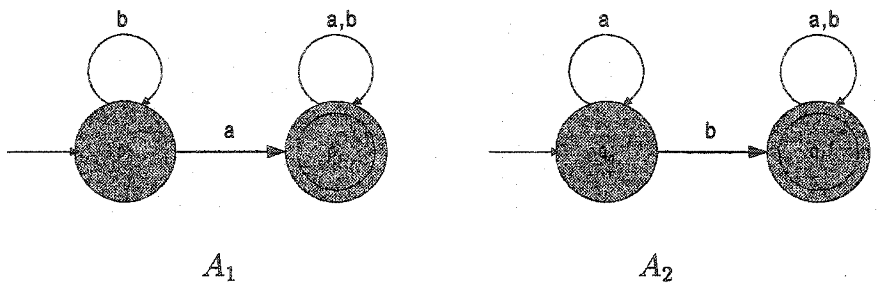


Figure 4.13: *Automata A_1 and A_2*

Consider two automata $A_1 = (Q_1, i_1, F_1, \Sigma, \delta_1)$ and $A_2 = (Q_2, i_2, F_2, \Sigma, \delta_2)$ such as those shown in Figure 4.13. In this figure, A_1 accepts all strings containing at least one a and A_2 accepts all strings containing at least one b. The intersection $A_1 \cap A_2 = (Q_I, i_I, F_I, \Sigma, \delta_I)$ is constructed considering pairs of states (p, q) where p and q are respectively states in A_1 and A_2. The construction of A_I works as follows.

The first step is to define the set of states Q_I. Since each state in A_I is a pair (p, q), the set of states Q_I is the set of all possible pairs of states. More formally, the set of states Q_I is defined as $Q_I = Q_1 \times Q_2$.

The initial state of A_I is the pair (i_1, i_2). The set of final states must be defined such that A_I accepts if and only if both A_1 and A_2 accept. Hence, the set of final states is the set of pairs (p_f, q_f) such that $p_f \in F_1$ and $q_f \in F_2$.

Finally, the transition function has to be defined such that $\delta_I^*((p,q), w)$ is an accepting state if and only if $\delta_1^*(p, w)$ and $\delta_2^*(q, w)$ are also accepting states in A_1 and A_2. To achieve that, a state (p, q) has a transition carrying the symbol a and going to (r, s) if and only if there is a transition carrying the symbol a from p to r in A_1 and another one from q to s in A_2. Therefore, only transitions appearing in both transducers are considered. More formally, the transition function of a state (q, p) is defined as

$$\delta_I((p,q),a) = \begin{cases} (\delta_1(p,a), \delta_2(q,a)) & \text{if } \delta_1(p,a) \neq \emptyset \wedge \delta_2(q,a) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}, \quad \forall a \in \Sigma.$$

This construction is used in [24] to prove the correctness of the intersection. Figure 4.14 shows the automaton resulting from the intersection of A_1 and A_2 when all unconnected states are removed.

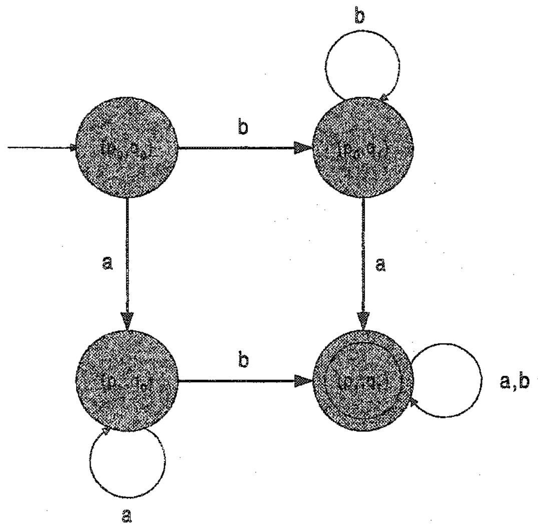


Figure 4.14: *Automaton A_I obtained by the intersection of A_1 and A_2*

As noted before, the composition of weighted string-to-string transducers is a generalization of state-pairs construction. The composed transducer $T_L \bullet T_R$ of two transducers $T_L = (Q_L, i_L, F_L, \Sigma_L, \Delta, E_L, \lambda_L, \rho_L)$ and $T_R = (Q_R, i_R, F_R, \Delta, \Sigma_R, E_R, \lambda_R, \rho_R)$ has pairs of states (l, r) and satisfies the following conditions [21]:

- its initial state is defined as (i_R, i_L),
- the set of final states is defined as $\{(l, r) \mid l \in F_L \text{ and } r \in F_R\}$,
- there is a transition t from (l, r) to (l', r') for each pair of transitions t_l and t_r such that the output symbol of t_l matches the input symbol of t_r.

Consider the transition t_c leaving the state-pair (t_l, t_r). The input symbol, output symbol and weight carried by t_c are respectively the input symbol of t_l, the output symbol of t_r and the $\otimes$-product of weights carried by t_l and t_r.

Algorithm 15 shows the pseudocode of the composition algorithm. In this algorithm, transitions are combined using the $\otimes$-product associated with the semiring over which the transducer is defined. The input of the algorithm are the two transducers T_L and T_R to be composed and its output is another transducer $T_C = (Q_C, i_C, F_C, \Sigma_L, \Sigma_R, E_C, \lambda_C, \rho_C)$.

Algorithm 15 Composition

```
Composition($T_L$,$T_R$)
 1: $F_C \leftarrow \emptyset$
 2: $E_C \leftarrow \emptyset$
 3: $i_C \leftarrow (i_L, i_R)$
 4: $\lambda_C \leftarrow \lambda_L \otimes \lambda_R$
 5: $\rho_C \leftarrow \rho_L \otimes \rho_R$
 6: $Queue \leftarrow i_C$
 7: while $Queue \neq \emptyset$ do
 8:   $(l, r) \leftarrow head[Queue]$
 9:   if $l \in F_L$ and $r \in F_R$ then
10:     $F_C \leftarrow F_C \cup (l, r)$
11:     $\rho_C \leftarrow \rho_L(l) \oplus \rho_R(r)$
12:
13:   for each $(l, \sigma_i, \epsilon, w, l') \in E[l]$ do
14:     $E_C[(l, r)] \leftarrow E_C[(l, r)] \cup ((l, r), \sigma_i, \epsilon, w, (l', r))$
15:     if $(l', r)$ is a new state then
16:       $Queue \leftarrow Queue \cup (l', r)$
17:
18:   for each $(r, \epsilon, \sigma_o, w, r') \in E[r]$ do
19:     $E_C[(l, r)] \leftarrow E_C[(l, r)] \cup ((l, r), \epsilon, \sigma_o, w, (l, r'))$
20:     if $(l, r')$ is a new state then
21:       $Queue \leftarrow Queue \cup (l, r')$
22:
23:   for each $(t_l, t_r) \in \Psi(l, r)$ do
24:     $E_C[(l, r)] \leftarrow E_C[(l, r)] \cup ((l, r), \sigma_i[t_l], \sigma_o[t_r], w[t_l] \otimes w[t_r], (l', r'))$
25:     if $(l', r')$ is a new state then
26:       $Queue \leftarrow Queue \cup (l', r')$
```

The notation used in Algorithm 15 will now be described. Given a transition $t = (q, \sigma_i, \sigma_o, w, q')$, $\sigma_i[t]$ denotes the input symbol carried by t, $\sigma_o[t]$ denotes the output symbol carried by t and $w[t]$ denotes the weight carried by t. $\Psi(l, r)$ denotes the set of transitions pairs (t_l, t_r) such that $\sigma_o[t_l] = \sigma_i[t_r]$, where t_l is a transition in T_L and t_r is a transition in T_R. More formally, this set is defined as

$$\Psi(l, r) = \{(t_l, t_r) \mid t_l \in E_L[l], t_r \in E_R[r] \text{ and } \sigma_o[t_l] = \sigma_i[t_r]\}$$

The algorithm works as follows. Lines 1-5 initialize the resulting transducer. The initial weight of both input transducers are combined using the $\otimes$-product to obtain the initial weight of T_C. The initial state of T_C is the pair of states (i_l, i_r). States to be explored are maintained in a queue which is initialized with the initial state (line 6).

The loop of lines 7-26 is executed for each state in the resulting transducer. Recall that the states of T_C are the states-pair created by the algorithm. A state (l, r) is extracted from the queue at line 8. The state is a final state in T_C if both l and r are final in their respective transducers. The accepting cost of the final state is the $\otimes$-product of $\rho(l)$ and $\rho(r)$ (lines 9-11).

Lines 13-16 deal with the case where state l has leaving transitions carrying an output ϵ-label. Recall that an output ϵ means that no output label is generated when the transition is traversed. In the context of composition, this means that the transitions of r could be matched with transitions of l', the destination state of the transitions carrying an output ϵ-label. Therefore, created transitions leaving (l, r) to (l', r) are identical to transitions carrying an output ϵ-label in l. The state (l', r) is inserted in the queue if not already present.

Lines 18-21 deal with the case where state r has transitions carrying an input ϵ-label. Recall that an input ϵ means that no symbol is consumed when the transition is traversed. Thus, the transitions of l could be matched with those of r', the destination state of transitions with an input ϵ. Therefore, created transitions leaving (l, r) to (l', r) are identical to transitions carrying an output ϵ-label in l. The state (l', r) is inserted in the queue if not already present.

Lines 23-26 consider all pairs of transitions (t_l, t_r) such that the output symbol of t_l matches the input symbol of t_r. For every matching pair, a new transition from state (l, r) to state (l', r'), carrying the input symbol of t_l, the output symbol of t_r and the $\otimes$-product of both weights is created. The state (l', r') is inserted in the queue if not already present.

This procedure is repeated until all created states have been explored and expanded. Figure 4.15 shows two transducers. Their composition is shown in Figure 4.16.

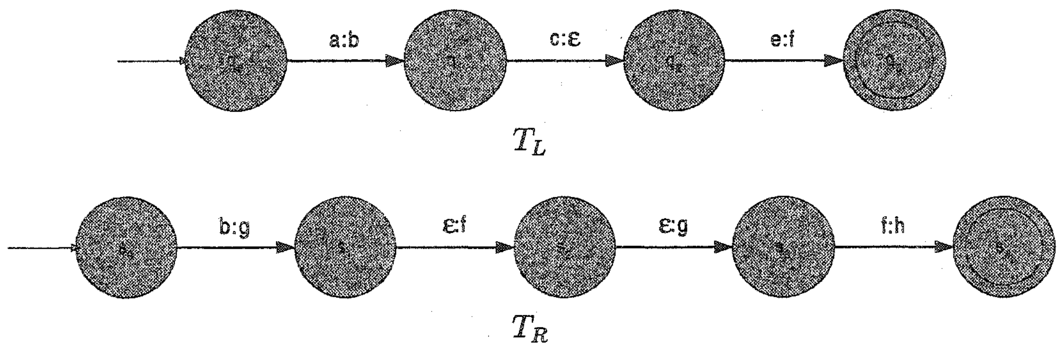


Figure 4.15: *String-to-string transducers T_L and T_R*

Note that several paths can used to reach the final state from the initial one. Each path represents a different way to deal with the ϵ-transitions of the input transducers.

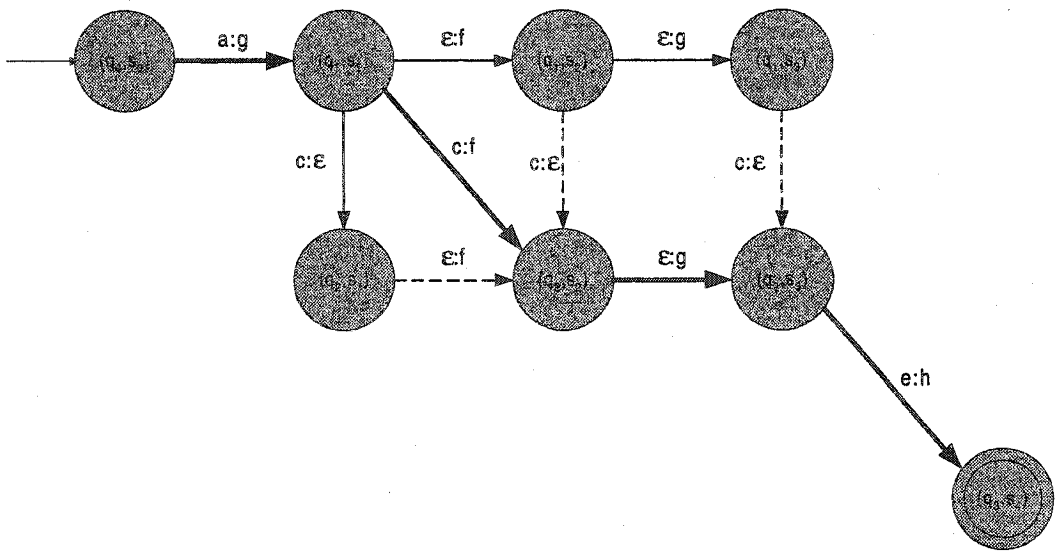


Figure 4.16: *Transducer T_C, resulting of the composition of T_L and T_R*

However, this composition is incorrect in the case of weighted transducers since the weights associated to the possible successful paths could be added in as many times as the number of distinct successful paths [19]. For example, the shortest-distance algorithm, implemented in the real semiring, applied to the composition of the two involved transducers leads to a wrong result since many paths will be considered while there is only one in the cascade made with the two original transducers.

To solve this problem, only one of those paths should be kept. To choose it, a filter T_F is inserted between T_L and T_R. This filter has the effect of removing redundant paths. Figure 4.17 shows one possible filter. In this figure, x denotes any symbol in the alphabet and $\epsilon 1$ and $\epsilon 2$ are special markers which have to be inserted in T_L and T_R [19].

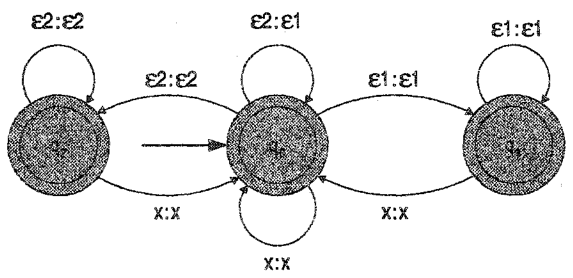


Figure 4.17: *Filter Transducer*

The filter works as follows. As long as the output symbol of T_L matches the input symbol of T_R, the filter remains in state 0 and the transitions are matched. If there is an ϵ-transition in T_L, the filter moves to state 1. In this state, only ϵ-transitions in T_L are considered. The filter remains in this state until a possible match occurs and then returns in state 0. Similarly, if there is an ϵ-transition in T_R, the filter moves to state 2. In this state, only ϵ-transitions in T_R are considered. The filter remains in this state until a possible match occurs and then returns in state 0.

In Figure 4.16, bold transitions denote the transitions retained by the filter and dashed transitions denote those removed by the filter. Note that the resulting transducer contains only one successful path.

This filter can be implicitly implemented in Algorithm 15 with a small modification. Instead of considering state-pairs, the algorithm could consider triplets (l, r, f) where l and r are the states of the original transducers and f is the filter state. Then, according to the value of the filter, ϵ-transitions are created or not. For example, lines 13-16 should be replaced by:

```
if $f \neq 2$ then
  for each $(l, \sigma_i, \epsilon, w, l') \in E[l]$ do
    $E_C[(l,r)] \leftarrow E_C[(l,r)] \cup ((l,r), \sigma_i, \epsilon, w, (l', r))$
    if $(l', r)$ is a new state then
      $Queue \leftarrow Queue \cup (l', r, 1)$
```

In this example, ϵ-transitions from T_L are created only if the state extracted from the queue is not in state 2 of the filter. Then, the destination state of the ϵ-transition is $(l', r, 1)$ since an ϵ-transition of T_L is created.

Not all states of the resulting transducer are connected. Thus, the connection algorithm should be applied on it to remove useless states.

Running Time Analysis

In the worst case, all state-pairs (l, r) will be created. Thus, the loop of lines 7-26 is executed $|Q_L| \cdot |Q_R|$ times. In this loop, transitions are considered. The creation of ϵ transitions is straightforward and depends only on the number of ϵ-transitions in T_L and T_R. Thus the running time of both loops at lines 13-16 and lines 18-21 take respectively $\mathcal{O}(|E_{L_\epsilon}|)$ and $\mathcal{O}(|E_{R_\epsilon}|)$ time.

The running time of the loop at lines 23-26 depends on the time required for computing the set Ψ. This is a well known problem called the relation join. If the transitions of T_R are sorted with respect to the output label and transitions of T_L are sorted with respect to the input label, this operation can be done in $\mathcal{O}(|E_L| \log |E_R|)$ if a binary search is used. Therefore, the running time of the algorithm is

$$\mathcal{O}((|Q_L| \cdot |Q_R|) \cdot (|E_L| \log |E_R|)).$$

4.3.2 Notes on Implementation

There is one critical point in the implementation of the composition algorithm and it is the computation of the set of pairs of transitions which can be matched. In the algorithm, this operation is denoted by $\Psi(l, r)$. This consists in computing the relational join between transitions in $E_L[l]$ and those of $E_R[r]$. The naive way to compute that is a nested loop to compare symbols of all possible pairs of transitions. The running time of this method is $\mathcal{O}(|E_L[l]| \cdot |E_R[r]|)$.

A more efficient method assumes that the transitions of T_L are sorted with respect to the output symbol and transitions of T_R are sorted with respect to the input symbol. This method consists in passing through all transitions in $E_L[l]$ and performing a binary search in $E_R[r]$ to find matching transitions. Since the binary search runs in $\mathcal{O}(\log n)$, the running time of this method is $\mathcal{O}(|E_L[l]| \log |E_R[r]|)$.

In practice, the speed can yet be improved. Indeed, if the number of searches is minimized, the speed of the algorithm will be improved. Accordingly, if $|E_L[l]| \leq |E_R[r]|$, then it is more efficient to pass through transitions in $E_L[l]$ and searching in $E_R[r]$. On the other hand, if $|E_L[l]| \geq |E_R[r]|$, then it is more efficient to pass through transitions in $E_R[r]$ and searching in $E_L[l]$.

Another critical point occurs when a new states-pair have to be inserted in the queue. This new pair is inserted in only if the state is indeed a new one. To assess that, all pairs created so far have to be kept in a set. A naive implementation is via a data structure such as a linked list. However, to ascertain that the pair does not already exist, it has to be compared to all other pairs in the list. A more suitable and efficient structure for this purpose is a hashtable. Indeed, the hashing function will spread out the pairs over the buckets of the hashtable, ensuring that a more reasonable number of pairs is compared with the new candidate.

4.3.3 Lazy Implementation

As is the case for determinization, the composition algorithm admits a lazy implementation. Indeed, the transitions leaving a state of the resulting transducer are computed only through the states-pair representing this state.

4.4 Summary

This section has presented three important operations to work with finite-state transducers:

- Epsilon-removal which removes the ϵ-transitions in a transducer resulting in a more efficient one since epsilons induce a delay in their use (recall that an epsilon does not consume/generate a symbol).
- The Determinization which creates a new deterministic (or sequential) transducer. Recall that a sequential transducer contains at most one sequence of states for any input string. Thus, the complexity depends only on the length of the input string and not on the size of the transducer.
- The composition which is a generalization of the intersection in automata theory. This operation is very important since it allows one to create complex transducers from simpler ones.

These operations allow the creation of efficient and complex transducers which are applicable in many different areas of computer science. The next chapter gives an example of transducers applied to speech recognition.

Chapter 5

Application of FST : Speech Recognition

Traditional speech recognition systems such as HTK are constructed using weighted automata. In speech recognition, the recognition network has many levels of representation. For example, possible sentences are represented by sequences of words which are themselves represented by sequences of phonemes. In the context of automata, these different representations are implemented using the substitution operation. For example, in the graph of words, a transition for a given word w is substituted by a subgraph representing its phonetic sequence. The major disadvantage of this approach is that a change in the network (for example, the addition of a new level of representation) implies that the program performing the search in the recognition network also has to be updated.

The composition operation allows FST to represent many levels of representations in a normalized way. Therefore, the recognizer can work on different recognition networks (with different levels of representation) without modifying the program itself.

This chapter presents how weighted transducers are used to construct a speech recognition system. The chapter begins by the description of each level of representation involved and how transducers implement them. Then, the method used to construct the knowledge network is discussed. Finally, the results obtained by experimentations are given.

Speech recognition is the process by which a computer identifies spoken words by analysing the speech signal. To achieve this, it is assumed that the speech signal is a sequence of symbols composing a message. These symbols are called speech vectors or observations and are extracted from the speech signal at regular intervals. The aim of speech recognition is to map a sequence of vectors of observations to a sequence of symbols such as words, syllables or phonemes.

Let $O = \{o_1, o_2, \cdots, o_t\}$ be a sequence of observations where o_t is the speech vector at time t. The speech recognition problem is to find the message w that maximizes $P(w|O)$. Since this probability is not directly computable, Bayes's Rule is used :

$$P(w|O) = \frac{P(O|w)P(w)}{P(O)} \tag{5.1}$$

where $P(w)$ is the probability associated to the language model and $P(O|w)$ is calculated using parametric models, the most commonly used in speech recognition being the Hidden Markov Model (HMM). Since $P(O)$ is constant for a given sequence of observations and only the arg max matters, this probability is not considered.

From the transducer's point of view, $P(O|w)$ is a transduction between the message and observations. This transduction may involve several stages relating different levels of representation.

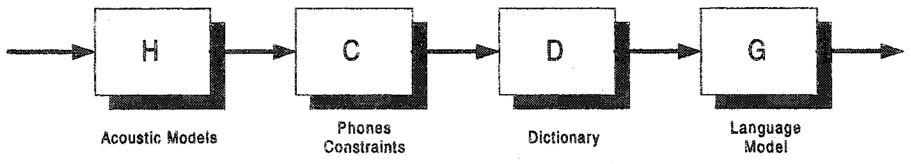


Figure 5.1: *Transducers involved in speech recognition*

Figure 5.1 shows the usual cascade of transducers used in speech recognition. Other intermediate transducers can be added to the chain. For example, transducers representing phonological rules should be added between transducers C and D.

The meaning of each transducer will now be described.

5.1 Transducers Involved in Speech Recognition

5.1.1 Transducer O

This string-to-weight transducer maps, for each observation o_t, every probability distribution function (PDF) d_i to the probability that d_i generates o_t. For each observation at a given time t, a set of transitions carrying a distribution identification d_i and the probability w that the observation was generated by this distribution, is created. Figure 5.2 shows how this acceptor should be implemented.

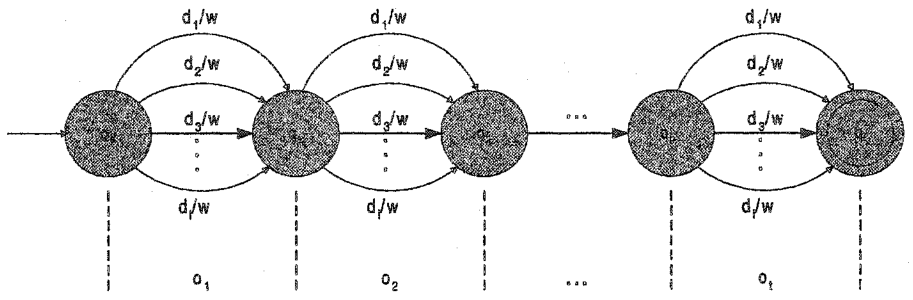


Figure 5.2: *Observations transducer*

The calculation of the PDF can be done in many ways. For example, neural networks or support vector machines could be used to compute this probability. However, the most widely used procedure represents each distribution by Gaussian mixture densities. The probability that o_t is generated by d_i, given a mixture of Gaussian densities, is given by:

$$p(o_t|d_i) = \sum_{m=1}^{M} c_m \frac{1}{\sqrt{(2\pi)^n |\Sigma_i|^n}} e^{-\frac{1}{2}(o_t-\mu_i)\Sigma_i^{-1}(o_t-\mu_i)}$$

where M is the number of Gaussians in the mixture, c_m is the weight of the Gaussian m, μ_i is the mean vector and Σ_i is the covariance matrix associated with distribution i. More details about Gaussian mixture densities and how they are computed can be found in [4]

In practice, this transducer is not really implemented. The recognition process performs an on-demand composition of transducer O and HCDG thus, transitions of O (represented implicitly) are created only when they are required for composition.

5.1.2 Transducer H

Transducer H represents the constraints imposed by modeling method used in speech recognition called, HMM for Hidden Markov Model. HMMs can be used to model phonemes, syllable, words or any larger speech unit. Usually, context-dependent phonemes are used as the speech unit. A triphone is a phoneme modeled according its neighbours. Triphones are denoted $a - b + c$ where b is the modeled phoneme, a and b are the neighbouring phonemes of b.

Transducer H maps a sequence of distributions to a sequence of triphone models (or of any other speech unit). Each triphone is typically modeled with 3 HMM states. Transitions in a HMM carry a distribution index as an input symbol, the transition weight and no output symbol except for the transition leaving the HMM which carries the triphone model associated with the HMM. Figure 5.3 shows the transducer H which is the union of all triphone models.

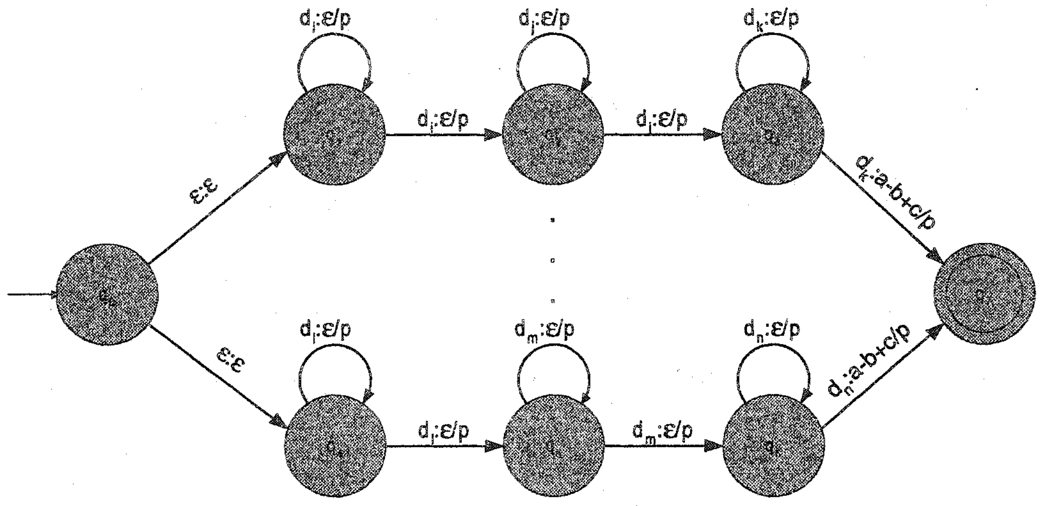


Figure 5.3: *Observations to HMM transducer.*

In this figure, p denotes transition probabilities involved in HMMs, $a - b + c$ is a triphone model and d_i is a distribution.

Note that the self loop present on each state in the HMM can be omitted from the transducer and implemented implicitly in the decoder.

5.1.3 Transducer C

In practice, the number of triphones to model can be very high. Indeed, in English, there are 36 phonemes and thus the number of possible triphones is 36^3. In order to avoid modelling all triphones, only some of them are modelled with a HMM. Modelled triphones are called *physical* triphones and the others are referred to as *logical* triphones.

Logical triphones are mapped to physical ones according to a set of rules. This process is usually done using a decision tree. The first goal of transducer C is to implement this mapping. Figure 5.4 shows how this transducer is constructed.

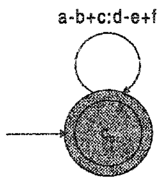


Figure 5.4: *Transducer mapping physical triphones to logical ones.*

The transducer has a self loop transition for every triphone. The input symbol is a triphone, physical or logical, and the output symbol is the physical triphone associated with the input one. Thus, when the input triphone is a physical model, the output symbol is the same triphone.

The second goal of transducer C is to map a sequence of triphones to a sequence of phonemes. However, not all triphone sequences are allowed. A sequence of triphones A, B is allowed if the terminal pair of triphone A matches the pair at the beginning of triphone B. For example, the sequence $a-b+c, b-c+d, c-d+e$ is allowed while $a-b+c, c-d+e$ is not. Figure 5.5 shows how this restriction is implemented with a transducer.

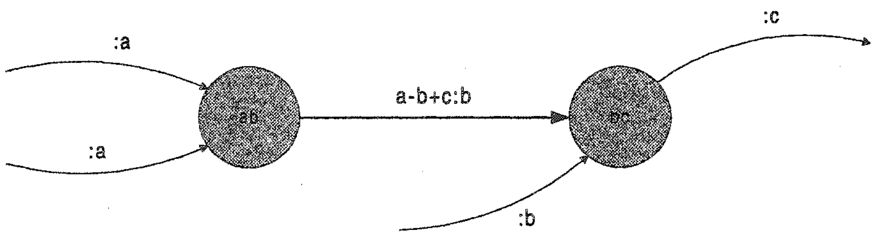


Figure 5.5: *Transducer implementing triphones constraints.*

Each state of the transducer implements a "memory" of the two previous phonemes in the sequence. Transitions leaving a state are those for which the two first phonemes composing the input triphone correspond to the state memory. All ingoing transitions of a state carry an input symbol such that the terminal pair coincides with the memory represented by this state.

5.1.4 Transducer D

In the context of speech recognition, the dictionary is a list of words with their phonetic transcriptions. Thus, the dictionary transducer implement the function $D : p^* \longrightarrow w$ which maps sequence of phonemes p to words w.

A string-to-string transducer is used to represent this relation. Figure 5.6 shows how this transducer is constructed.

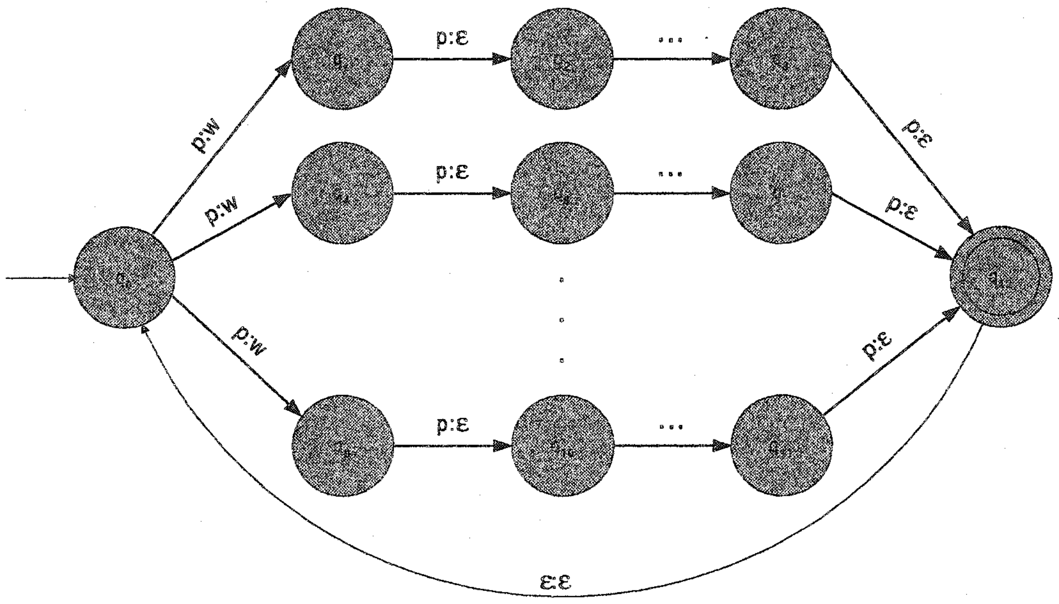


Figure 5.6: *Dictionary Transducer*

In this figure, p is any phoneme and w is a word in the dictionary. The ϵ-transition leaving the final state to the initial state has been added to allow sequences of words. However, this loop transition induces an unbounded delay in the transducer when two words have the same pronunciation (homophones). This point will be discussed later.

5.1.5 Transducer G

Transducer G represents the language model. The language model gives a priori information about the probability of sequence of words $(P(w))$. The transducer shown by Figure 5.7 implements a trigram model. In this model, the probability of a word given the two preceding words in the sequence is denoted $p(w_3|w_1w_2)$.

However, it is possible that a triple of words was not in the text used to train the language model. In this case, the probability of the word given the preceding word $(p(w_3|w_2))$ added to a penalty ψ_{w1w2} called the back-off penalty is used. Similarly, the unigram probability added to the back-off penalty is used when the bigram is also not available.

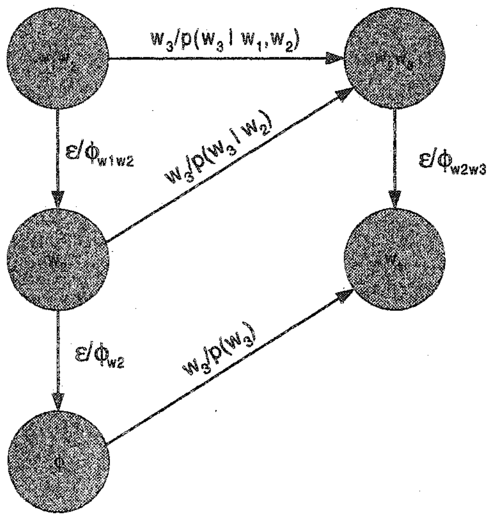


Figure 5.7: *Language Model Transducer*

In transducer G, each state encodes a "memory" of two, one or no words. Transitions leaving a state q carry a word and the probability of this word given the words in the memory of q. In Figure 5.7, ϕ_{w1w2} denotes the back-off penalty for going to a unigram state (state with only 1 word memory).

Transducers can be used to describe other N-gram models such as bigram or 5-gram. They can also be used to describe other types of language models such as grammar based syntactic structure.

5.1.6 Phonological Rules

In natural language, some phonological phenomena at the boundary of words such as the deletion or the insertion of phonemes happen frequently. These phenomena can be modelled with a transducer which can be inserted in the chain of transducers. An example of a pholonological rule is that when the last phoneme of a word is *t* and the first phoneme of the following word is *y*, then *t* and *y* can be optionally replaced by the single phoneme *ch*. This rule applies to words "got you" which can be pronounced in two ways:

$$\text{g aa t} = y\ uw$$
$$\text{g aa} = ch\ uw$$

where the symbol = denotes the word boundary. Figure 5.8 shows how this phonological rule can be implemented by a phoneme-to-phoneme transducer.

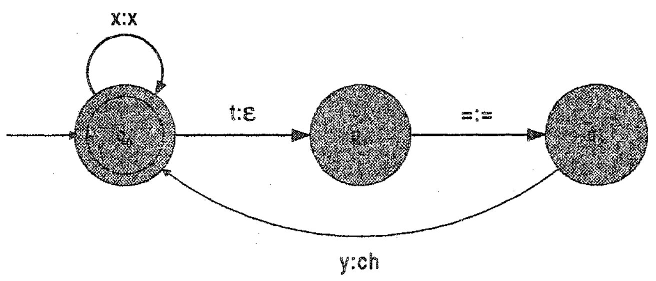


Figure 5.8: *Transducer representing a phonological rule.*

In this figure, the symbol x represents all phonemes in the language and the symbol = is the word boundary. This transducer can be described as follows. All sequences of phonemes are accepted by the transducer thanks to the self loop at the initial state. Moreover, the sequence $t = y$ is replaced by the phoneme $= ch$ since the transition leaving q_0 removes the phoneme t if it is followed by a word boundary and the phoneme y is replaced by ch if it follows the phoneme t and the word boundary. Thus, both sequences are accepted by the transducer which represents the phonological rule.

As noted before, phonological rules can easily be modelled in the recognition network by adding the transducers describing them in the chain of transducers between transducer C and transducer D.

5.2 Transducers combination

The transducer HCDG is constructed using the composition operation. However, in the case of a large vocabulary system, the intermediate results grow very rapidly and there is not enough memory to perform the composition. The problem is solved by using the determinization operation since in the case of transducers used in speech recognition, the determinization considerably decreases the number of states and transitions which is due to redundancy.

Therefore, the creation of HCDG proceeds in several steps. The transducer DG is obtained by the composition $D \bullet G$ and it has to be determinized. Recall that transducer D maps sequences of phonemes to words. The presence of homophones makes transducer DG not determinizable since an unbounded delay is introduced. Indeed, The presence of homophones allows for two different words for the same sequence of phonemes. To make determinization possible, auxiliary phoneme symbols are introduced to distinguish homophones. Figure 5.9 shows an example disambiguated dictionary.

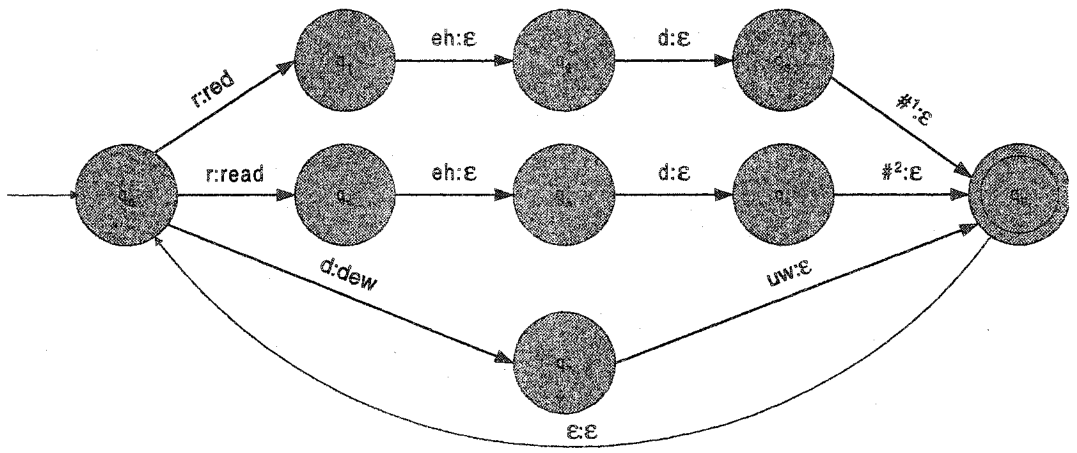


Figure 5.9: *Disambiguated Dictionary Transducer*

Auxiliary symbols are denoted $\#^i$ in the figure. Now, the transducer DG can be determinized and minimized. The next step is the composition $C \bullet DG$. However, the composition will fail since the auxiliary symbols added in D are unknown by C. Therefore, the markers have to be propagated along the cascade by adding to each state of transducer C a self loop $(q, \#^i, \#^i, 0, q)$ for all i.

If the transducer C introduces new ambiguities, other auxiliary symbols have to be used. The same operations are repeated for all steps of the construction of HCDG. Thus, the construction of HCDG is computed by

$$HCDG = Min(Det(H \circ Det(C \circ Det(D \circ G))))$$

where Min denotes the minimization operation and Det is the determinization operation. Auxiliary symbols added during the construction of HCDG have to be removed at the end. The transducers shown in Figure 5.10 remove auxiliary symbols at the input and output by composing them with HCDG as follows: $L \circ HCDG \circ R$.

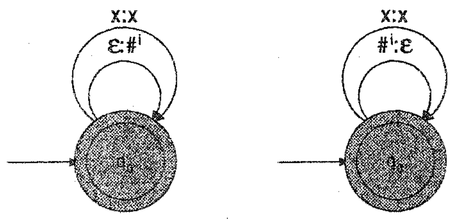


Figure 5.10: *Transducers used to remove auxiliary symbols*

In this figure, x denotes all non-auxiliary symbols.

5.3 Experiments

An implementation of the algorithms described in earlier chapters has been used to construct the recognition transducer for a French vocabulary of 20000 words on the BREF database [9]. The transducer has been constructed as outlined before using these models:

- Accoustic models of 6013 distributions each modeled with a mixture of 8 Gaussians, speaker-independent and gender-independent.
- 17997 models of context-dependent triphones.
- Dictionary of 20000 pronunciations
- Trigram language model of 79845 trigram probabilities, 311131 bigram probabilities and 20003 unigram probabilities.

Transducer	# of states	# of transitions
D	588518	6744116
G	124587	1020488
$D \bullet G$	1439910	2903364
$Det(D \bullet G)$	1108514	2268297
$C \bullet Det(D \bullet G)$	1812045	5168839
$Det(C \bullet Det(D \bullet G))$	1885965	5571984
$H \bullet Det(C \bullet Det(D \bullet G))$	9148639	13278375
$Det(H \bullet Det(C \bullet Det(D \bullet G)))$	8771686	11781976
$Min(Det(H \bullet Det(C \bullet Det(D \bullet G))))$	6095031	9073327

Table 5.1: Size of transducers used to construct the recognition network

Intermediate transducers have been determinized at each step of the construction of HCDG. Table 5.1 gives the size in number of transitions and number of states of all intermediate and final transducers.

The minimization operator is in fact the compaction of transducer such as described in Chapter 2. In order to increase the speed of the recognizer, the final transducer has been sorted topologically with respect to input ϵ-transitions, and its weights have been pushed toward the initial state.

5.3.1 Results

The recognition network has been used to perform recognition on 576 sentences spoken by 87 different speakers. A beam of 130 has been used. The computer used was Pentium III running 700 MHz running under Linux. Results are given in Table 5.2.

Accuracy on words	78.82 %
$x\cdot$ real-time	$2x$

Table 5.2: Results of the recognition task

These results correspond to those obtained with a traditional speech recognition system [6]. However, the transducer-based system has several advantages compared to the traditional one.

First of all, the transducer approach is more flexible since it represents a general framework. Indeed, all speech knowledge is expressed with the same representation and thus, the decoder does not have to be modified when a new level of representation is added to the chain of transducers. This leads to a simpler implementation of the Viterbi decoder. Indeed, when a new level of representation is added in a traditional system, the decoder has to be modified to take it into account.

Another advantage is that the optimization algorithms, such as determinization and minimization, are applied to the entire network whereas in traditional recognizer, optimizations are only applied to local parts of the network [13].

A disadvantage of transducers is that the recognition network construction requires lots of memory since it is entirely constructed. The intermediate transducers are often very big and thus the optimization operations take a lot of memory. In traditional systems, some parts are constructed statically and other parts are constructed during the recognition procedure which helps save memory since only some parts of automata have to be used.

5.4 Summary

This chapter has presented how finite-state transducers can be used to build a speech recognition system. The main points presented in this chapter are:

- Transducers used to construct the knowledge network.
- How transducer operations can be used to construct and optimize the knowledge network.
- The advantages and disadvantages of the transducer approach over the traditional ones.

Chapter 6

Conclusion

The purpose of this work was to describe the algorithms implementing operations on weighted finite-state transducers. These algorithms have been described in the general case of semirings which permit their use with any transducer representing a binary relation mapping a sequence of input symbols to a sequence of output symbols associated with weights.

The following important algorithms have been presented:

Composition:

Composition is an essential operation since it allows to construct complex transducers from smaller ones, each of which represents a different level of representation and whose sequence form a cascade of binary relations. The resulting transducer is equivalent to this cascade in the sense that its binary relation is the same as that represented by the cascade.

Determinization

Determinization can be used to decrease the transducer's size when it contains redundancies. This operation is also used to prepare the transducer for minimization which results in the smallest transducer describing the language. These operations are often used to optimize transducers in the composition process, which can lead to considerable memory savings.

Epsilon-removal

Transducers produced by several operations are often the result of various complex operations introducing ϵ-transitions. These transitions have the disadvantage of inducing a delay in the input symbol processing, which can lead to an explosion of transitions in the composition process. This operation is applied to remove these kinds of transitions without altering the language described by the transducer.

Weight-pushing

In pruning-based applications, this optimization is quite important since the distribution of weights along the paths have a big influence on their execution. The Viterbi decoder is a good example of such an application since the distribution of weights can lead to a 40% improvement in execution speed.

An example of the use of these operations has been given via a description of a speech recognition system based on transducers. The accuracy obtained with this system is comparable to that obtained with traditional systems.

The major advantage of the transducer approach is its flexibility. Indeed, all speech knowledge is expressed with the same representation and thus, the decoder does not have to be modified when a new level of representation is added to the chain of transducers. This leads to a simpler implementation of the Viterbi decoder.

Another advantage is that the optimization algorithms, such as determinization and minimization, are applied to the entire network whereas in traditional recognizers, optimizations are only applied to local parts of the network. This often leads to faster systems [13].

Glossary

Alphabet A finite set of symbols.

Binary relation Function mapping a sequence of symbols to another sequence of symbols.

Connected state A state reachable from the initial state and which can reach a final state.

Cycle A cycle is a path $v_1 \rightsquigarrow v_2$ such that $v_1 = v_2$.

ϵ-cycle An ϵ-cycle is cycle containing only ϵ-transitions.

DFA Deterministic Finite-State Automaton. A FSA in which any input string has a unique sequence of states.

DFS Depth-First Search. Search strategy which consists in exploring a transducer deeper whenever it is possible.

Epsilon transition Transition for which no symbol is consumed or generated.

FSA Finite-State Automaton. Useful model used in computer science consisting of a finite set of states connected with transitions.

FSM Finite-State Machine. See FSA.

FST Finite-State Transducer. A FSA which outputs a string. Transitions in FST carry an output symbol in addition to the usual input symbol of FSA transitions.

HMM Hidden Markov Model. System used to make models in speech recognition.

Language A set of strings.

Language Model Model given a priori information about sequences of words.

Monoid A set with a binary operator and a neutral element over this operator.

NFA Nondeterministic Finite-State Automaton. A FSA which is not deterministic.

Path A sequence of states connected by consecutive transitions.

Path weight Weight associated to a path.

Phonological rule Rule representing words boundary phenomena.

SCC Strongly Connected Component. A SCC is a set of states in which there exists a path between all combination of states.

Semiring A set together with two binary operators and two neutral elements.

Set A group of elements represented as a unit.

State States are the basic elements of FSA. There are three types of states: initial, final and normal states.

String A sequence of symbols.

Transduction Function mapping an input string to an output string.

Transition A transition connects a source state to a destination state. It carries an input symbol and should, in addition, carry either an output symbol, a weight or both.

Transitive closure Extension of the transition function such that if there is a transition between state q_1 and q_2 and another one between q_2 and q_3 then, there is a transition between q_1 and q_3 in the extended transition function.

ϵ-transitive closure Transition closure which takes into account only ϵ-transitions.

WFSA Weighted Finite-State Automaton. FSA which outputs a weight instead of a simple accept/reject value.

WFST Weighted Finite-State Transducer. Transducer which outputs a weight in addition to the output string.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The design and analysis of computer algorithms.* Addison Wesley, 1974.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers Principle, Techniques and Tools.* Addison Wesley, 1986.

[3] AT&T. *FSM Library.* http://www.research.att.com/sw/tools/fsm/tech.html.

[4] C. Becchetti and L.P. Ricotti. *Speech Recognition, Theory and C++ Implementation.* Wiley, 1999.

[5] R. Bellman. On a routing problem. *Quaterly of Applied Mathematics*, 1958.

[6] G. Boulianne and P. Dumouchel J. Brousseau, P. Ouellet. Le système de RAPT du CRIM. In *12e Congrès francophone AFRIF-AFIA de Reconnaissance des Formes et Intelligence Artificielle (RFIA 2000)*, 2000.

[7] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* The MIT Press: Cambridge, 1992.

[8] E.W. Dijkstra. A note on two problems in connexion with graph. *Numerische Mathematik*, 1959.

[9] J.M. Dolmazon, F. Bimbot, M. El Beze, J.C. Caerou, J.Zeiliger, and M. Adda-Decker. ARC B1 - Organisation de la première campagne AUPELF pour l'évaluation des système de dictée vocale. *JST97 FRANCIL*, 1997.

[10] David Eppstein. Finding the k shortest paths. In *IEEE Symposium on Foundations of Computer Science*, pages 154–165, 1994.

[11] L.R. Ford and D.R. Fulkerson. Flows in network. *Princeton University Press*, 1962.

[12] J. Gross and J. Yellen. *Graph Theory and its Application.* Zipper Books, 1999.

[13] S. Kanthak, H. Ney, M. Riley, and M. Mohri. A comparison of two LVR search optimization techniques. In *Proceedings of the International Conference on Spoken Language Processing 2002 (ICSLP '02)*, 2002.

[14] W. Kuich and A. Salomaa. Semirings, automata, languages. *Number 5 in EATCS Monographs on Theorical Computer Science*, 1986.

[15] E.Q. V. Martins, M.M.B. Pascoal, and J.L.E. Dos Santos. The k shortest paths problem. *International Journal of Foundations of Computer Science*, June 1998.

[16] E.Q.V. Martins, M.M.B. Pascoal, and J.L.E. Dos Santos. A new improvement for a k shortest paths algorithm. *Investigacão Operacional*, 2000.

[17] M. Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 1997.

[18] M. Mohri. Generic epsilon-removal algorithm for weighted automata. In *Proceedings of the Fifth International Conference on Implementation and Application of Automata (CIAA'2000)*, 2000.

[19] M. Mohri, F. C. N. Pereira, and M. Riley. Weighted automata in text and speech processing. In *Proceedings of the 12th biennial European Conference on Artificial Intelligence (ECAI-96), Workshop on Extended finite state models of language*, 1996.

[20] M. Mohri, F.C.N. Pereira, and M. Riley. Weighted finite-state transducers in speech recognition. In *Proceedings of the ISCA Tutorial and Research Workshop, Automatic Speech Recognition: Challenges for the new Millenium (ASR2000)*, 2000.

[21] M. Mohri, F.C.N. Pereira, and M. Riley. Weighted finite-state transducers in speech recognition. *Computer and Speech Language*, 2002.

[22] M. Mohri and M. Riley. A weight pushing algorithm for large vocabulary speech recognition. In *Proceedings of the 7th European Conference on Speech Communication and Technology (Eurospeech '01)*, 2001.

[23] M. Mohri and M. Riley. An efficient algorithm for the n-best-strings problem. In *Proceedings of the International Conference on Spoken Language Processing 2002 (ICSLP '02)*, 2002.

[24] J.E. Hopcroft R. Motwani and J.D. Ullman. *Introduction to Automata Theory, Languages & Computability Second Edition.* Addisson-Wesley, 2000.

[25] D. O'Shaughnessy. *Speech Communications.* IEEE Press, 2000.

[26] F.C.N. Pereira and M.D. Riley. *Finite State Language Processing*, chapter Speech Recognition by Composition of Weighted Finite Automata. MIT Press, 1997.

[27] M. Mohri F.C.N. Pereira and M. Riley. The design principles of a weighted finite-state transducer library. *Theoretical Computer Science*, January 2000.

[28] W. Pijls and A. Kolen. A general framework for shortest path algorithms. In *Discussion Paper.* Erasmus University Rotterdam, 1992.

[29] J-E. Pin. Tropical semirings. In *Idempotency.* Cambridge University Press, 1998.

[30] S. Sedgewick. *Algorithms in C++ Part 5: graph Algorithms.* Addison Wesley, 2002.

[31] M. Simon. *Automata Theory.* World Scientific Publishing Co. Pte. Ltd, 1999.

[32] M Sipser. *Introduction to the theory of computation.* PWS Publishing Company, 1997.

[33] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2), 1972.

[34] M. Thorup. On RAM priority queues. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1996.

[35] S. Zhang. Weighted finite-state transducers in speech recognition: A compaction algorithm for non-determinizable transducers. Master's thesis, Université de Montréal, To be submitted,2002.