INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Determinization of String-to-String/Weight Finite State Transducers in Speech Recognition

Jun Qiu

School of Computer Science
McGill University, Montreal
July, 2000

A thesis submitted to the
Faculty of Graduate Studies and Research
In partial fulfillment of the requirements for the degree of
Master of Science

Table of Contents

List of Figures

List of Tables

Résumé

Ce mémoire résume mes travaux de recherche sur la déterminization de transducteurs pondérés de chaînes de caractères vers une chaîne de caractères. Ce document débute par une définition formelle des transducteurs pondérés de chaînes de caractères vers une chaîne de caractères. Par la suite, trois algorithmes de déterminization sont développés: algorithme *SSW_determinization* pour la déterminization complète de transducteurs déterminizables (c'est une reproduction couronnée de succès de la fonction non-documentée *fsmdeterminize* d'AT&T), l'algorithme de déterminization partielle de transducteurs non déterminizables, *PSSW_determinization* et, finalement, l'algorithme de déterminization sur demande, *DSSW_determinization.* Ces algorithmes ont été implantés et des tests fonctionnels incluant des tests de reconnaissance de la parole ont été faits.

Les avantages et désavantages de ces algorithmes sont discutés et analysés. Une attention particulière a été portée à la *DSSW_determinization.* Cet algorithme peut s'appliquer autant pour les transducteurs non déterminizables que ceux déterminizables tout en utilisant un coût de mémoire très petit comparativement aux autres algorithmes.

Abstract

This thesis has carried on a systematic research on the determinization of the string-to-string/weight transducers. It begins from the formal definitions of the string-to-string/weight transducers. Then, three determinization algorithms have been developed. It includes *SSW_determinization* algorithm only for the complete determinization of the determinizable transducers (AT&T determinization software *fsmdeterminize* has been successfully reproduced by this algorithm), the partial determinization algorithm *PSSW_determinization* for the determinization of the non-determinizable transducers, and *DSSW_determinization* algorithm for the determinization on the demand. These algorithms has been implemented, and a functional test (including ASR test) of these determinization programs has also been carried on.

The advantages and disadvantages of these algorithms have been discussed and analyzed. Special attention has been paid to the *DSSW_determinization*. The *DSSW_determinization* can be applied to the determinization of both determinizable and non-determinizable transducers, and it has a very low memory cost compared with that of the *SSW_determinization* and *PSSW_determinization*.

Acknowledgments

I wish to thank my thesis supervisors Professor Pierre Dumouchel, and Professor Gerald Ratzer for their guidance, advice, and encouragement throughout the research. They provided insight into the research of the determinization algorithms for the string-to-string/weight transducers. This thesis is benefited from their careful reading and constructive criticism.

I also truly thank CRIM (Centre de recherche informatique de Montreal). It provided wonderful research environments and financial support for this research.

Lots of people have helped me in the preparation of this thesis. First, I would like to thank Gilles Boulianne and Pierre Ouellet of the team for the collaboration in the development and implementation of the determinization algorithms. I benefited greatly from formal and informal discussions with them.

I wish to thank the School of Computer Science for the graduate courses and the research environment. Thanks to Franca Cianci, Vicki Keirl, Teresa De Angelis, Lise Minogue, and Lucy St-James, for easing the procedure of dealing with the School.

Finally, I wish to thank my wife Huan Adele Wang, for her support and encouragement during my study.

Chapter 1
Introduction

Speech Recognition, also known as Automatic Speech Recognition (ASR), is a wide research area of Computer Science. It is very exciting and challenging. Speech Recognition systems generally assume that the speech signal is a realization of some message encoded as a sequence of one or more symbols [1]. To recognize the underlying symbol sequence given a spoken utterance, the continuous speech waveform is first converted to a sequence of equally spaced discrete parameter vectors. These speech vectors are then transduced into messages by several stages [1,2]. Normally, a transduction stage is modeled by a finite-state device, which is a string-to-string (like the dictionary), string-to-weight (like the language model), or string-to-string/weight transducer (like the hidden Markov models).

The application of string-to-string/weight transducer in natural language and speech processing is a new research area [3]. This area is attracting a great deal of attention in the research of Speech Recognition because that some models in ASR can only be represented by full string-to-string/weight transducers (for example, the hidden Markov models). Therefore, a detailed and systematic research on the string-to-string/weight transducers is necessary.

One important research topic on the string-to-string/weight transducers is their determinization algorithms. The determinization algorithms try to construct an equivalent sequential transducer of a string-to-string/weight transducer. Instead of the original non-sequential transducer this sequential transducer dramatically increases the searching speed in the Speech Recognition process. The running time of sequential transducers for specific input depends linearly, only on the size of the input. In most cases the determinization of transducer not only increase time efficiency but also space efficiency.

The purpose of this thesis is to research of the determinization algorithms for the string-to-string/weight transducers. At present, only an executable determinization software *fsmdeterminize* from AT&T is available in our ASR research. This software can

only be applied for the complete determinization, which is to get the whole resulting sequential transducer of a determinizable string-to-string/weight transducer. However, a goal of the research carried here is to reproduce the AT&T work, and therefore the determinization result obtained from *fsmdeterminize* is considered as a standard to evaluate our newly developed determinization algorithms. In this thesis, besides the complete determinization algorithm other two algorithms, the partial determinization algorithm and the determinization on the demand algorithm are also developed. The thesis is organized as follows:

First, in Chapter 2, we give a review on the Finite-State Transducers and Speech Recognition. Since few publications about string-to-string/weight transducers can be found, this chapter introduces the definitions and properties of automata, of string-to-string transducers, and of string-to-weight transducers. The information from this chapter is very helpful to define the string-to-string/weight transducers and to develop the determinization algorithms for them.

In Chapter 3, we define the string-to-string/weight and sequential string-to-string/weight transducers. Then, based on the determinization algorithm of the string-to-weight transducers an algorithm named *SSW_determinization* is developed for the determinization of the determinizable string-to-string/weight transducers.

In Chapter 4, we discuss the determinizability of the transducers, and then develop an algorithm *PSSW_determinization* for the partial determinization of the non-determinizable string-to-string/weight transducers.

In Chapter 5, we develop the determinization on the demand algorithm, which is named *DSSW_determinization*. This algorithm has specific characteristics compared with that of *SSW_determinization* and *PSSW_determinization*. For example, it can be applied to both determinizable and non-determinizable transducers.

In Chapter 6, we systematically describe the implementation of these newly developed determinization algorithms.

In Chapter 7, we design and present a functional test on these determinization programs.

Finally, in Chapter 8, we summarize the whole thesis with conclusions and future work.

Chapter 2

Review of the Finite-State Transducers and Speech Recognition

2.1 Introduction

Since the emergence of Computer Science finite-state devices, such as finite-state automata, graphs, and finite-state transducers, have been studied and are extensively used in areas such as program compilation, hardware modeling, and database management. Although finite-state devices have been known for some time in computational linguistics, more powerful formalisms such as context-free grammars or unification grammars have typically been preferred. However, the latest mathematical and algorithmic advances in the field of finite-state technology have had a great impact on the representation of electronic dictionaries and on natural language and speech processing. As a result, significant developments have been made in many related research areas [8,16,17,19].

Some of the most interesting applications of finite-state machines are concerned with computational linguistics [4,20,21,23,24]. We can describe these applications from two different views. Linguistically, finite automata are convenient since they allow us to describe easily most of the relevant local phenomena encountered in the empirical study of language by compact representations [5]. Parsing context-free grammars can also be dealt with using finite-state machines, the underlying mechanisms in most of the methods used in parsing are related to automata [6]. From the view of the computational point, the use of finite-state machines is mainly motivated by considerations of time and space efficiency. We know that both time and space concerns are very important in modern computer science development. For examples, in multimedia database management system the size of data is dramatically increased compared to traditional database systems [7]. Similarly in language and speech recognition, a small string-to-string/weight transducer has more than 3500K nodes with 50M arcs. The characteristics in language processing from large-scale dictionaries in morphology to large lexical grammars in syntax need some lexical approaches (underlying are sequential/subsequential transducers

mechanism) to increase processing time and decrease storage space (memory and second storage). Actually, the effect of the size increase on time and space efficiency is the main computational problem not only in language and speech processing but also in modern Computer Science. In language and speech processing, time efficiency is achieved by using deterministic (or sequential) automata. In general, the running time of deterministic finite-state machines for specific input depends linearly, only on the size of the input. Space efficiency is achieved with classical minimization algorithms for deterministic automata. This minimization treatment process is not needed when each transition of the deterministic finite-state machine is only handling symbols but strings. Applications such as compiler construction have shown deterministic finite automata to be very efficient [9]. At present, we cannot find a single university in the world with a Computer Science department without a class to introducing the theory of finite automata.

Recently, much progress has been made in the applications of finite automata in natural language processing which rang from the construction of lexical analyzers, and the compilation of morphological and phonological rules to speech processing [3]. Here, speech recognition is a large field. The transducers we discuss in this thesis are necessary devices for the speech recognition technology. To get the best transducer is one goal in speech processing.

Sequential finite-state transducers are very important devices in natural language and speech processing [1,3,11,14,15]. Sequential finite-state transducers, simply sequential transducers are also called deterministic transducers. This concept is an extension from deterministic automata to transducers with deterministic inputs. That is a machine which outputs a string or/and weights in addition to accepting (deterministic) inputs.

Even though sequential finite-state transducers are now used in all areas of computational linguistics, the recent work in this field is not yet described in Computer Science textbooks. At present, research on the application of transducers on speech recognition is carried on mainly by **AT&T**, **CRIM** and Carnegie Mellon University.

In this chapter we are going to give a detailed description of the related finite-state devices used for language processing and speech recognition. As basic concepts we first give an introduction on the definitions and properties of finite-state automata and finite-

state transducers. Then we consider the case of string-to-string transducers, which have been successfully used in the representation of large-scale dictionaries, computational morphology, and local grammars and syntax. Considered next are sequential string-to-weight transducers. These transducers are very useful in speech processing. Language models, phoneme lattices and word lattices are among the objects that can be represented by these transducers. The related algorithms used for these devices are other focuses of this chapter, and we will also give an introduction on speech recognition systems.

Another goal of this chapter is that by reviewing the representative publications in language and speech processing, to develop and implement (in the following chapters) the determinization algorithms of string-to-string/weight transducers for the speech recognition systems.

2.2 Finite-State Automata

Finite-State-Automata (FSA) can be seen as defining a class of graphs and also as defining languages. The following is a simple description on the definitions of FSA and some closure properties. Other information, such as deterministic FSA (sequential FSA), decidability properties and space & time efficiency discussion are available from references [7,20,21,23,25].

2.2.1 Definitions

Definition 2.2.1 (**FSA**):

A finite-state automaton A is a 5-tuple (Σ, Q, i, F, E),

where:

- Σ is a finite set called the alphabet
- Q is a finite set of states
- $i \in Q$ is the initial state
- $F \subseteq Q$ is the set of final states
- $E \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is the set of edges

By this definition FSAs can be seen as a class of graphs.

Definition 2.2.2 (**extended set of edges**):

The set of strings built on an alphabet Σ is also called the *free monoid* Σ^*. The formal definition of the star * operation can be found in reference [29]. The extended set of edges $\hat{E} \subseteq Q \times \Sigma^* \times Q$ is the smallest set such that

(i) $\forall q \in Q, (q, \varepsilon, q) \in \hat{E}$

(ii) $\forall w \in \Sigma^*$ and $\forall a \in \Sigma \cup \{\varepsilon\}$, if $(q_1, w, q_2) \in \hat{E}$ and $(q_2, a, q_3) \in E$ then $(q_1, w \bullet a, q_3) \in \hat{E}$

Definition 2.2.3 (**extended transition function**):

The transition function d of a FSA is a mapping from $Q \times (\Sigma \cup \{\varepsilon\})$ to 2^Q, and satisfies $d(q', a) = \{q \in Q | \exists (q', a, q) \in E\}$. The extended transition function $\hat{d}$, mapping from $Q \times \Sigma^*$ onto 2^Q, is that function such that

(i) $\forall q \in Q, \hat{d}(q, \varepsilon) = \{q\}$

(ii) $\forall w \in \Sigma^*$ and $\forall a \in \Sigma \cup \{\varepsilon\}, \hat{d}(q, w \bullet a) = \bigcup_{q_1 \in \hat{d}(q, w)} d(q_1, a)$

Now, a language $L(A)$ can be defined on finite-state automaton A:

$$L(A) = \{w \in \Sigma^* | \hat{d}(i, w) \cap F \neq \varnothing\}$$

A language is said to be regular or recognizable if it can be defined by a FSA.

2.2.2 Closure Properties

The set of recognizable language is closed under the following operations:

(1) **Union**. If A_1 and A_2 are two FSAs, it is possible to compute a FSA $A_1 \cup A_2$ such that $L(A_1 \cup A_2) = L(A_1) \cup L(A_2)$.

(2) **Concatenation**. If A_1 and A_2 are two FSAs, it is possible to compute a FSA $A_1 \bullet A_2$ such that $L(A_1 \bullet A_2) = L(A_1) \bullet L(A_2)$.

(3) **Intersection**. If $A_1 = (\Sigma, Q_1, i_1, F_1, E_1)$ and $A_2 = (\Sigma, Q_2, i_2, F_2, E_2)$ are two FSAs, it is possible to compute a FSA denoted $A_1 \cap A_2$ such that $L(A_1 \cap A_2) = L(A_1) \cap L(A_2)$. Such an automaton can be constructed as follows:

$$A_1 \cap A_2 = (\Sigma, Q_1 \times Q_2, (i_1, i_2), F_1 \times F_2, E) \text{ with}$$

$$E = \bigcup_{(q_1,a,r_1)\in E_1,\,(q_2,a,r_2)\in E_2} \{((q_1,q_2),a,(r_1,r_2))\}.$$

(4) **Complementation**. If A is a FSA, it is possible to compute a FSA $-A$ such that $L(-A) = \Sigma^* - L(A)$.

(5) **Kleene Star**. If A is a FSA, it is possible to compute a FSA A^* such that $L(A^*) = L(A)^*$.

2.3 Finite-State Transducers

Finite-State Transducer (FST) is an extension of a FSA. Each arc in a FST is labeled by a pair of symbols rather than by a single symbol.

2.3.1 Definitions

Definition 2.3.1 (**FST**):

A Finite-State transducer is a 6-tuple $(\Sigma_1, \Sigma_2, Q, i, F, E)$,

where:

- Σ_1 is the input alphabet among a finite set
- Σ_2 is the output alphabet among a finite set
- Q is a finite set of states
- $i \in Q$ is the initial state
- $F \subseteq Q$ is the set of final states
- $E \subseteq Q \times \Sigma_1 \times \Sigma_2 \times Q$ is the set of edges

Definition 2.3.2 (**path**):

If a FST $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$, a path of T is a sequence $((p_i,a_i,b_i,q_i))_{i=1,n}$ of edges E such that $q_i = p_{i+1}$ for $i = 1$ to n-1. Where, (p_i,a_i,b_i,q_i) is an edge, q_i is a state which can be reached from state p_i with an input symbol a_i and an output symbol b_i.

Definition 2.3.3 (**successful path**):

Given a FST $T = (\Sigma_1, \Sigma_2, Q, i, F, E)$, a successful path $((p_i,a_i,b_i,q_i))_{i=1,n}$ of T is a path of T such that $p_1 = i$ and $q_n \in F$.

These definitions only are part of the important definitions on Finite-State transducers. The concepts defined by these definitions will be frequently used in the following sections. Other definitions and closure properties (for example Union, Inversion, Letter transducer including ε-free transducer and Composition) on FSTs can be found from the literature [12,25].

2.4 Sequential String-to-String Transducers

Sequential string-to-string transducers are the most useful transducers used in natural language and speech processing. Many works have been done on this topic [3,12,22,25].

2.4.1 Sequential Transducers

In language and speech processing, sequential transducers are defined as transducers with a deterministic input (string or just a symbol). At any state of such transducers, at most one outgoing arc is labeled with a given element of the alphabet. This means the input is distinct. The output label might be a string (or a single symbol), including the empty string ε. Of course, the output of a sequential transducer is not necessarily deterministic. The formal definition of a sequential string-to-string transducer is as follows:

A sequential string-to-string transducer is a 7-tuple $(Q, i, F, \Sigma, \Delta, \delta, \sigma)$, where:

- Q is the set of states
- $i \in Q$ is the initial state
- $F \in Q$, the set of final states
- Σ and Δ, finite sets corresponding respectively to the input and output alphabets of the transducer
- δ, the state transition function which maps $Q \times \Sigma$ to Q
- σ, the output function which maps $Q \times \Sigma$ to Δ^*

δ and σ are partial functions (a state $q \in Q$ does not necessarily admit outgoing transitions labeled on the input side with all elements of the alphabet). These functions can be extended to mappings from $Q \times \Sigma^*$ by the following classical recurrence relations:

$\forall s \in Q, \forall w \in \Sigma^*, \forall a \in \Sigma, \quad \delta(s, \varepsilon) = s, \delta(s, wa) = \delta(\delta(s, w), a);$

$\sigma(s, \varepsilon) = \varepsilon, \sigma(s, wa) = \sigma(s, w)\sigma(\delta(s, w), a).$

Thus, a string $w \in \Sigma^*$ is accepted by T iff $\delta(i, w) \in F$, and in that case the output of the transducer is $\sigma(i, w)$.

2.4.2 Subsequential and p-Subsequential Transducers

Subsequential transducers are an extension of sequential transducers. By introducing the possibility of generating an additional output string at the final states the application of the transducer to a string can then possibly finish with the concatenation of such an additional output string to the usual output. Such extended sequential transducers with an additional output string at final states are called subsequential transducers.

Language processing often requires a more general extension. Indeed, the ambiguities encountered in language (for example ambiguity of grammars, ambiguity of morphological analyzers, or ambiguity of pronunciation dictionaries) cannot be handled by sequential or subsequential transducers because these devices only have a single output to a given input. Since we cannot find any reasonable case in language in which the number of ambiguities would be infinite, we can efficiently introduce p-subsequential transducers, namely transducers provided with at most p final output strings at each final state to deal with linguistic ambiguities. However, the number of ambiguities could be very large in some cases. Notice that 1-subsequential transducers are exactly the subsequential transducers.

Composition operations are very useful since they allow the construction of more complex transducers from simpler ones [30]. Considering the relationships reflected by these transducers are mappings from strings to strings. So, the composition operation defined for mappings can be used by these transducers. For example, if we have two sequential/p-subsequential transducers T_1 and T_2 the result of an application of $T_2 \circ T_1$ to a string w can be computed by first considering all output strings associated with the input w in the transducer T_1, then applying T_2 to all these strings. The output strings obtained after this application represent the result $(T_2 \circ T_1)(w)$. In fact, instead of waiting for the result of the application of T_1 to be completely given, one can gradually apply T_2 to the

output strings of T_1 yet to be completed. This is the basic idea of the composition algorithm, which allows one to construct directly the transducer $T_2 \circ T_1$ given T_1 and T_2.

A very important concept here is the sequential/p-subsequential function. Similarly, we define sequential/p-subsequential functions to be those functions that can be represented by sequential/p-subsequential transducers. The following theorems give a brief introduction on the characterizations and properties of subsequential and p-subsequential functions (of course, also that of sequential and p-subsequential transducers). Here, the expression p-subsequential means two things, the first is that a finite number of ambiguities is admitted, the second indicates that this number equals exactly p.

Theorem 2.4.1 (**composition**):

Let $f : \Sigma^* \rightarrow \Delta^*$ be a sequential/p-subsequential and $g : \Delta^* \rightarrow \Omega^*$ be a sequential/q-subsequential function, then $g \circ f$ is sequential/pq-subsequential.

Theorem 2.4.2 (**union**):

Let $f : \Sigma^* \rightarrow \Delta^*$ be a sequential/p-subsequential and $g : \Delta^* \rightarrow \Omega^*$ be a sequential /q-subsequential function, then $g + f$ is 2-subsequetial/$(p + q)$-subsequential.

The linear complexity of their use makes sequential or p-subsequential transducers both mathematically and computationally of particular interest. However, not all transducers, even when they realize functions (rational functions), admit an equivalent sequential or subsequential transducer. More generally, sequential functions can be characterized among rational functions by the following theorem.

Theorem 2.4.3 (**characterization of sequential function**):

Let f be a rational function mapping Σ^* to Δ^*. f is sequential iff there exists a positive integer K such that:

$$\forall u \in \Sigma^*, \forall a \in \Sigma, \exists w \in \Delta^*, |w| \leq K: f(ua) = f(u)w$$

That is, for any string u and any element a, $f(ua)$ is equal to $f(u)$ concatenated with some bounded string, Notice that this implies that $f(u)$ is always a prefix of $f(ua)$, and more generally that if f is sequential then it preserves prefixes.

The fact that not all rational functions are sequential could reduce the interest of sequential transducers. The following theorem shows however that transducers are exactly compositions of left and right sequential transducers.

Theorem 2.4.4 (**composition of left and right sequential transducers**):

Let f be a partial function mapping Σ^* to Δ^*. f is rational iff these exist a left sequential function l: $\Sigma^* \rightarrow \Omega^*$ and a right sequential function r: $\Omega^* \rightarrow \Delta^*$ such that $f = r \circ l$.

Left sequential functions or transducers are those we previously defined. Their application to a string proceeds from left to right. Right sequential functions apply to strings from right to left. According to the theorem, considering a new sufficiently large alphabet Ω allows one to define two sequential functions l and r decomposing a rational function f. This result considerably increases the importance of sequential functions in the theory of finite-state machines as well as in the practical use of transducers.

Sequential transducers offer other theoretical advantages. In particular, while several important tests such as the equivalence are undecidable with general transducers, sequential transducers have the following decidability property.

Theorem 2.4.5 (**decidability**):

Let T be a transducer mapping Σ^* to Δ^*. It is decidable whether T is sequential.

The following theorems describe the characterizations of subsequential and p-subsequential functions.

Theorem 2.4.6 (**characterization of subsequential function**):

Let f be a partial function mapping Σ^* to Δ^*. F is subsequential iff:

(1) f has bounded variation

(2) for any rational subset Y of Δ^*, $f^{-1}(Y)$ is rational

Theorem 2.4.7 (**characterization of *p*-subsequential function**):

Let $f = (f1, ..., fp)$ be a partial function mapping $D\ o\ m(f) \subseteq \Sigma^*$ to $(\Delta^*)^p$. f is p-subsequential iff:

(1) f has bounded variation

(2) for all i $(1 \leq i \leq p)$ and any rational subset Y of Δ^*, $f_i^{-1}(Y)$ is rational

Theorem 2.4.8 (**characterization of *p*-subsequential function**):

Let f be a rational function mapping Σ^* to $(\Delta^*)^p$. f is p-subsequential iff it has bounded variation.

2.5 String-to-Weight Transducers [3,10]

2.5.1 String-to-Weight Transducers

String-to-weight transducers are transducers with input strings and output weights. Normally the weights are interpreted as (negative) logarithms of probabilities. String-to-weight transducers are very useful and widely used in various domains such as language modeling, representation of word or phonetic lattices. In most applications to natural language processing string-to-weight transducers are used in the following way: start from the initial state, read and follow a path corresponding to a given input string and output a number obtained by adding the weights up along this path. If the transducer is not sequential, that is when it does not have a deterministic input, we have to proceed in the same way for all the paths corresponding to the input string. In natural language processing, specifically in speech processing, we keep the path with minimum of the weights associated to these paths. This corresponds to the Viterbi approximation in speech recognition or in other related areas in which hidden Markov model (HMMs) are used. In all such applications, we choose the path with the minimum weight as the best path. The formal definition of a string-to-weight transducer is as follows:

A string-to-weight transducer T is defined by $T = (Q, \Sigma, I, F, E, \lambda, \rho)$, where:

- Q is a finite set of states
- Σ the input alphabet

- $I \subseteq Q$ is the set of initial states
- $F \subseteq Q$, the set of final states
- $E \subseteq Q \times \Sigma \times R_+ \times Q$ a finite set of transitions, where R_+ is the set of output weights
- λ the initial weight function mapping I to R_+
- ρ the final weight function mapping F to R_+

Compared to the definition of a string-to-string transducer, we can define for T a partial transition function δ mapping $Q \times \Sigma$ to 2^Q by:

$$\forall (q, a) \in Q \times \Sigma, \delta(q, a) = \{q' | \exists x \in R_+ : (q, a, x, q') \in E\},$$

and an output function σ mapping E to R_+ by:

$$\forall t = (p, a, x, q) \in E, \sigma(t) = x.$$

The following concepts and extensions are very important for string-to-weight transducers. Although we have defined some of them in section 3 in general, more details based on string-to-weight transducer are introduced.

A **path** π in T from $q \in Q$ to $q' \in Q$ is a set of successive transitions from q to q': $\pi = ((q_0, a_0, x_0, q_1), \ldots, (q_{m-1}, a_{m-1}, x_{m-1}, q_m))$, with $\forall i \in [0, m-1]$, $q_{i+1} \in \delta(q_i, a_i)$. We can extend the definition of σ to paths by: $\sigma(\pi) = x_0 x_1 \ldots x_{m-1}$.

The $\pi \in q \sim q' | w$ refers to the set of paths from q to q' labeled with the input string w. The definition of δ can be extended to $Q \times \Sigma^*$ by:

$\forall (q, w) \in Q \times \Sigma^*, \delta(q, w) = \{q' : \exists$ path π in T, $\pi \in q \sim q' | w\}$ and to $2^Q \times \Sigma^*$, by:

$$\forall R \subseteq Q, \forall w \in \Sigma^*, \delta(R, w) = \bigcup_{q \in R} \delta(q, w).$$

The **minimum of the outputs** of all paths from q to q' labeled with w is defined as:

$$\theta(q, w, q') = \min_{\pi \in (q \sim q') | w} \sigma(\pi).$$

A **successful path** in T is a path from an initial state to a final state. A string $w \in \Sigma^*$ is accepted by T iff there exists a successful path labeled with w: $w \in \delta(I, w) \cap F$. The output corresponding to an accepted string w is then obtained by taking the minimum of the outputs of all successful paths with input label w:

$$\min_{(i,f) \in I \times F: f \in \delta(i,w)} (\lambda(i) + \theta(i, w, f) + \rho(f)).$$

A transducer T is said to be **trim** if all states of T belong to a successful path. String-to-weight transducers clearly realize functions mapping Σ^* to R_+. Since the operations we need to consider are addition and min, and since $(R_+ \cup \{\infty\}, \min, +, \infty, 0)$ is a **semiring** (this semiring is also called a tropical semiring, and is widely used in language and speech processing), we call these functions **formal power series**. They have the following characterizations which we imported from formal language theory [24,25]:

1) (S, w) is the image of a string w by a formal power series S . (S, w) is called the coefficient of w in S,
2) by the coefficients, $S = \Sigma_{w \in \Sigma^*} (S, w)w$ can be used to define a power series,
3) the **support** of S is the language defined by:

$$supp(S) = \{w \in \Sigma^* : (S, w) \neq \infty\}.$$

A formal power series S is rational iff it is realizable by a string-to-weight transducer (recognizable).

A string-to-weight transducer T is said to be **unambiguous** if for any given string w there exists at most one successful path labeled with w.

2.5.2 String-to-Weight Sequential Transducers

Recall that a transducer is said to be sequential if its input is deterministic, that is, if at any state there exists at most one outgoing transition labeled with a given element of the input alphabet Σ. Sequential string-to-weight transducers have many advantages over non-sequential string-to-weight transducers, such as time and space efficiency. But not each string-to-weight transducer has an equivalent sequential string-to-weight transducer. The formal definition of a sequential string-to-weight transducer is follows:

Definition 2.5.1 (**sequential transducer**):

A string-to-weight sequential transducer $T = (Q, i, F, \Sigma, \delta, \sigma, \lambda, \rho)$ is an 8-tuple, where:

- Q is the set of its states

- $i \in Q$ its initial state
- $F \subseteq Q$ the set of final states
- Σ the input alphabet
- δ the transition function mapping $Q \times \Sigma$ to Q, δ can be extended as in the string case to map $Q \times \Sigma^*$ to Q
- σ the output function which maps $Q \times \Sigma$ to R_+, where R_+ is the set of output weights, σ can also be extended to $Q \times \Sigma^*$
- $\lambda \in R_+$ the initial weight
- ρ the final weight function mapping F to R_+

A string $w \in \Sigma^*$ is accepted by a sequential transducer T if there exists $f \in F$ such that $\delta(i, w) = f$. Then the output associated to w is: $\lambda + \sigma(i, w) + \rho(f)$.

Considering the benefits of time and space efficiency the sequential transducer is preferred in language and speech processing. But, like we mentioned before, not all transducers are sequential transducers. The process used to transfer a non-sequential transducer to an equivalent sequential transducer is called determinization. Unfortunately, not all transducers have an equivalent sequential transducer, which also means that not all transducers can be determinized. The following definition can be used to determine whether a transducer can admit determinization.

Definition 2.5.2 (**determinization**):

Two states q and q' of a string-to-weight transducer $T = (Q, I, F, \Sigma, \delta, \sigma, \lambda, \rho)$, not necessarily sequential, are said to be **twins** if:

$\forall (u, v) \in (\Sigma^*)^2, (\{q, q'\} \subset \delta(I, u), q \in \delta(q, v), q' \in \delta(q', v)) \Rightarrow \theta(q, v, q) = \theta(q', v, q')$.

If some two states q and q' of a string-to-weight transducer T are twins we say T has **twins property**. If a string-to-weight transducer has twins property it is determinizable.

Notice that according to the definition, two states that do not have cycles with the same string v are twins. In particular, two states that do not belong to any cycle are necessarily twins. Thus, an acyclic transducer has the twins property.

The sequential power series in the tropical semiring, namely functions that can be realized by sequential string-to-weight transducers. Many rational power series defined on the tropical semiring considered in practice are sequential, in particular acyclic transducers represent subsequential power series.

The following theorem gives an intrinsic characterization of sequential power series:

Threorem 2.5.1 (**characterization of sequential power series**):
Let S be a rational power series defined on the tropical semiring. S is sequential iff it has bounded variation.

The proof on this theorem is based on twins property [3].

2.6 Determinization Algorithm for Power Series

In speech recognition systems the tropical semiring is widely used. The determinization algorithm will be frequently applied to the power series defined on tropical semiring. Therefore, the following algorithm is presented in the case of a tropical semiring $(R_+ \cup \{\infty\}, \min, +, \infty, 0)$ on which the transducer is defined. This algorithm is easily changed to fit other semirings by replacing **min** and **+** by their own binary operations.

The following determinization algorithm constructs an equivalent string-to-weight sequential transducer $T_2 = (Q_2, i_2, F_2, \Sigma, \delta_2, \sigma_2, \lambda_2, \rho_2)$ to a given non-sequential one $T_1 = (Q_1, \Sigma, I_1, F_1, E_1, \lambda_1, \rho_1)$ defined on tropical semiring [3,10,22].

PowerSeriesDeterminization(T_1, T_2)

1 $F_2 \leftarrow \varnothing$

2 $\lambda_2 \leftarrow \min_{i \in I_1} \lambda_1(i)$

3 $i2 \leftarrow \bigcup_{i \in I_1} \{(i, \lambda_2^{-1} + \lambda_1(i))\}$

4 $Q \leftarrow \{i_2\}$

5 **while** $Q \neq \varnothing$

6 **do** $q_2 \leftarrow$ head[Q]

7 **if** (there exists $(q, x) \in q_2$ such that $q \in F_1$)

8 **then** $F_2 \leftarrow F_2 \cup \{q_2\}$

9 $\rho_2(q_2) \leftarrow \min_{q \in F_1,\, (q,\, x) \in q_2} \quad x + \rho_1(q)$

10 **for** each a such that $\Gamma(q_2, a) \neq \varnothing$

11 **do** $\sigma_2(q_2, a) \leftarrow \min_{(q,\, x) \in \Gamma(q_2,\, a)} [x + \min_{t = (q,\, a,\, \sigma_1(t),\, n_1(t)) \in E_1} \quad \sigma_1(t)]$

12 $\delta_2(q_2, a) \leftarrow \bigcup_{q' \in \nu(q_2,\, a)} \quad \{(q', \quad \min_{(q,\, x,\, t) \in \chi(q_2,\, a),\, n_1(t) = q'}$

$[\sigma_2(q_2, a)]^{-1} + x + \sigma_1(t)\}$

13 **if** ($\delta_2(q_2, a)$ is a new state)

14 **then ENQUEUE**(Q, $\delta_2(q_2, a)$)

15 **DEQUEUE**(Q)

The key points in this algorithm are further explained as follows:

1. Line 2 and line 3 tell us that the initial weight λ_2 of τ_2 is the minimum of all the initial weights of T_1. The initial state i_2 is a subset made of pairs (i, x), where i is an initial state of T_1, and $x = \lambda_1(i) - \lambda_2$. We use a queue Q to maintain the set of subsets q_2 to be examined. Initially, Q contains only the subset i_2. The subsets q_2 are the states of the resulting transducer. Q_2 is a final state of T_2 iff it contains at least one pair (q, x), with q a final state of T_1 (line 7-8). The final output associated to q_2 is then the minimum of the final outputs of all the final states in q_2 combined with their respective residual weight (line 9).

2. For each input label a such that there exists at least one state q of the subset q_2 admitting an outgoing transition labeled with a, one outgoing transition leaving q_2 with the input label a is constructed (line 10-14). The output $\sigma_2(q_2, a)$ of this transition is the minimum of the outputs of all the transitions with input label a that leave a state in the subset q_2, when combined with the residual weight associated to that state (line 11).

3. The destination $\delta_2(q_2, a)$ of the transition leaving q_2 is a subset made of pairs (q', x'). where q' is a state of T_1 that can be reached by a transition labeled with a, and x' the corresponding residual weight (line 12). x' is computed by taking the minimum of all the transitions with input label a that leave a state q of q_2 and reach q', when combined with the residual weight of q minus the output weight $\sigma_2(q_2, a)$. Finally, $\delta_2(q_2, a)$ is enqueued in Q iff it is a new subset.

4. $n_1(t)$ is the destination state of a transition $t \in E_1$. Hence, $n_1(t) = q'$, if $t = (q, a, x, q') \in E_1$. The sets $\Gamma(q_2, a)$, $\chi(q_2, a)$ and $\nu(q_2, a)$ used in the algorithm are defined by:

$$\Gamma(q_2, a) = \{(q, x) \in q_2 : \exists t = (q, a, \sigma_1(t), n_1(t)) \in E_1\}$$

$$\chi(q_2, a) = \{(q, x, t) \in q_2 \times E_1 : t = (q, a, \sigma_1(t), n_1(t)) \in E_1\}$$

$$\nu(q_2, a) = \{(q' : \exists (q, x) \in q_2 : \exists t = (q, a, \sigma_1(t), q') \in E_1\}$$

$\Gamma(q_2, a)$ denotes the set of pairs (q, x), elements of the subset q_2, having transitions labeled with the input a. $\chi(q_2, a)$ denotes the set of triples (q, x, t) where (q, x) is a pair in q_2 such that q admits a transition with input label a . $\nu(q_2, a)$ is the set of states q' that can be reached by transitions labeled with a from the states of the subset q_2.

Notice that several transitions might reach the same state with different residual weights. Since we are only interested in the best path, namely the path corresponding to the minimum weight, we can keep the minimum of these weights for a given state element of a state (line 11 of the algorithm).

The complexity (both space and time) of this power series determinization algorithm is exponential. However, in some cases in which the degree of nondeterminism of the initial transducer is high, the determinization algorithm turns out to be fast and the resulting transducer has less states. For example, in the speech recognition research group of **CRIM** we use a revised power series determinization algorithm (developed in Chapter 3) to determinize a string-to-string/weight transducer with 1.5M states and 24.7M arcs, the resulting sequential transducer only has 1.0M states and 4.6M arcs. This algorithm is very efficient in practice.

It has been proved that if the determinization algorithm terminates, then the resulting transducer T_2 is equivalent to T_1 [8].

This power series determinization algorithm is applied to a tropical semiring, that is the string-to-weight transducer. In our speech recognition research group we use string-to-string/weight transducers. Actually, it is possible to develop other determinization algorithms for different semirings based on this PowerSeriesDeterminization algorithm. For weighted string-to-string transducers, subsets in the algorithm are made of triples (q, w, x) where q is a state of the original transducer, w is a residual string and x is a residual weight. So, we have to consider the pair (w, x) as output in the determinization of string-to-string/weight transducers. Also some special cases have to be handled. In fact, based on this general algorithm an efficient determinization algorithm has been developed to deal with the string-to-string/weight transducers. Details about this developed algorithm and its implementation will be shown in the following chapter of this thesis.

2.7 Determinizable Transducers

The determinizable transducers can be simple defined as those transducers with which the determinization algorithm terminates. If a transducer is not determinizable the algorithm will keep running until resources are used up.

It has been declared early in this paper that the complexity of the application of sequential transducers is linear in the size of the string to which it applies. This property makes it worthwhile to use the power series determinization in order to speed up the application of transducers. Unfortunately, not all transducers can be determinized using the power series determinization because determinization does not apply to all transducers. Therefore it is important to be able to test the determinizability of a transducer.

We have known (definition 2.5.2) that if a transducer defined on the tropical semiring has the twins property then it is determinizable. There are transducers that do not have the twins property and that still determinizable. Normally it is not a easy job to characterize such transducers because we need more complex conditions [3].

Actually, if we wish to construct the result of the determinization of transducer T for a given input string w, we do not need to expand the whole result of the determinization, but only the necessary part of the determinized transducer. When restricted to a finite set the function realized by any transducer is sequentiable since it has

bounded variation. Acyclic transducers have the twins property, so they are determinizable. Therefore, it is always possible to expand the result of the determinization algorithm for a finite set of input strings, even if T is not determinizable [5].

2.8 Minimization of Transducers

Normally, the minimization operation is applied to a sequential transducer, to reduce its size for the space efficiency [13,18,26]. We already have a successful minimization algorithm for sequential power series defined on the tropical semiring. However, when we are handling a transducer where the inputs of its transitions are only symbols, this sort of minimization algorithm is not applicable because the minimization result will generate string input.

In our research group we use string-to-string/weight transducers with only symbols as the inputs of their transitions, thus we do not need this sort of minimization algorithm. However, one of the most interesting and most important aspects of our research is that can we reduce the size of a transducer if the original transducer is not determinizable (to improve time and space efficiency). Therefore, the concept of minimization in our research is defined as the algorithm used to partially determinize non-determinizable string-to-string/weight transducers. The details about this algorithm are described in chapter 4 of this thesis.

2.9 Introduction to Speech Recognition

Speech recognition systems generally assume that the speech signal is a realization of some message encoded as a sequence of one or more symbols (see Figure 2.1[1]). To recognize the underlying symbol sequence given a spoken utterance, the continuous speech waveform is first converted to a sequence of equally spaced discrete parameter vectors. This sequence of parameter vectors is assumed to form an exact representation of the speech waveform on the basis that for the duration covered by a single vector (around 10 ms), the speech waveform can be reasonably regarded as being quasi-stationary [1].

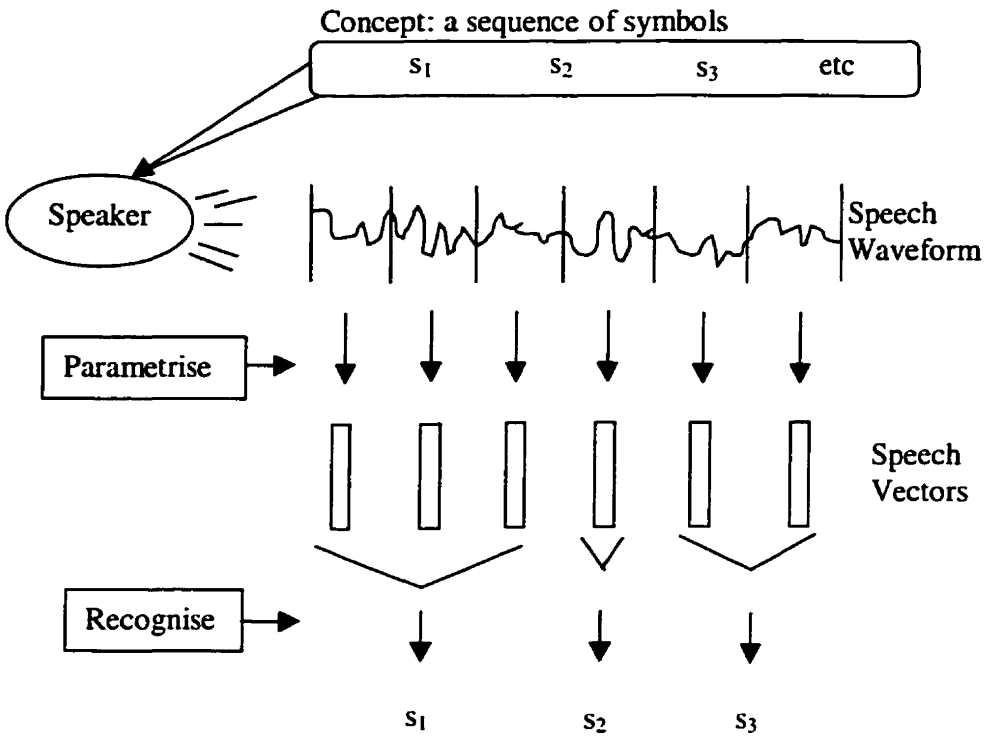


Figure 2.1 Message Encoding/Decoding

The recognizer is the most important part of speech recognition, its role is to figure out what is the underlying symbol sequence from a sequence of speech vectors. Thus, the main aspect of current speech processing can be simply described as : given an observation sequence *o* (or speech vectors), find which intended message *w* is most likely to generate that observation sequence by maximizing [11]:

$$P(w, o) = P(o|w)P(w)$$

Where, $P(o|w)$ refers to the probability of the transduction between intended messages and observations, and $P(w)$ is the frequency of the message being spoken. More generally, the transduction between messages and observations may involve several stages relating successive levels of representation :

$$P(s_0, s_k) = P(s_k|s_0)P(s_0)$$

$$P(s_k|s_0) = \Sigma_{s_1, \ldots, s_{k-1}} P(s_k|s_{k-1}) \ldots P(s_1|s_0) \quad (1)$$

Each s_j is a sequence of units of an appropriate representation, for instance phonemes or syllables in speech recognition. We know that at any intermediate level :

$$P(s_j|s_i) = \Sigma_{s_l} P(s_j|s_l)P(s_l|s_i)$$

For computational reasons, formulation (1) can be approximated as following:

$$\widehat{P}(s_0, s_k) = \widehat{P}(s_k|s_0) + \widehat{P}(s_0)$$

$$\widehat{P}(s_k|s_0) \approx \min_{s_1, \ldots, s_{k-1}} \Sigma_{1 \le j \le k} \widehat{P}(s_j|s_{j-1})$$

Where, $\widehat{P} = -\log P$. Therefore, if the approximation is reasonable, the most likely message s_0 is the one minimizing $\widehat{P}(s_0, s_k)$.

Normally, a transduction stage is modeled by a finite-state device. For example a hidden Markov models (HMMs). Some of these finite-state devices are string-to-weight transducers, which are widely used at several stages of speech recognition. Phoneme lattices, language models, and word lattices are typically represented by such transducers. In fact, the transducers and algorithms we discussed in the previous sections apply to speech recognition. A speech recognizer is just a composition of transducers with different functions outputting weights, or both strings and weights.

Recall that a speech recognizer, namely the domain of the speech recognition systems above signal processing, can be composited by transducers as following [1,3,11]:

$$M \circ D \circ C \circ A \circ O$$

where language O represents the acoustic observation sequences, A is a transduction from acoustic observation sequences to context-dependent phoneme sequences, C the context-dependency model mapping sequences of context-dependent phonemes to context-

independent phones, D a pronunciation dictionary mapping phoneme sequences to word sequences, M is a weighted language specifying the language model (mapping sequences of words to sentences). M is also called an acceptor [28].

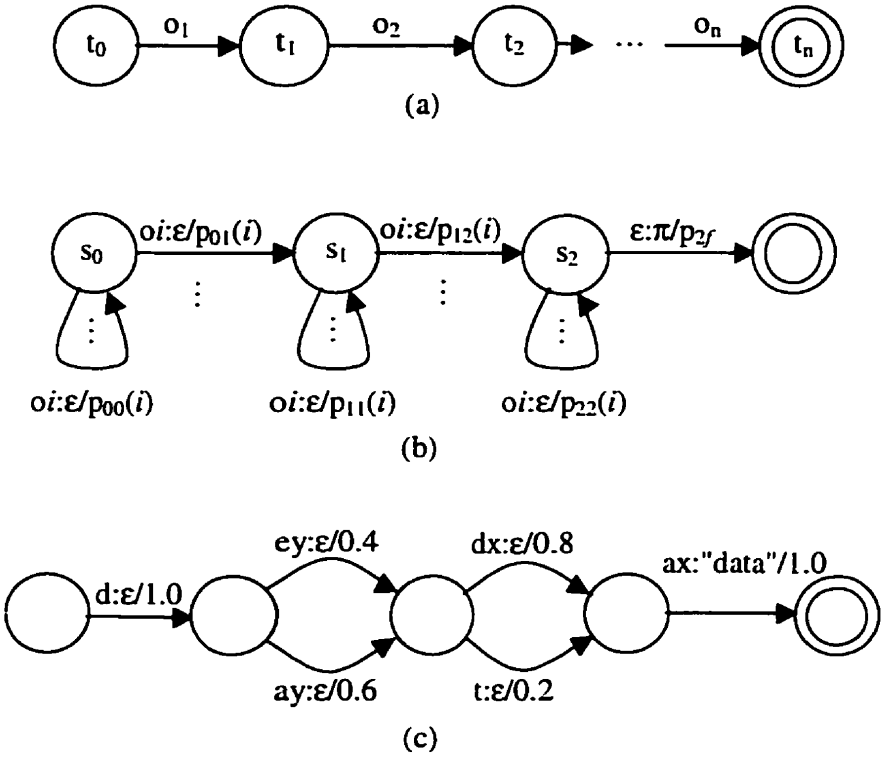


Figure 2.2 Models as Automata

The acoustic observation automaton O for a given utterance has the form shown on Figure 2.2(a). Each state represents a fixed point in time t_i, and each transition has a label, o_i, drawn from a finite alphabet that quantifies the acoustic signal between adjacent time points and is assigned probability 1.0.

The transducer A from acoustic observation sequences to phoneme sequences is built from phoneme models. A phoneme model is a transducer from sequences of acoustic observation labels to a specific phoneme that assigns to each acoustic observation sequence the likelihood that the specified phoneme produced it. Thus, different paths through a phoneme model correspond to different acoustic realizations of the phoneme.

Figure 2.2(b) shows a common topology for phoneme models. A is then defined as the closure of the sum of the phoneme models.

The transducer D from phoneme sequences to word sequences is built similarly to A. A word model is a transducer from phoneme sequences to the specified word that assigns to each phoneme sequence the likelihood that the specified word produced it. Thus different paths through a word model correspond to different phonetic realizations of the word. Figure 2.2(c) shows a typical topology for a word model. D is then defined as the closure of the sum of the word models.

So far, our recognizer has only used context-independent phoneme models. In other words, the likelihood assigned by a phoneme model in A is assumed conditionally independent of neighboring phonemes. Similarly, the pronunciation of each word in D is assumed independent of neighboring words. Therefore, each of the transducers has a particular simple form, that of the closure of the sum of (inverse) substitutions. That is, each symbol in a string on the output side replaces a language on the input side. This replacement of a symbol from one alphabet (for example, a word) by the automaton that represents its substituted language from a over a finer-grained alphabet (for example, phonemes) is the usual stage-combination operation for speech recognizers.

However, it has been shown that context-dependent phoneme models, which model a phoneme in the context of its adjacent phonemes, provide substantial improvements in recognition accuracy compared to context-independent phoneme models. Further, the pronunciation of a word will be affected by its neighboring words, inducing context dependencies across word boundaries.

Some strategies have been tried to solve the context-dependency problem, such as triphone models. Among these strategies the best is by interposing a new transducer C between A and D that convert between context-dependent and context-independent units. Of course, the size of originally constructed context-dependency transducer is large. Fortunately, transducer determinization and minimization techniques can be used to make context-dependency transducers as compact as possible.

Finally, the acceptor M encodes the language model, for instance an n-gram model. Combining those automata, we obtain $\pi_2(O \circ A \circ D \circ C \circ M)$, which assigns a

probability to each word sequences. The highest-probability path through that automaton estimates the most likely word sequence for the given utterance.

In general, considering space limitation and size of these automata, this cascade of compositions cannot be explicitly expanded. We need an approximation method to search it. Very often a beam pruning is used: only paths with weights within the beam (the difference of the weights from the minimum weights so far is less than a certain predefined threshold) are kept during the expansion of the cascade of composition. Furthermore, one is only interested in the best path or a set of paths of the cascade of transducers with the lowest weights.

A set of paths with the lowest weights can be represented by an acyclic string-to-weight transducer. Each path of that transducer corresponds to a sentence. The weight of the path can be interpreted as a negative *log* of the probability of that sentence given the sequence of acoustic observations (utterance). Such acyclic string-to-weight transducers are called word lattices. Of course, these acyclic transducers can be efficiently determinized and minimized because the sequential property with them.

Speech recognition is a wide research area of Computer Science, it is very exciting and challenging. The above is just a glimpse at speech recognition system. More details can be found from the references listed at the end of this thesis, such as from the reference [27].

In this chapter, we have made a review on the results of the research on Finite State Machine and Finite State Transducers (string-to-string and string-to-weight transducers). These results are very helpful for us to carry on the research on the determinization of the string-to-string/weight transducers in the following chapters.

Chapter 3

String-to-String/Weight Transducers and the SSW_determinization Algorithm

Chapter 2 has introduced the string-to-string transducers and the string-to-weight transducers. The string-to-string transducers are widely used in language processing [2,3,7,24], such as the representation of very large dictionaries, the compilation of morphological and phonological rules and the syntax. Similarly, the string-to-weight transducers are found at several stages of speech recognition [3,7,11]. Phone lattices, language models, and word lattices are typically represented by the string-to-weight transducers. Weights in these graphs correspond to negative logarithms of probabilities. They are added along a path. For a given string there might be many different paths in a transducer. The minimum of the total weights of these paths is only considered as a relevant information.

Automatic Speech Recognition is a new research area in Computer Science. Recently, much research has been carried on the string-to-weight or string-to-string/weight transducers because the domain of the speech recognition systems above signal processing can be represented by a composition of finite-state transducers outputting weights, or both strings and weights. In current Automatic Speech Recognition (ASR) system one important research topic is on the determinization of the string-to-string/weight transducers because some models used in the ASR systems can only be represented by string-to-string/weight transducers such as the HMMs (hidden Markov models) in our research.

This chapter gives a systematic introduction on the string-to-string/weight transducers. It includes the definitions and also a developed determinization algorithm used for the determinizable string-to-string/weight transducers.

3.1 String-to-String/Weight Transducers

The string-to-string/weight transducers are also called weighted string-to-string transducers. The definition of string-to-string/weight transducers is similar to the

definition of string-to-string transducers or string-to-weight transducers. The only difference is that the output of a string-to-string/weight transducer is a pair composed by a string and a weight.

3.1.1 General String-to-String/Weight Transducers

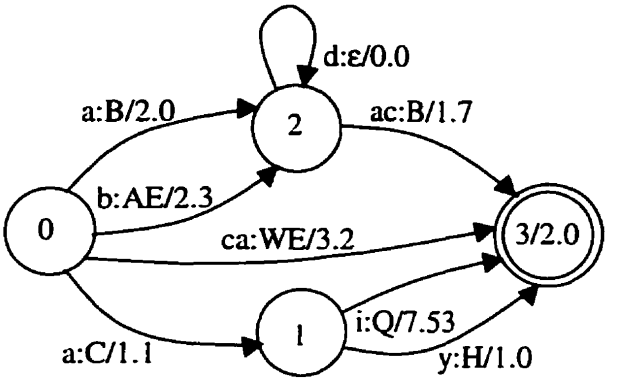


Figure 3.1 A typical string-to-string/weight transducer

Figure 3.1 is a typical string-to-string/weight transducer. A formal definition of the string-to-string/weight transducers is given as the following.

A string-to-string/weight transducer T is defined by $T = (Q, \Sigma, \Delta, I, F, E, \lambda, \rho)$, where :

- Q is a finite set of states
- Σ and Δ, finite sets corresponding respectively to the input and output alphabets of the transducer
- $I \subseteq Q$ is the set of initial states
- $F \subseteq Q$, the set of final states
- $E \subseteq Q \times \Sigma \times \Delta \times R_+ \times Q$ a finite set of transitions
- λ the initial weight function mapping I to R_+
- ρ the final weight function mapping F to R_+

the set E can be extended to include transitions $Q \times \Sigma^* \times \Delta^* \times R_+ \times Q$, where their input and output can be strings.

Without extension of E, this definition defines the string-to-string/weight transducers used in our ASR research. Each arc of these transducers has a feature that its input and output are symbols like in Figure 3.2. The symbol refers to a string with a length equals to 1 or 0 (an empty string ε).

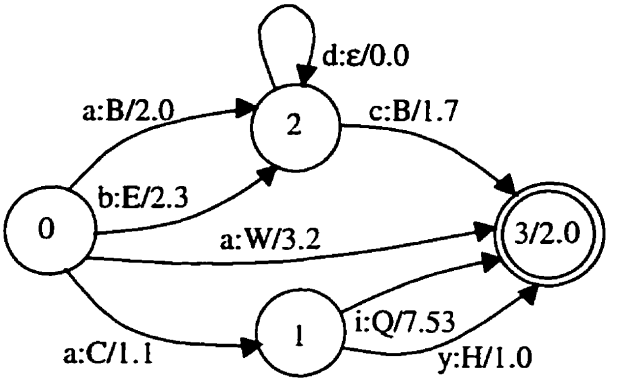


Figure 3.2 A string-to-string/weight transducer used in our ASR research

3.1.2 Sequential String-to-String/Weight Transducers

As it is known that the sequential property of a transducer is desired in the Automatic Speech Recognition process. The sequential transducers described here are transducers with a deterministic input. At any state of such transducers, at most one outgoing arc is labeled with a given element of the alphabet. Figure 3.3 gives an example of a sequential string-to-string/weight transducer.

A string-to-string/weight sequential transducer $T = (Q, i, F, \Sigma, \Delta, \delta, \sigma, \lambda, \rho)$,

where :

- Q is the set of its states
- $i \in Q$ its initial state
- $F \subseteq Q$ the set of final states
- Σ and Δ, finite sets corresponding respectively to the input and output alphabets of the transducer
- δ the transition function mapping $Q \times \Sigma$ to Q
- σ the output function which maps $Q \times \Sigma$ to $\Delta \times R_+$

- $\lambda \in R_+$ the initial weight
- ρ the final weight function mapping F to R_+

the transition function δ can be extended as in the string case to map $Q \times \Sigma^*$ to Q, and the output function σ can also be extended to $Q \times \Sigma^*$ to $\Delta^* \times R_+$.

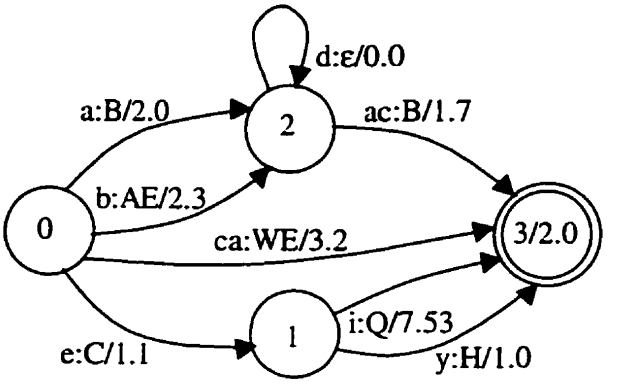


Figure 3.3 A sequential string-to-string/weight transducer

If the extensions of δ and σ are not allowed, the defined sequential string-to-string/weight transducers will have only symbols as input and output of their arcs like in Figure 3.4. This type sequential string-to-string/weight transducers are widely used in current ASR researches.

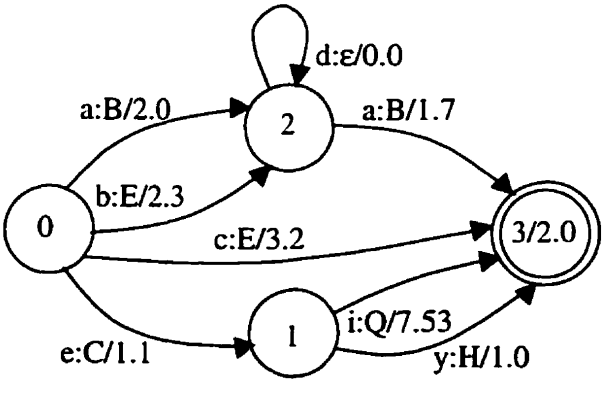


Figure 3.4 A sequential string-to-string/weight transducer used in our ASR research

Even though the sequential property is expected, not all string-to-string/weight transducers are sequential. In fact, in most cases the original transducer is not sequential. Therefore a determinization algorithm is needed. In the next section an algorithm used to determinize the determinizable string-to-string/weight transducers has been developed.

3.2 SSW_determinization Algorithm

Chapter 2 has introduced a general *PowerSeriesDeterminization* algorithm, which is applied to a tropical semiring such as a string-to-weight transducer. This algorithm can be also used to develop determinization algorithms for other semirings.

The semiring defined on $(\Sigma^* \cup \{\infty\}, \wedge, \bullet, \infty, \varepsilon)$ is called string semiring (here, ∞ a new element). The cross product of two semirings defines a semiring. The general algorithm also applies when the semiring is the cross product of $(\Sigma^* \cup \{\infty\}, \wedge, \bullet, \infty, \varepsilon)$ and $(R_+ \cup \{\infty\}, \min, +, \infty, 0)$. This allows us to determinize transducers outputting pairs of strings and weights - the string-to-string/weight transducers.

For the string-to-string/weight transducers, subsets in the algorithm are made of triples $(q, w, x) \in Q \times \Sigma^* \cup \{\infty\} \times R_+ \cup \{\infty\}$ where q is a state of the original transducer, w is a residual string and x is a residual weight. So, we have to consider the pair (w, x) as output in the determinization of string-to-string/weight transducers. Also, considering this *PowerSeriesDeterminization* algorithm is general some special cases have to be handled.

Based on the general algorithm and the features of the string-to-string/weight transducers a new determinization algorithm used for the string-to-string/weight transducers has been developed. This algorithm is named *SSW_determinization*.

Figure 3.5 gives the detailed pseudocode of *SSW_determinization* algorithm. This algorithm constructs a sequential string-to-string/weight transducer $T_2 = (Q_2, i_2, F_2, \Sigma, \Delta, \delta_2, \sigma_2, \lambda_2, \rho_2)$ equivalent to a given determinizable string-to-string/weight transducer $T_1 = (Q_1, \Sigma, \Delta, I_1, F_1, E_1, \lambda_1, \rho_1)$.

SSW_determinization(T_1, T_2)

1 $F_2 \leftarrow \varnothing$

2 $\lambda_2 \leftarrow \min_{i \in I_1} \lambda_1(i)$

3 $i_2 \leftarrow \bigcup_{i \in I_1} \{(i, \lambda_2^{-1} + \lambda_1(i))\}$

4 $Q \leftarrow \{i_2\}$

5 **while** $Q \neq \varnothing$

6 **do** $q_2 \leftarrow$ head[Q]

7 **if**($q \neq -2$ and for any $(q, w, x) \in q_2$)

8 **if**(there exists $(q, \varepsilon, x) \in q_2$ such that $q \in F_1$ or $q == -1$)

9 **then** $F_2 \leftarrow F_2 \cup \{q_2\}$

10 $\rho_2(q_2) \leftarrow \min_{q \in F_1 \cup -1,\ (q, \varepsilon, x) \in q_2,\ \rho_1(-1)=0} \quad x + \rho_1(q)$

11 **if**(there exists $(q, w, x) \in q_2$ such that $w \neq \varepsilon$, $q \in F_1$ or $q == -1$)

12 **then** $q_2' \leftarrow \bigcup_{(q, w, x) \in q_2,\ q \in F_1 \cup -1,\ \rho_1(-1)=0,\ w \neq \varepsilon} (q, w, x + \rho_1(q))$

13 $w' \leftarrow \bigcup_{(q, w, x) \in q_2'} firstSymbol(w)$

14 **for** each symbol $s \in w'$

15 **do** $\sigma_2(q_2, \varepsilon) \leftarrow (s, \min_{(q, w, x) \in q_2',\ firstSymbol(w) = s} [x + \rho_1(q)])$

16 $\delta_2(q_2, \varepsilon) \leftarrow \bigcup_{(q, w, x) \in q_2',\ firstSymbol(w) = s} (-1, w \bullet (\sigma_2(q_2, \varepsilon)|w)^{-1},$

17 $(\sigma_2(q_2, \varepsilon)|x)^{-1} + x + \rho_1(q))$

18 **if**($\delta_2(q_2, \varepsilon)$ is a new state)

19 **then ENQUEUE**(Q, $\delta_2(q_2, \varepsilon)$)

20 **for** each a such that $\Gamma(q_2, a) \neq \varnothing$

21 **do** $\sigma_2'(q_2, a) \leftarrow (\bigwedge_{(q, w, x) \in \Gamma(q_2, a)} [w \bullet (\sigma_1(t)|w)],$

$\min_{(q, w, x) \in \Gamma(q_2, a)} [x + \min_{t=(q, a, \sigma_1(t), n_1(t)) \in E_1} \sigma_1(t)|x])$

22 $\delta_2'(q_2, a) \leftarrow \bigcup_{q' \in \nu(q_2, a)} \{(q', w \bullet (\sigma_1(t)|w) \bullet (\sigma_2'(q_2, a)|w)^{-1},$

$\min_{(q, w, x, t) \in \chi(q_2, a),\ w = w_0,\ \sigma_1(t)|_w = w_1,\ n_1(t) = q'} [(\sigma_2'(q_2, a)|x)^{-1} + x$

$+ \sigma_1(t)|x])\}$

```
23              if(σ2'(q2, a)|w is not a symbol and also not a empty string)
24                w'' ← σ2'(q2, a)|w
25                σ2(q2, a) ← (firstSymbol(w''), σ2'(q2, a)|x)
26                w'' ← removeFirstSymbol(w'')
27                δ2(q2, a) ← (-2, w'', 0) ∪ δ2'(q2, a)
28                if(δ2(q2, a) is a new state)
29                  then ENQUEUE(Q, δ2(q2, a))
30              else σ2(q2, a) ← σ2'(q2, a)
31                   δ2(q2, a) ← δ2'(q2, a)
32                   if (δ2(q2, a) is a new state)
33                     then ENQUEUE(Q, δ2(q2, a))
34        else if(q == -2)
35          σ2(q2, ε) ← (firstSymbol(w''), 0)
36          w''← removeFirstSymbol(w'')
37          if(w'' == ε)
38            then δ2(q2, ε) ← δ'2(q2, a)
39          else δ2(q2, ε) ← (-2, w'', 0) ∪ δ'2(q2, a)
40          if(δ2(q2, ε) is a new state)
41            then ENQUEUE(Q, δ2(q2, ε))
42      DEQUEUE(Q)
```

Figure 3.5 Algorithm for the determinization of a string-to-string/weight transducer T_1 defined on the semiring $(\Sigma \cup \{\infty\}, \wedge, \bullet, \infty, \varepsilon) \times (R_+ \cup \{\infty\}, \min, +, \infty, 0)$.

This algorithm considers two basic requirements on the resulting transducer T_2. First, the original transducer T_1 used in our ASR research has its each arc with a format *symbol:symbol/weight*. Second, each final state of the original transducer T_1 has only accepting weight (or output weight). These two characteristics are kept in the resulting transducer T_2.

In this algorithm, $n_1(t)$ is defined as the destination state of a transition $t \in E_1$. Hence, $n_1(t) = q'$, if $t = (q, a, w, x, q') \in E_1$. The sets $\Gamma(q_2, a)$, $\gamma(q_2, a)$ and $\nu(q_2, a)$ used in the algorithm are defined by:

$$\Gamma(q_2, a) = \{(q, w, x) \in q_2 : \exists t = (q, a, \sigma_1(t), n_1(t)) \in E_1\}$$

$$\gamma(q_2, a) = \{(q, w, x, t) \in q_2 \times E_1 : t = (q, a, \sigma_1(t), n_1(t)) \in E_1\}$$

$$\nu(q_2, a) = \{(q' : \exists (q, w, x) \in q_2 : \exists t = (q, a, \sigma_1(t), q') \in E_1\}$$

$\Gamma(q_2, a)$ denotes the set of triples (q, w, x), elements of the subset q_2, having transitions labeled with the input a. $\gamma(q_2, a)$ denotes the set of quadruples (q, w, x, t) where (q, w, x) is a triple in q_2 such that q admits a transition with input label a . $\nu(q_2, a)$ is the set of states q' that can be reached by transitions labeled with a from the states of the subset q_2.

Notice that the state number q could take value *-1* or *-2* (for example in Line 7 and Line 8). We know that q is state number of the old transducer, q can never be a negative value. In fact, these states with a negative state number are generated and used to handle the special cases met during the determinization process. For easy understanding on this algorithm a detailed introduction is as follows.

1. Line 1 refers that the initial final state set of T_2 is empty.

2. Line 2 and Line 3 indicate that the initial weight λ_2 of T_2 is the minimum of all the initial weights of T_1. The initial state i_2 is a subset made of pairs (i, x), where i is an initial state of T_1, and $x = \lambda_1(i) - \lambda_2$. Fortunately, each transducer used in our automatic speech recognition research has only one initial state. This makes an easy implementation of line 2 and line 3. The next step is to put this initial state i_2 into an empty queue Q. Here, Q is used to maintain the set of subset q_2 not yet be extended (or determinized). Each subset in Q corresponds to one state of Q_2 for the new transducer T_2. Initially, Q contains only the subset i_2 (line 4).

3. F_2 is the set of the final states of the sequential transducer T_2. q_2 represents a final state iff it contains at least one triple (q, ε, x), where q is a final state of T_1 or equals to *-1* (see line 8 - 9), ε refers to an empty residual string, x is residual weight. This type

triple is named *final triple*. The final output weight associated to q_2 is then the minimum output weight of all the final triples in q_2 (line 10).

4. When q equals to -1 refers a final state without outgoing arcs. This state does not exist. It is designed and assumed to be one special state of the old transducer during the determinization.

Line 11 meets the **special case 1**. In this case subset q_2 contains triple (q, w, x) such that $w \neq \varepsilon$, $q \in F_1$ or q equals -1. This type triple is named *sub-final triple*. When *sub-final triple* appears in q_2 means that a final state with accepting output (w, x) has been reached. According to the basic requirements if each final state of the original transducer only has accepting weight the resulting transducer's final states cannot have accepting output composed by string and weight. Accepting output of any final state of the resulting transducer has to be a weight.

How can this be handled? Firstly a new subset q_2' has to be constructed with the triples (q, w, x) such that $w \neq \varepsilon$, $q \in F_1$ or q equals -1 in q_2. This q_2' is considered as part of q_2. Next is to construct the set of outgoing output symbols w' from the set of w in q_2' (line 12, line 13). Line 14 to line 19 are used to finish the expanding based on w'. Each output symbol s in w' corresponds to an outgoing arc from q_2, the input symbol of this arc is ε, the weight is the minimum weight of all sub-final triples with same s in q_2' (line 14, line 15). The destination of this arc is a subset $\delta_2(q_2, \varepsilon)$ formed by triples in q_2' with same s. $\delta_2(q_2, \varepsilon)$ contains triples that q equals -1. The residual string of each triple in $\delta_2(q_2, \varepsilon)$ is the string by removing s from the residual string w of the corresponding triple in q_2'. The weight of each triple in $\delta_2(q_2, \varepsilon)$ is the result of that the sum of the corresponding triple's residual weight x and its extra accepting cost in q_2' (if q equals -1 its accepting cost is zero) minuses the output weight of the newly constructed outgoing arc. If $\delta_2(q_2, \varepsilon)$ is a new subset then put it into queue Q (line 18, line 19). Continue this loop until each symbol in w' has been checked.

Therefore, q equals -1 means that q is considered as a special final state in the original transducer without outgoing arc and with a final weight zero. It is special because that state -1 does not exist in Q_1. State -1 is needed during the determinization when subset q_2 has at least one *sub-final triple*.

5. For each input symbol a such that there exists at least one state q of the subset q_2 admitting an outgoing transition labeled with a, one temporary outgoing transition leaving q_2 with the input symbol a is constructed (line 20 – line21). The output $\sigma_2'(q_2, a)$ of this temporary transition is composed by two parts. One is residual string which is the largest common prefix of the output strings of all the transitions with input symbol a that leave a state in the subset q_2, when concatenated at the end with the residual string associated to that state. Another part is residual weight which is the minimum of the output weights of all the transitions with input symbol a that leave a state in the subset q_2, when combined with the residual weight associated to that state.

The temporary destination state $\delta_2'(q_2, a)$ of the transition leaving q_2 is a subset made of triples (q', w', x'), where q' is a state of T_1 that can be reached by a transition labeled with a, w' is the corresponding residual string, x' is the corresponding residual weight. It is possible that $\delta_2'(q_2, a)$ has triples with same q' but different residual strings. Each residual string w' is constructed by removing the output string $\sigma_2'(q_2, a)|_w$ from the head of a string which is a concatenation of two strings. One of these two strings is the residual string w of the corresponding triple (q, w, x) in subset q_2, where q' can be reached from q by the transition $\sigma_1(t)$ with input symbol a. Another string is just the output string $\sigma_1(t)|_w$. x is computed by taking the minimum of output weights of all the transitions with input symbol a that leave a state q in the subset q_2 and reach the same state q' with the same residual string w', when combined with the residual weight of q minus the output weight $\sigma_2'(q_2, a)|_x$.

6. If $\sigma_2'(q_2, a)|_w$ is a symbol or an empty string this temporary transition $\sigma_2'(q_2, a)$ is a transition leaving q_2 with the input symbol a named $\sigma_2(q_2, a)$, and the destination state for this transition is $\delta_2'(q_2, a)$ named $\delta_2(q_2, a)$ (line 30 – 31). If $\sigma_2(q_2, a)$ is a new state then put it into queue Q.

Otherwise, if $\sigma_2'(q_2, a)|_w$ is a string with a length larger than 1 this temporary transition $\sigma_2'(q_2, a)$ has to be handled as a special case because of the basic requirements. This special case is defined as **special case 2.**

Review that according to the basic requirements during the determinization if the output of an outgoing arc A of a new state is string (at least two symbols) A has to be

transferred to *symbol:symbol/weight* format as same as the arc format of the original transducer. In *SSW_determinization* this special case is handled by the following:

(1) Assign the string part $\sigma_2'(q_2, a)|_w$ to w''(line 24).

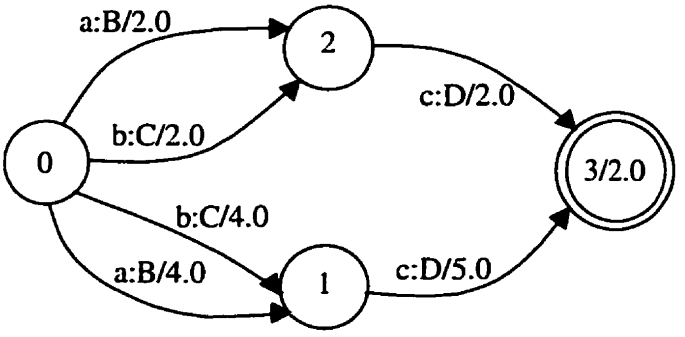
(2) Make a transition from q_2 with input symbol a, output symbol is the first symbol of w'', and its output weight is $\sigma_2'(q_2, a)|_x$ (line 25). Then, remove the first symbol of w''.

(3) The destination state $\delta_2(q_2, a)$ in line 27 is an union of a new triple and $\delta_2'(q_2, a)$. The new triple (*-2*, w'', *0*), where *-2* is a special value of q' not existed in Q_1, it means that the state $\delta_2(q_2, a)$ is an expanding state of the string $\sigma_2'(q_2, a)|_w$, w'' is the residual string, and the residual weight is zero. Here, temporary state $\delta_2'(q_2, a)$ is used to refer that the final expanding state of this string is $\delta_2'(q_2, a)$. If state $\delta_2(q_2, a)$ is a new state, then enqueue it into queue Q (line 28, 29).

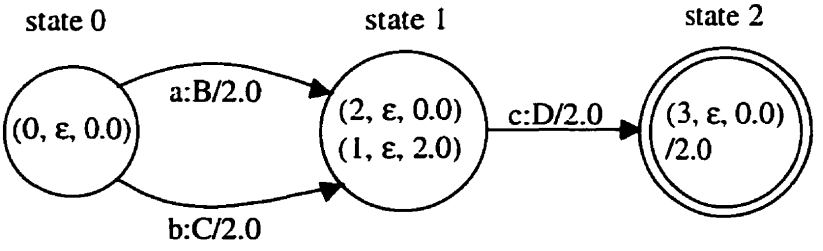
7. If the special case 2 described in the last paragraph has happened. Eventually a subset (*-2*, w'', *0*) $\cup$ $\delta'_2(q_2, a)$ will be assigned to q_2. Line 34 meets this condition. In line 35 a transition is made from q_2 with input symbol ε, output symbol is the first symbol of w'', and its output weight is zero. Then, remove the first symbol of w''. If w'' is an empty string assign the state $\delta_2'(q_2, a)$ to $\delta_2(q_2, a)$. Otherwise, construct a destination state $\delta_2(q_2, a)$ by the union of a new triple (*-2* as q', w'' as residual string, zero as residual weight) and the state $\delta_2'(q_2, a)$. Next step is to check whether state $\delta_2(q_2, a)$ is a new state. If $\delta_2(q_2, a)$ is a new state then put it into queue Q.

8. In line 42 the first element of Q is removed, and the algorithm returns to line 5. The algorithm will continue until the Q is an empty queue. Finally when this algorithm is terminated a sequential string-to-string/weigh transducer T_2 is returned with the same functions of the non-sequential transducer T_1.

9. The examples of this algorithm are shown in Figure 3.6 and Figure 3.7. Notice that an input string *ac* admits several outputs in T_1 of Figure 3.6: {(BD, 6), (BD, 11)}. Only one of these outputs ((BD, 6), with the smallest output weight) is kept in the resulting sequential transducer T_2 since it is only interested in the output with the minimum output weight for any given string.



(a)



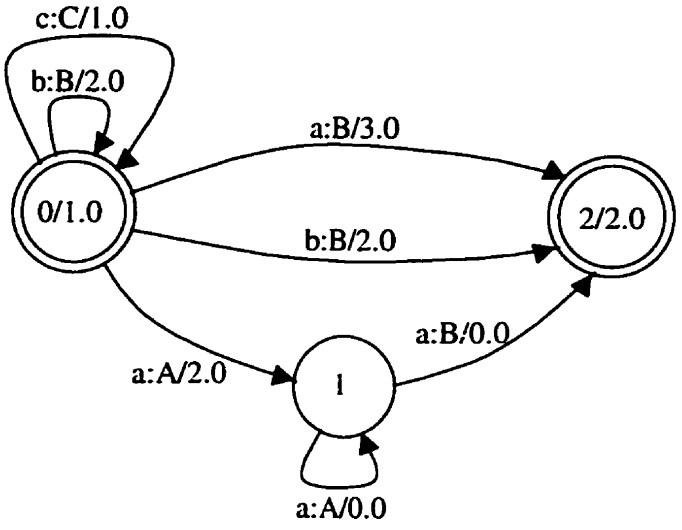
(b)

Figure 3.6 (a) a non-sequential string-to-string/weight transducer T_1 (b) a sequential string-to-string/weight transducer T_2 obtained from the SSW_determinization of T_1

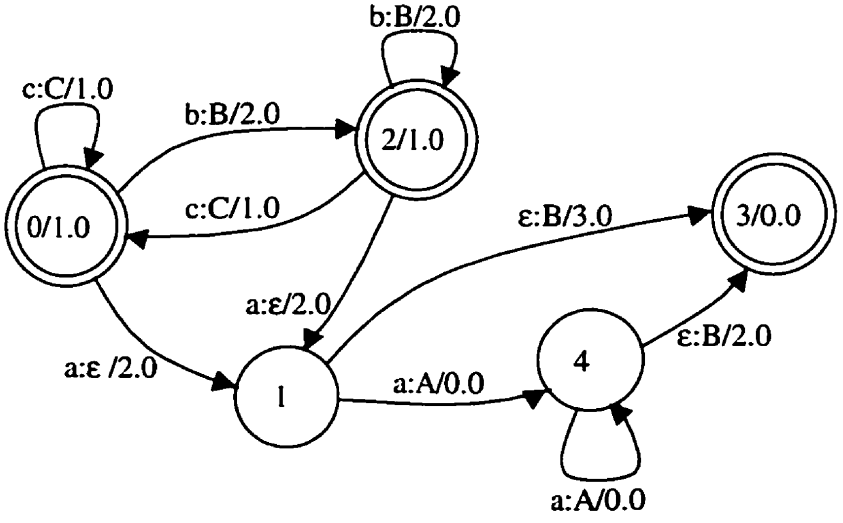
In Figure 3.6, after the determinization both state number and arc number are reduced. However, in some special case the state number or both state number and arc number is/are increased after the determinization such as in Figure 3.7. This is also a successful determinization result because the resulting transducer is sequential. The sequential transducer will dramatically increase the searching speed when it is applied to the ASR process instead of the equivalent non-sequential one [3].

Notice that several transitions might reach the same state with a same residual string but a priori different residual weights. Since only the best path is interested, namely the path corresponding to the minimum weight, the algorithm keeps only the minimum of

these weights for a given state element of a subset (line 22). This case can be observed in the final determinization step of T_1 in Figure 3.6.



(a)



(b)

Figure 3.7 (a) a non sequential string-to-string/weight transducer T_1 (b) sequential transducer T_2 obtained from the determinization of T_1

3.3 A Proof of the SSW_determinization Algorithm

It is very important to prove that if the determinization algorithm terminates then the resulting sequential transducer T_2 is equivalent to T_1.

Here, we emphasis the termination of the determinization algorithm is because that there are transducers with which determinization does not halt. It then generates an infinite number of subsets. We define determinizable transducers as those transducers with which the algorithm terminates. The details about the non-determinizable transducers will be discussed in Chapter 4.

Assume that the determinization algorithm terminates, then the resulting transducer T_2 is equivalent to T_1. The following is the proof.

We denote by $\theta_1(q, w, q', w')$ the output with the minimum weight of all paths from q to q' with a same output string w'. By construction we have:

$$\lambda_2 = \min_{i_1 \in I_1} \lambda_1(i_1)$$

We define the residual output string $s(q, w)$ and the residual output weight associated to q in the subset $\delta_2(i_2, w)$ as the weight $c(q, w)$ associated to the triple containing q in $\delta_2(i_2, w)$. By induction on $|w|$ we show that the subsets constructed by the algorithm are the sets $\delta_2(i_2, w)$, $w \in \Sigma^*$, such that:

$$\forall w \in \Sigma^*,\ \delta_2(i_2, w) = \bigcup_{q \in \delta_1(I_1, w)} \{ (q, s(q, w), c(q, w) \} \qquad (1)$$

$$c(q, w) = \min_{i_1 \in I_1} \left(\lambda_1(i_1) + \theta_1(i_1, w, q, w') \right) - \sigma_2(i_2, w)|_x - \lambda_2$$

$$\sigma_2(i_2, w) = \left(w' \bullet s(q, w)^{-1},\ \min_{q \in \delta_1(I_1, w)} \left(\lambda_1(i_1) + \theta_1(i_1, w, q, w',) \right) - \lambda_2 \right)$$

A pair $(q, s(q, w))$ belongs at most to one triple of a subset since for all paths reaching q with a same residual string $s(q, w)$, only the minimum of the residual output weight is kept. If $s(q, w)$ equals to an empty string the output string of $\sigma_2(i_2, w)$ is w'. Notice also that, by definition of **min**, in any subset there exists at least one state q with a residual output weight $c(q, w)$ equal to 0.

A string w is accepted by T_1 iff there exists $q \in F_1$ such that $q \in \delta_1(I_1, w)$. And its output string is w'. Using the equation (1), it is accepted iff $\delta_2(I_2, w)$ contains a triple $(q,$

$s(q, w)$, $c(q, w)$) with $q \in F_1$ and $s(q, w)$ equals to an empty string. This is exactly the definition of the final states F_2 (line 8). So T_1 and T_2 accept the same set of strings. Therefore, T_2 and T_1 are equivalent.

3.4 Space and Complexity of the SSW_determinization Algorithm

Both space and time complexity of the determinization algorithm for the determinizable string-to-string/weight transducers are exponential to the size of the original transducer T_1. However, in some cases in which the degree of non-determinism of the initial transducer is high, the determinization algorithm turns out to be fast because the resulting transducer has much less states than the initial one.

The proof on the space and time complexity of the *SSW_determinization* is same as that of the determinization of automata or string-to-weight transducers. It can be found in reference [14].

In this chapter, the definitions of the string-to-string/weight transducers have been addressed. And the *SSW_determinization* algorithm for the determinization of the determinizable string-to-string/weight transducers has been provided.

Since the *SSW_determinization* returns the whole resulting transducer a large amount of memory is needed for the determinization, especially when the size of the resulting transducer is large.

In fact, some non-determinizable string-to-string/weight transducers may be applied in the ASR systems. Therefore, it is also very important to reduce the non-determinism degree of the non-determinizable string-to-string/weight transducers. The transducers' determinizability and the determinization algorithm used for the non-determinizable string-to-string/weight transducers will be discussed in the next chapter.

Chapter 4

Determinizability and Partial Determinization of the String-to-String/Weight Transducers

In Chapter 3, it is mentioned that even a large set of transducers admits determinization there are transducers with which determinization does not halt. We define these transducers as non-determinizable transducer. On the other hand we define determinizable transducers as the transducers with which determinization terminates successfully. Because the determinization of non-determinizable transducers will generate an infinite number of subsets until the resources are used up it is always desired to predict whether a transducer is determinizable before the determinization process is carried on. Therefore to predict whether a transducer is determinizable before determinization is a useful topic in the determinization research area. This topic is called the determinizability test [8]. Many research results on this topic are available for the string-to-weight transducers. These research results have formed a group of theorems used to determine the determinizability of a string-to-weight transducer. These theorems are introduced in the first section of this chapter. Considering the similarity between string-to-string/weight transducers and string-to-weight transducers these theorems can also be directly applied to the string-to-string/weight transducers.

In this chapter another important topic is the partial determinization algorithm. If a transducer is non-determinizable this algorithm can be used to do partial determinization on it. This algorithm is named *PSSW_determinization*.

4.1 Theorems Used to Predict the Determinizability

Before using *SSW_determinization* to do determinization on a string-to-string/weight transducer T_1, it is always desired to know whether T_1 is determinizable because the non determinizable transducer will eventually take all memory resource by generating an infinite number of subsets.

The following theorems are the research results on the determinizability of the string-to-weight transducers [8]. These theorems can also be used for the string-to-string/weight transducers.

Theorem 4.1

Let $T_I = (Q_I, \Sigma, I_I, F_I, E_I, \lambda_I, \rho_I)$ be a string-to-weight transducer defined on the tropical semiring. If T_I has the twins property then it is determinizable.

Actually, the twins property is not a necessary condition for a transducer to be determinizable. Some transducers are determinizable even they do not have twins property. Unfortunately, it is complicated to specify the conditions for these transducers. However, in the case of trim unambiguous transducers, the twins property provides a characterization of determinizable transducers. See Theorem 4.2.

Theorem 4.2

Let $T_I = (Q_I, \Sigma, I_I, F_I, E_I, \lambda_I, \rho_I)$ be a trim unambiguous string-to-weight transducer defined on the tropical semiring. Then T_I is determinizable if and only if it has the twins property.

This theorem directly leads to Theorem 4.3, and the definition of an algorithm for testing the determinizability of trim unambiguous transducers.

Theorem 4.3

Let $T_I = (Q_I, \Sigma, I_I, F_I, E_I, \lambda_I, \rho_I)$ be a trim unambiguous string-to-weight transducer defined on the tropical semiring. There exists an algorithm to test the determinizability of T_I.

According to the theorem 4.3, testing the determinizability of T_I is equivalent to testing for the twins property. Mohri [8] gives an algorithm used to test the twins property of a transducer *TI*. This algorithm is close to that of Weber and Klemm for testing the sequentiability of string-to-string transducer. It is based on the construction of an automaton A = (Q, I, F, E) similar to the cross product of T_I with itself. This algorithm

takes polynomial time with respect to the size of A. The determinizability test algorithm used for an unambiguous trim transducer can be simply described as the following:

1. Compute the transitive closure of I: $T(I)$.
2. Determine the set of pairs (q_1, q_2) of $T(I)$ with distinct states $q_1 \neq q_2$.
3. For each such $\{q_1, q_2\}$ compute the transitive closure of $(q_1, q_2, 0)$ in A. If it contains (q_1, q_2, c) with $c \neq 0$, then T_I does not have the twins property.

This algorithm is very useful when we know that T_I is unambiguous. In many practical cases, the transducer one wishes to determinize is ambiguous. It is always possible to construct an unambiguous transducer T' from T. This will increase the time complexity of the determinizability test to be exponential in the worst case [8,17].

Because the definitions of twins property and trim unambiguous also can be applied to the string-to-string/weight transducers (See Chapter 2) these theorems can be used to the string-to-string/weight transducers directly. Of course, the testing determinizability algorithm can also be implemented by aware of the characteristic of the string-to-string/weight transducers.

Now, there are two ways to test the determinizability of a string-to-string/weight transducer. When a transducer is equivalent to a trim unambiguous transducer the testing determinizability algorithm introduced above can be used. Another way is to see whether the *SSW_determinization* on the transducer can terminates successfully. Once a string-to-string/weight transducer has been determined as a non-determinizable transducer a partial determinization algorithm developed in the following section can be applied to this transducer to reduce its non-determinism degree.

4.2 PSSW_determinization Algorithm

The purpose to determinize a transducer is to reduce its non-determinism degree because the non-determinism property will always slow down the searching speed in the Speech Processing. The best result is to get its equivalent sequential transducer. The complexity of the application of sequential transducers is linear in the size of the string to which it applies.

If a transducer is non-determinizable its equivalent sequential transducer does not exist. However, it is still possible to reduce its non-determinism degree. There are many

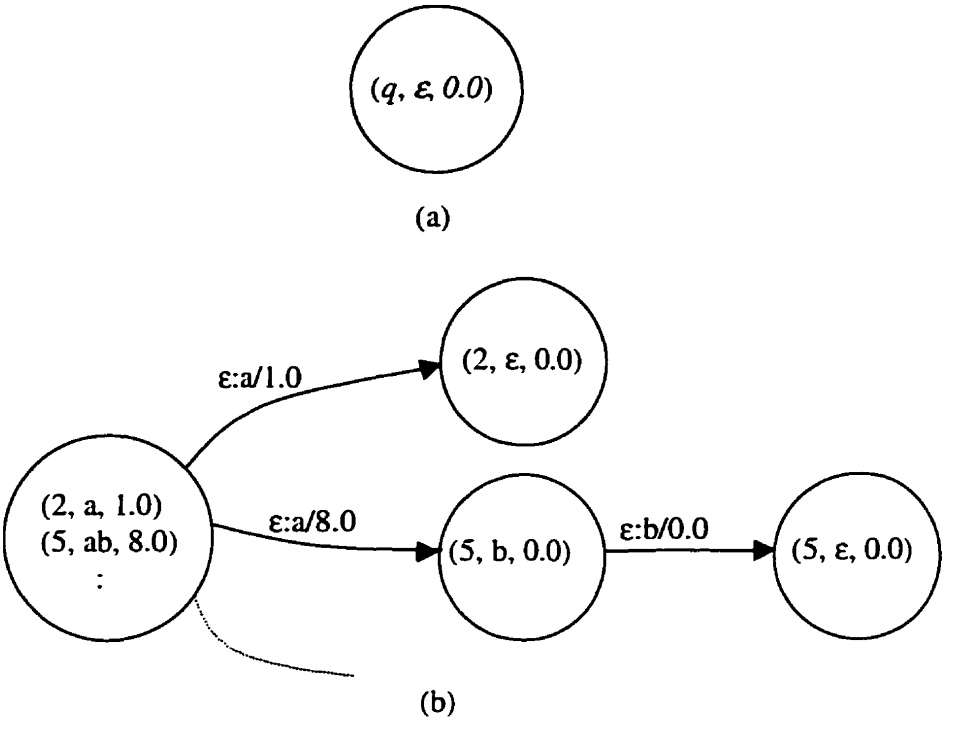


Figure 4.1 (a) a typical state can be directly determinized
(b) a subset with multiple triples

strategies can be used for this purpose. The basic idea is called local determinization [3,8], which the determinization is limited in several local areas to avoid getting in a non determinizable cycle.

Therefore, based on the *SSW_determinization* and the local determinization idea by changing the determinization strategy it may be possible to develop an algorithm for a non-determinizable transducer to reduce its non-determinism degree. The new ideas about this algorithm are diagramed in Figure 4.1.

Figure 4.1 (a) represents a subset has only one triple with an empty residual string and zero residual weight. Only this type subset is designed to be determinized by using the same way as in *SSW_determinization*. Figure 4.1 (b) represents a subset with multiple triples. This type subset has to be expanded into several subsets by using the strategy diagramed in Figure 4.1 (b). The resulting subsets have the same format as in Figure 4.1 (a). Therefore, only the subset with the format in Figure 4.1 (a) needs to be determinized. Because the number of this type subset less than $|Q_1|$ this algorithm will eventually

terminates. This algorithm is named *PSSW_determinization*, which is a partial determinization algorithm compared with *SSW_determinization*.

Figure 4.2 gives the detailed pseudocode of *PSSW_determinization* algorithm. This algorithm constructs a string-to-string/weight transducer T_2 equivalent to a given non-determinizable transducer $T_1 = (Q_1, \Sigma, \Delta, I_1, F_1, E_1, \lambda_1, \rho_1)$.

PSSW_determinization(T_1, T_2)

1 $F_2 \leftarrow \varnothing$

2 $\lambda_2 \leftarrow \min_{i \in I_1} \lambda_1(i)$

3 $i_2 \leftarrow \bigcup_{i \in I_1} \{(i, \lambda_2^{-1} + \lambda_1(i))\}$

4 $Q \leftarrow \{i_2\}$

5 **while** $Q \neq \varnothing$

6 **do** $q_2 \leftarrow$ head[Q]

7 **if**(there exists $(q, \varepsilon, x) \in q_2$ such that $q \in F_1$)

8 **then** $F_2 \leftarrow F_2 \cup \{q_2\}$

9 $\rho_2(q_2) \leftarrow \min_{q \in F_1, (q, \varepsilon, x) \in q_2} x + \rho_1(q)$

10 **if**(q_2 has only one triple with an empty residual string, (q, ε, x))

11 **for** each a such that $\Gamma(q_2, a) \neq \varnothing$

12 **do** $\sigma_2(q_2, a) \leftarrow (\bigwedge_{(q, \varepsilon, x) \in \Gamma(q_2, a)} [(\sigma_1(t)|_w)],$

$$\min_{(q, \varepsilon, x) \in \Gamma(q_2, a)} [x + \min_{t=(q, a, \sigma_1(t), n_1(t)) \in E_1} \sigma_1(t)|_x])$$

13 $\delta_2(q_2, a) \leftarrow \bigcup_{q' \in \nu(q_2, a)} \{(q', (\sigma_1(t)|_w) \bullet (\sigma_2(q_2, a)|_w)^{-1},$

$$\min_{(q, \varepsilon, x, t) \in \chi(q_2, a), w = w_0, \sigma_1(t)|_0 = w_1, n_1(t) = q'} [(\sigma_2(q_2, a)|_x)^{-1} + x + \sigma_1(t)|_x])\}$$

14 **if** ($\delta_2(q_2, a)$ is a new state)

15 **then ENQUEUE**(Q, $\delta_2(q_2, a)$)

```
16        else
17          for each triple (q, w, x) ∈ q2
18            if(w == ε)
19              σ2(q2, ε) ← (0, x)
20            else
21              σ2(q2, ε) ← (firstSymbol(w), x)
22              w← removeFirstSymbol(w)
23            δ2(q2, ε) ← (q, w, 0)
24            if(δ2(q2, ε) is a new state)
25              then ENQUEUE(Q, δ2(q2, ε))
26        DEQUEUE(Q)
```

Figure 4.2 Algorithm for the partial determinization of a non-determinizable string-to-string/weight transducer T_1 defined on the semiring $(\Sigma \cup \{\infty\}, \wedge, \bullet, \infty, \varepsilon) \times (R_+ \cup \{\infty\}, \min, +, \infty, 0)$.

Notice that the result of this algorithm is useful only when the original transducer T_1 is non determinizable. We do not use this *PSSW_determinization* when the transducer is determinizable.

Because this algorithm is based on the *SSW_determinization*, the transducer's basic requirements (see Chapter 3, page 32) have been kept. And the resulting transducer is an equivalent transducer that accepts and outputs same strings with the original transducer, but not sequential. However, the degree of non-determinism of the resulting transducer has been reduced, which is very important in the application of the Speech Processing.

Figure 4.3 is a typical non-determinizable string-to-string/weight transducer T_1. There is a cycle from its state *0* to state *1*. By using the theorems in section 4.1 the determinizability of T_1 can be predicted:

1. T_1 is trim. By definition (see Chapter 2, page 15), a transducer T is said to be trim if all states of T belong to a successful path. It is obvious that state *0* and state *1* belong to the successful path $0 \rightarrow 1$.

2. Also T_l is unambiguous. Because T_l has only two states and one of them is an accept state. It is very easy to prove that for any given string w there exists at most one successful path labeled with w, which means T_l is unambiguous (see Chapter 2, page 15).
3. T_l has no twins property. By the definition (see Chapter 2, page 16), two states q and q' are twins if, when they can be reached from the initial state by the same string u, the minimum outputs of loops at q and q' labeled with any string v are identical . T has twins property when any two states q and q' of T are twins. It is obvious that the transducer T_l in Figure 4.3 does not satisfy the requirements to have twins property.

Therefore, T_l is a trim unambiguous string-to-string/weight transducer without twins property. According to the theorem 4.2 in section 4.1 T_l is non-determinizable.

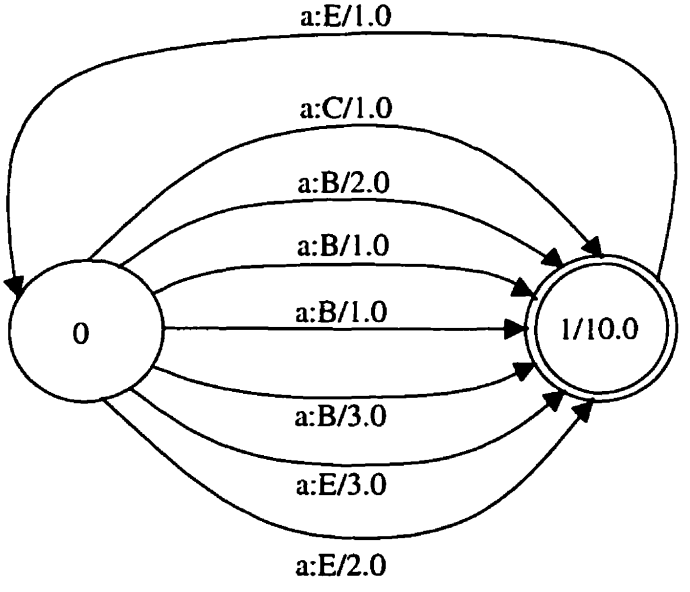


Figure 4.3 A typical non-determinizable string-to-string/weight transducer T_l

Figure 4.4 is the determinization of T_1 by *SSW_determinization*. It can be seen that the determinization will generate infinite number subsets, which also means that T_1 is non-determinizable.

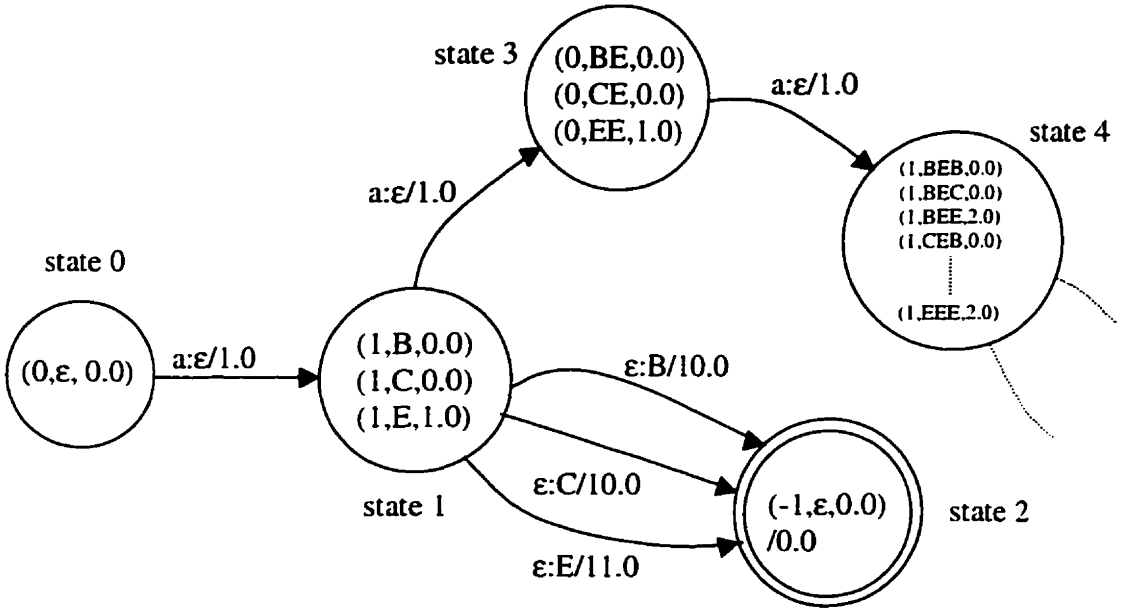


Figure 4.4 The determinization of T_1 by SSW_determinization

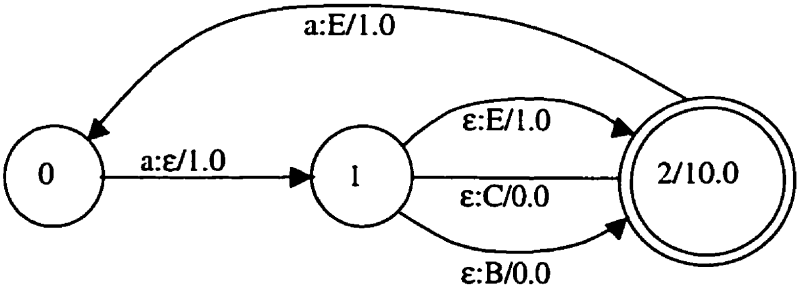


Figure 4.5 The partial determinization result of the transducer T_1 in Figure 4.3

Figure 4.5 is the result of the partial determinization on T_1. Compared with the original transducer the partial determinization is successful.

In this chapter, the determinizability of the string-to-string/weight transducers has been discussed. Based on the *SSW_determinization* and the local determinization idea a partial determinization algorithm used for the non-determinizable transducers has been developed. However, a weak point for this algorithm is that the resulting transducer is not sequential even though its non-determinism degree has been reduced compared with that of the original transducer. Besides this disadvantage the *PSSW_determinization* also suffers the same problem with the *SSW_determinization*, which the memory resource may be not enough for the determinization if the transducer is too large.

In the next chapter, an algorithm called determinization on the demand will be discussed. This algorithm can avoid the disadvantage of the *PSSW_determinization* and *SSW_determinization*. Such as it can be applied to construct the result of the determinization of any string-to-string/weight transducer T for a given input string w whenever the transducer T is determinizable or non-determinizable.

Chapter 5

Determinization of the String-to-String/Weight Transducers on the Demand

Determinization on the demand or determinization on the fly is a very important technique in Automatic Speech Recognition [8]. One disadvantage of the complete *SSW_determinization* is that the memory required may exceed the limitation of the resource when the size of the transducer is too large. Also, the partial determinization *PSSW_determinization* cannot satisfy the sequential requirement. Fortunately as it is known that if one wishes to construct the result of the determinization of a transducer T for a given input string w, one does not need to expand the whole result of the determinization, but only the necessary part of the determinized transducer. When restricted to a finite set the function realized by any transducer is sequential since it has bounded variation (see Chapter 2, page 13, Theorem 2.4.8). Therefore, it is always possible to expand the result of the determinization algorithm for finite set of input strings, even if T is not determinizable. Determinization on the demand is an algorithm used to determinize a state only when it is ordered. This algorithm can construct the result of the determinization of any string-to-string/weight transducer T for a given input string w.

In this chapter a determinization algorithm named *DSSW_determinization* is developed based on the determinization algorithm *SSW_determinization*. And, for easier understanding a detailed example analysis is provided to see the special features of this algorithm.

5.1 DSSW_determinization Algorithm

One characteristic of the determinization on the demand is that this algorithm is a dynamic determinization method. This means the determinization is always in the waiting state. It waits for a state to be ordered by the user. This ordered state is not determinized before it is ordered. Another characteristic is that the determinization on the demand only keeps necessary information related to the newly determinized state for the further

determinization. Hence, determinization on the demand will take the lowest memory compared with other determinization algoritms (see Chapter 7).

The determinization on the demand algorithm is named *DSSW_determinization*. It is based on the determinization algorithm *SSW_determinization*.

Figure 5.1 gives the detailed pseudocode of *DSSW_determinization* algorithm. This algorithm returns all determinized states one by one for the determinization of a transducer $T_1 = (Q_1, \Sigma, \Delta, \delta_1, I_1, F_1, E_1, \lambda_1, \rho_1)$ for a given input string.

DSSW_determinization(T_1, s)

1 **if** the ordered state number $s = 0$

2 $F_2 \leftarrow \varnothing$

3 $\lambda_2 \leftarrow \min_{i \in I_1} \lambda_1(i)$

4 $i_2 \leftarrow \bigcup_{i \in I_1} \{(i, \lambda_2^{-1} + \lambda_1(i))\}$

5 $q2 \leftarrow \{i_2\}$

6 **else** find the subset q_2 for the ordered state s from **HashTable**

7 **if** q_2 is not exist

8 **RETURN** "state s cannot be reached"

9 fresh **HashTable**

10 **if**($q \neq -2$ and for any $(q, w, x) \in q_2$)

11 **if**(there exists $(q, \varepsilon, x) \in q_2$ such that $q \in F_1$ or $q == -1$)

12 **then** $F_2 \leftarrow F_2 \cup \{q_2\}$

13 $\rho_2(q_2) \leftarrow \min_{q \in F_1 \cup -1, (q, \varepsilon, x) \in q_2, \rho_1(-1)=0} x + \rho_1(q)$

14 **if**(there exists $(q, w, x) \in q_2$ such that $w \neq \varepsilon$, $q \in F_1$ or $q == -1$)

15 **then** $q_2' \leftarrow \bigcup_{(q, w, x) \in q_2, q \in F_1 \cup -1, \rho_1(-1)=0, w \neq \varepsilon} (q, w, x + \rho_1(q))$

16 $w' \leftarrow \bigcup_{(q, w, x) \in q_2'} firstSymbol(w)$

17 **for** each symbol $s \in w'$

18 **do** $\sigma_2(q_2, \varepsilon) \leftarrow (s, \min_{(q, w, x) \in q_2', firstSymbol(w) = s} [x + \rho_1(q)])$

19 $\delta_2(q_2, \varepsilon) \leftarrow \bigcup_{(q,\, w,\, x) \in q_2',\ firstSymbol(w) = s} (-1,\ w \bullet (\sigma_2(q_2, \varepsilon)|_w)^{-1},$

20 $(\sigma_2(q_2, \varepsilon)|_x)^{-1} + x + \rho_1(q))$

21 **insert** $\delta_2(q_2, \varepsilon)$ into **HashTable**

22 **for** each a such that $\Gamma(q_2, a) \neq \varnothing$

23 **do** $\sigma_2'(q_2, a) \leftarrow (\bigwedge_{(q,\, w,\, x) \in \Gamma(q_2,\, a)} [w \bullet (\sigma_1(t)|_w)],$

$$\min_{(q,\, w,\, x) \in \Gamma(q_2,\, a)} [x + \min_{t = (q,\, a,\, \sigma_1(t),\, n_1(t)) \in E_1} \sigma_1(t)|_x])$$

24 $\delta_2'(q_2, a) \leftarrow \bigcup_{q' \in \nu(q_2,\, a)} \{(q',\ w \bullet (\sigma_1(t)|_w) \bullet (\sigma_2'(q_2, a)|_w)^{-1},$

$$\min_{(q,\, w,\, x,\, t) \in \chi(q_2,\, a),\ w = w_0,\ \sigma_1(t)|_w = w_1,\ n_1(t) = q'} [(\sigma_2'(q_2, a)|_x)^{-1} + x$$

$$+ \sigma_1(t)|_x])\}$$

25 **if**($\sigma_2'(q_2, a)|_w$ is not a symbol and also not a empty string)

26 $w'' \leftarrow \sigma_2'(q_2, a)|_w$

27 $\sigma_2(q_2, a) \leftarrow (firstSymbol(w''),\ \sigma_2'(q_2, a)|_x)$

28 $w'' \leftarrow removeFirstSymbol(w'')$

29 $\delta_2(q_2, a) \leftarrow (-2,\ w'',\ 0) \cup \delta_2'(q_2, a)$

30 **insert** $\delta_2(q_2, a)$ into HashTable

31 **else** $(q == -2)$

32 $\sigma_2(q_2, \varepsilon) \leftarrow (firstSymbol(w'),\ 0)$

33 $w'' \leftarrow removeFirstSymbol(w')$

34 **if**$(w'' == \varepsilon)$

35 **then** $\delta_2(q_2, \varepsilon) \leftarrow \delta'_2(q_2, a)$

36 **else** $\delta_2(q_2, \varepsilon) \leftarrow (-2,\ w'',\ 0) \cup \delta'_2(q_2, a)$

37 **insert** $\delta_2(q_2, \varepsilon)$ into HashTable

38 **RETURN** state s

Figure 5.1 Algorithm for the determinization on the demand of a string-to-string/weight transducer T_1 defined on the semiring $(\Sigma \cup \{\infty\}, \wedge, \bullet, \infty, \varepsilon) \times (R_+ \cup \{\infty\}, \min, +, \infty, 0)$.

Most lines of this algorithm are same as the *SSW_determinization*. The new characteristics are described as the follows:

1. The first ordered state is considered as state zero. In this algorithm the first ordered state is supposed to be the initial state of the resulting transducer T_2. Actually, this is not necessary. The determinization can be started from any state of the original transducer T_1. If the first ordered state is not the initial state of the resulting transducer T_2 one has to provide a state of the original transducer T_1, which is taken as the starting point of the determinization for a given string. In fact, the starting point of the determinization is always from the request of the initial state of the resulting transducer T_2.
2. All expanded subsets from subset q_2 are put into a hash table. Any expanded subset is considered as a new subset corresponding to a new state in the resulting transducer T_2 but not determinized yet. These subsets are kept and wait for the next ordered state. If the subset q_2 for the ordered state cannot be found in the hash table the algorithm will return an error message because that this means the ordered state cannot be reached from the last ordered state which has been determinized. If the subset is found. Then fresh the hash table waiting for the new expanded subsets during the determinization (line 9).
3. Because the hash table only keeps the expanded subsets from the latest determinized state, the memory used for the determinization on the demand is very low compared with *SSW_determinization* and *PSSW_determinization*.
4. In fact, it is also possible to introduce a buffer to this algorithm. The buffer can keep some determinized states. If the ordered state can be found from these states it will be returned immediately without computing it again. When the buffer size is large enough the whole transducer can be kept. If a buffer is implemented the hash table should keep all subsets related to the buffered determinized states (see Chapter 6).
5. The total number of the ordered state for an input string w equals to $|w|$.

5.2 An Example for Determinization on the Demand

A simple non-sequential string-to-string/weight transducer T_1 used for the example analysis of the *DSSW_determinization* algorithm is shown in Figure 5.2.

1. Suppose a user needs to pass through a sequential transducer of T_1 for a given string "*ac*". Since *DSSW_determinization* is used the first ordered state, which is the initial state of the sequential transducer is shown in Figure 5.3.

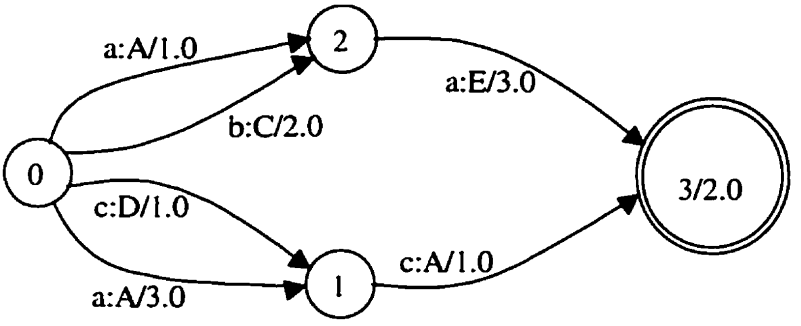


Figure 5.2 A string-to-string/weight transducer T_1 used for the example analysis

2. The user gets state zero with three output arcs as shown in Figure 5.3. The subsets 1, 2 and 3 are stored in a hash table.

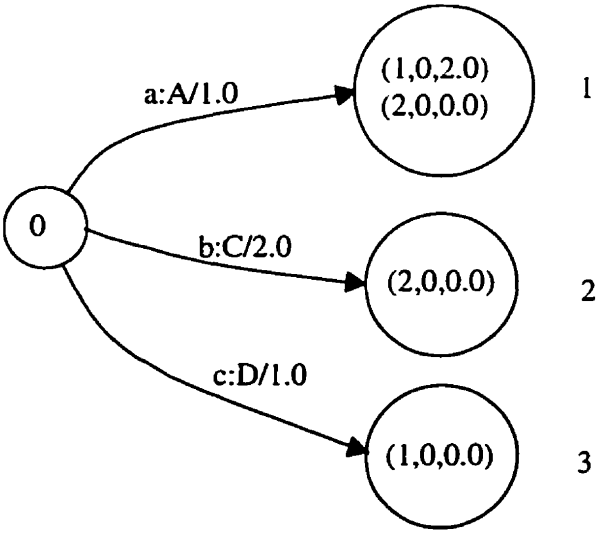


Figure 5.3 The initial state

3. Since the string is "*ac*" the first input symbol is "*a*". The user will choose state 1 as the next ordered state. Then the subset 1 is picked up from the hash table and the hash table is refresh to store the subsets when determinizing subset 1. The result is shown in Figure 5.4. The subset 4 is stored in the hash table.
4. Similarly, the next ordered state is state 4.

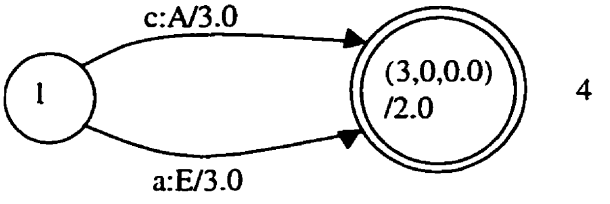
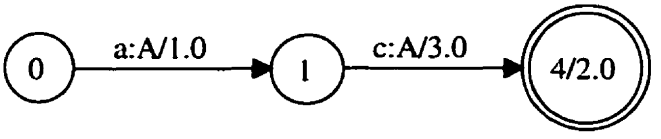
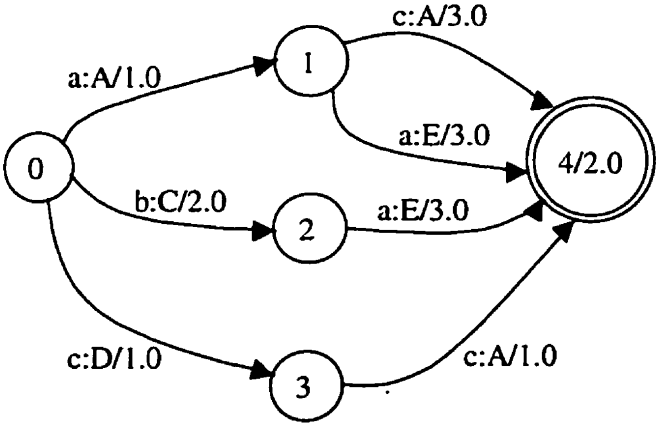


Figure 5.4 The second ordered state



(a)



(b)

Figure 5.5 (a) a sequential transducer of T_1 for a given string "ac"
(b) the whole sequential transducer of T_1

5. In step 4 the user needs to order state 4 because the input symbol is "c". There is no reason to choose other states. But, if for some mistake the user decides to order state 2 and if no buffer implemented in *DSSW_determinization.* The user will receive an error message "state 2 cannot be reached". This is because the subset 2 cannot be found in the hash table that has been cleaned up in step 3. If there is buffer implemented the user will possibly get the determinized state 2 (also see Chapter 6), which is only useful when the user try to get the whole sequential transducer of T_1. Therefore, when a string is given the user should order the state only according to this given string to avoid error or mistake.
6. When the determinization terminates from the view of the user the passed sequential transducer of T_1 for the given string "*ac*" is a transducer shown in Figure 5.5 (a), which is a part of the whole sequential transducer of T_1. See Figure 5.5.

So far, three determinization algorithms for the string-to-string/weight transducers have been developed. These three algorithms are similar. However, they have distinct features and can be used for different purposes. The most useful and most interesting one among them is the determinization on the demand algorithm described in this chapter because the low memory cost (also see Chapter 7), and also this algorithm can be applied to any string-to-string/weight transducer whenever the transducer is determinizable or non-determinizable,

The introduction on the implementation of these three determinization algorithms is followed in the next chapter.

Chapter 6

Implementation of the Determinization Algorithms

Three determinization algorithms are available for the determinization of the string-to-string/weight transducers. They are the complete determinization algorithm *SSW_determinization* used for the determinizable transducers, the partial determinization algorithm *PSSW_determinization* used for the non determinizable transducers, and the determinization on the demand algorithm *DSSW_determinization*. This chapter will discuss the implementation of these algorithms.

Automatic Speech Recognition is a big research area consisting of different parts [2,3,8]. For example the composition of the transducers is another important part besides the determinization. ASR systems are a rapidly developed research project where implementation needs to be frequently improved according to the latest research results. Therefore, it is very important for the implementation of the determinization to keep the consistency among different parts.

In this chapter the related implementation characteristics in our ASR research group are introduced first. Following this are the new data structures used for the determinization, such as the data structure for the subsets. The introduction on the implementation is focused on the determinization on demand algorithm because it includes the most implementation characteristics. Other special implementation characteristics of the *SSW_determinization* and the *PSSW_determinization* are also included.

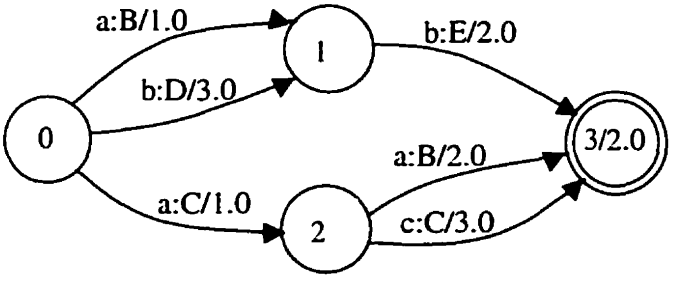
6.1 Related Implementation Characteristics in Our ASR Research Group

The following implementation characteristics are going to be kept in the implementation of the determinization algorithms.

1. The string-to-string/weight transducer is stored on disc as a text file in the format shown in Figure 6.1. The text file has to be transferred to a binary file before the transducer is loaded into the memory by using a method called *ReadFSM*. The software used for this transfer is *fsmcompile* from AT&T (available from WEB page [27]).

0	1	1	2	1.0
0	1	2	4	3.0
0	2	1	3	1.0
1	3	2	5	2.0
2	3	1	2	2.0
2	3	3	3	3.0
3	2.0			

(a)



(b)

Figure 6.1 A string-to-string/weight transducer.
(a) the text file, (b) the corresponding transducer

2. The design of the data structure for a string-to-string/weight finite state transducer is very important. This data structure should keep the memory as low as possible and includes all the information. The sketch diagram of the FST Data structure used in the ASR research is shown in Figure 6.2. The FST data structure can be divided into two parts. The first part has five fields including the necessary information for most important applications on the transducer such as the determinization and the composition. The second part is named *other fields* used to describe the transducer's characteristic and performance.

In the first part of the FST data structure the first field *arcs* is a pointer to an array of arc pointer that each element points to a *_fstarc* structure. The *_fstarc* structure is composed of five fields shown in Figure 6.2:

- *f* is the from state
- *t* is the destination state
- *i* refers to the input symbol
- *o* represents the output symbol
- *c* is the cost for this transition

Where, c is *float number*, others are *integers*. Arc pointer array is sorted by from state and input symbol.

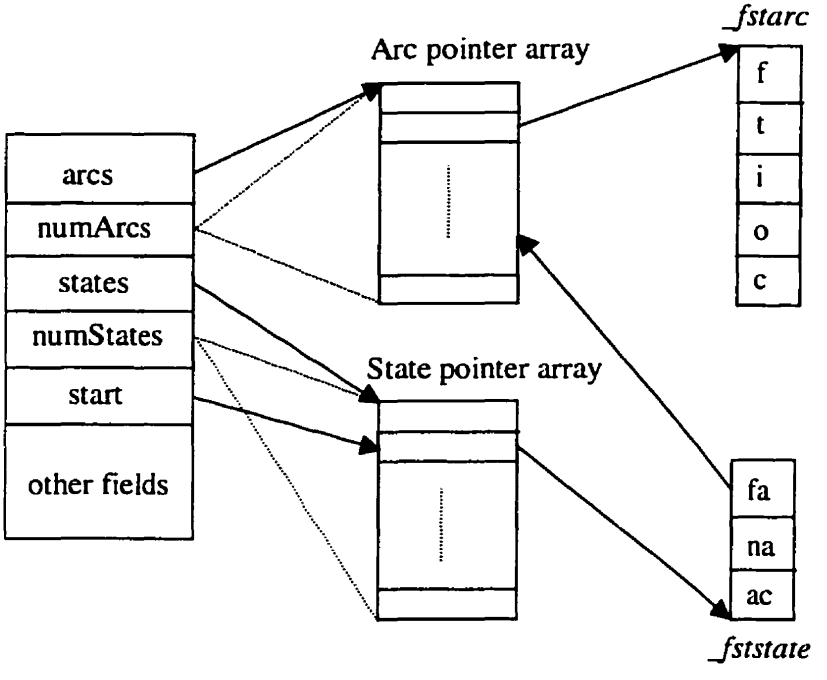


Figure 6.2 FST Data Structure

The second field *numArcs* is an integer that refers to the total number of the arcs. The third field *states* is a pointer to an array of state pointer that each element points to a *_fststate* data structure. The *_fststate* structure includes the following three fields:

- *ac* indicates the accept cost. If *ac* is less than a very large number the state is an accept state. Otherwise the state is a non-accept state. *ac* is a float number

- *na* is the number of arcs of the state. *na* is an integer
- *fa* is a pointer to one element of the arc pointer array. The pointed element points to the first arc of the state

The fourth field *numStates* is the total number of states in the FST. The fifth field *start* is also a pointer that points to one element of the state pointer array. This element points to the start state of the Finite State Transducer. Normally, this state is pointed by the first element of the state pointer array.

3. ANSI C is the programming language used in the ASR system.

6.2 New Data Structures Used in the Implementation

A new data type in the determinization algorithms is the subset. Be aware of the characteristics of the subsets three basic data structures are designed to represent them. See Figure 6.3.

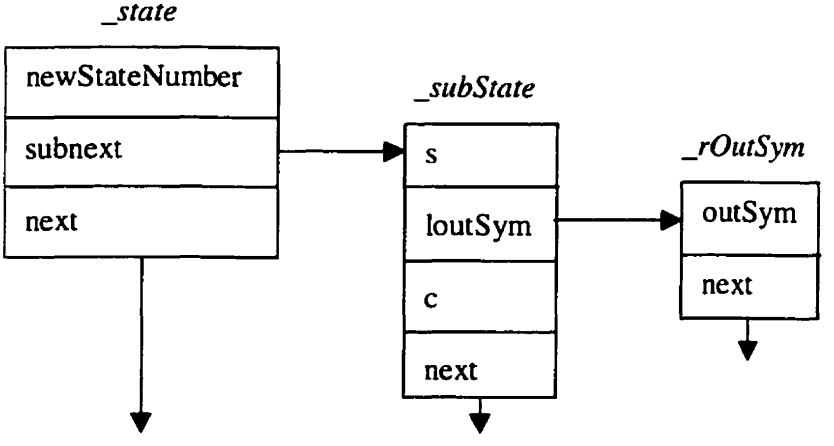


Figure 6.3 The data structure of the subsets

The *_state* data structure has three fields:

- *newStateNumber* is an integer. According to the algorithms each subset corresponds to one state in the resulting transducer T_2. The state number is assigned to *newStateNumber* during the determinization
- *subnext* is a pointer to a list of data structure called *_subState*
- *next* is a pointer to the next *_state*.

The *subState* data structure is used to describe the triple (q', w', x'). It includes four fields that can be described as the following:

- s is an integer. It is an old state number in the original transducer T_1. It represents q'
- *loutSym* is a pointer to a list of *_rOutSym* data structure. It is the residual string composed of zero or many symbols
- c is the residual weight. It represents x'
- *next* is a pointer to the next *_subState* structure

The *_rOutSym* data structure is used to describe the residual string w' in a triple (q', w', x'). It consists the following two elements:

- *outSym* is an integer used to represent a symbol in w'
- *next* is a pointer to the next *_rOutSym* structure. A list of *_rOutSym* structure represents a residual string w'

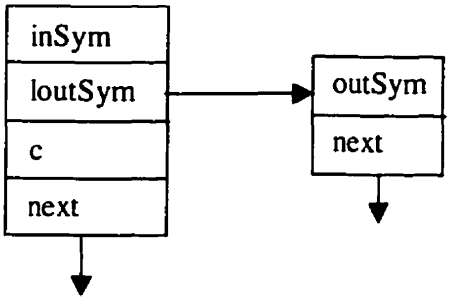


Figure 6.4 The _newArc data structure

Another new data type is the intermediate arc. The intermediate arc represents the intermediate (or temporary) transition σ_2' (see Chapter 3, page 31). When special case 2 happened the intermediate arc has the format *symbol:string/weight*. The *_newArc* data structure used to describe the intermediate arcs is shown in Figure 6.4. It includes the following elements:

- *inSym* is an integer that represents the input symbol
- *loutSym* is the output string. It is a pointer to a list of *_rOutSym* data structure

- *c* is the cost of the transition
- *next* is a pointer to the next *_newArc* structure

6.3 Implementation of the DSSW_determinization Algorithm

As it has been mentioned that the most interested deteriminization algorithm is the determinization on the demand. The implementation of this algorithm also has the most characteristics among three determinization algorithms.

In this section the implementation of the *DSSW_determinization* algorithm has been introduced by viewing its design of implementation, data flow and function call sequence. More information about the implementation is available in the Appendix, from the source files.

6.3.1 Module Design

For easier implementation and maintenance a module design strategy is used in the implementation. The following picture describes the module design of the program *DSSW_determinization.*

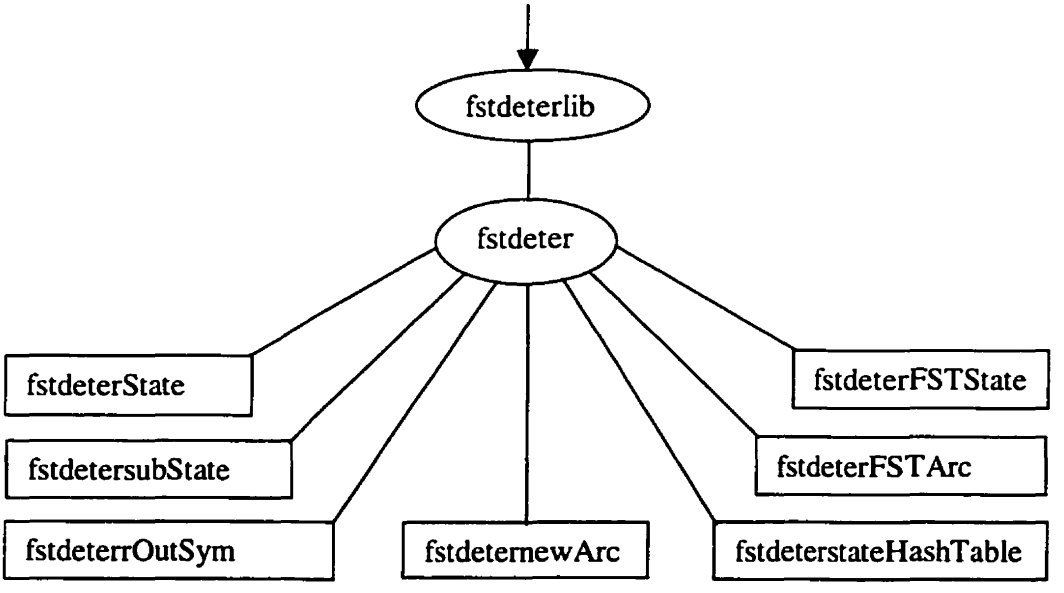


Figure 6.5 Module design of the SSW_determinization

1. **Module fstdeterlib**

It is designed to be an interface that contains all methods can be used directly by the user. The *fstdeterlib* includes the following methods:

- **InitFSTDeter**

  This method has to be called before the determinization. It initiates the hash table and the global variables used for the determinization.

- **GetFSTStartState**

  This method has to be called after the method *InitFSTDeter*. It finds out the start state(s) of the original transducer T_1, and then creates a subset representing the start state of the resulting transducer T_2. This subset will then be passed to module *fstdeter* for the further determinization processing. The method *GetFSTStartState* returns a pointer to the start state of the resulting transducer T_2. If an error happened it returns a NULL pointer.

The GetFSTStartState method attributes		
Argument	Type	Description
return value	FSTState	A pointer to a FST state
fst	FST	A pointer to a FST transducer

- **GetFSTStateByNum**

The GetFSTStateByNum method attributes		
Argument	Type	Description
return value	FSTState	A pointer to a FST state
fst	FST	A pointer to a FST transducer
s	int	The ordered state number

This method is used when a state is ordered. If the ordered state has been determinized and kept it returns the ordered state immediately. Otherwise it picks up the subset representing the ordered state from the hash table, and pass the subset to module *fstdeter* for further determinization processing. When the

method *GetFSTStateByNum* is called the number of the ordered state has to be provided by the user and the ordered state has to be reachable from the last ordered state. If an error happened, such as the ordered state cannot be reached from the last ordered state a NULL pointer is returned.

- **GetFSTStateAcceptCostByNum**

  This method does the same as that of the method *GetFSTStateByNum* except that it returns the accept cost of a FST state. If an error happened such as if the ordered state cannot be reached from the last ordered state a *float* number *-1* is returned.

The GetFSTStateAcceptCostByNum method attributes		
Argument	Type	Description
return value	float	The accept cost of a FST state
fst	FST	A pointer to a FST transducer
s	int	The ordered state number

- **GetFSTStateNumArcsByNum**

  This method does the same as that of the method *GetFSTStateByNum* except that it returns the number of total arcs of a FST state. If an error happened such as if the ordered state cannot be reached from the last ordered state an *integer* *-1* is returned.

The GetFSTStateNumArcsByNum method attributes		
Argument	Type	Description
return value	int	The number of arc belonging to a state
fst	FST	A pointer to a FST transducer
s	int	The ordered state number

- **GetFSTStateFirstArcByNum**

  This method does the same as that of the method *GetFSTStateByNum* except that it returns a pointer that points to one element of the arc pointer array. This

element then points to the first arc of a FST state. If an error happened such as if the ordered state cannot be reached from the last ordered state a NULL pointer is returned.

The GetFSTStateFirstArcByNum method attributes		
Argument	Type	Description
return value	FSTArc	A pointer to the arc pointer array
fst	FST	A pointer to a FST transducer
s	int	The ordered state number

- **SetCompValue**

  This method is very important. It sets the value used to compare whether the residual weights of two triples in two different subsets are equal.

The SetCompValue method attributes		
Argument	Type	Description
value	float	A value used for comparison

- **SetFSTBufferSize**

  This method is used to define the size of the buffer in the determinization. The default value is 300000. If the buffer size equals to zero no determinized state is kept in the buffer.

The SetFSTBufferSize method attributes		
Argument	Type	Description
number	int	An integer used for the buffer size

- **GetDeterFST**

  It gets the whole resulting sequential transducer T_2 if the buffer size is large enough. This function put the whole transducer into a text file.

- **ViewNewFSTState**

This function is used for the debugging purpose. It prints all information of a FST state of the resulting transducer T_2.

The ViewNewFSTState method attributes		
Argument	Type	Description
state	FSTState	A pointer to a FST state

More details about the module *fstdeterlib* are available in the Appendix, from *fstdeterlib.h* and *fstdeterlib.c*.

2. **Module fstdeter**

This module provides the services used by the *fstdeterlib* module. These services are used to determinize the subset passed from the *fstdeterlib* and transfer the subset to a FST state of the resulting transducer T_2. Also during the determinization all special cases (special case 1 and special case 2, see Chapter 3 page) are handled in this module.

3. **Module fstdeterstate**

The module *fstdeterstate* contains the definition of the structure *_state* and provides its utility methods such as the method *MakeState* that creates a *_state* data.

4. **Module fstdetersubState**

This module contains the definition of the structure *_subState* and provides its utility methods.

5. **Module fstdeterrOutSym**

This module contains the definition of the structure *_rOutSym* and provides its utility methods.

6. **Module fstdeternewArc**

This module contains the definition of the structure *_newArc* and provides its utility methods.

7. **Module fstdeterFSTState**

This module defines two variables and provides their utility methods. These two variables are the following:

- **fstStates**. It is a state pointer array that each element points to a *_fststate* data structure. The *fstStates* is designed as a buffer to keep the determinized states.

The default buffer size is 300000. This size can be changed by the method *SetBufferSize* from the module *fstdeterlib*.

- **fstStates1**. It is also a state pointer array which only keeps the just determinized state in its first element *fstStates1*[0]. The *fstStates1* is used when the buffer size is zero or the buffer is full.

8. **Module fstdeterFSTArc**

This module defines two variables and provides their utility methods. These two variables are the following:

- **fstArcs**. It is an arc pointer array that each element points to a *_fstarc* data structure. This variable keeps all arcs of the determinized states in the buffer.
- **fstArcs1**. It is also an arc pointer array which only keeps the arcs belonging to the just determinized state kept by *fstStates1*[0].

9. **Module fstdeterstateHashTable**

This module defines two hash tables and provides their utility methods. These two hash tables are the following:

- **stateHashTable**. This variable is used to keep all subsets expanded from the determinized states in the buffer.
- **stateHashTable1**. It only keeps the subsets expanded from the just determinized state kept by *fstStates1*[0].

More details on each module can be found in the Appendix, from its corresponding source file(s).

6.3.2 Data Flow and Function Call Sequence

The data flow and function call is started when one method in module *fstdeterlib* is invoked. Suppose that the state ordered by the user is not determinized yet and can be reached from the last ordered state. Then the data flow and function call sequence is diagrammed in Figure 6.6.

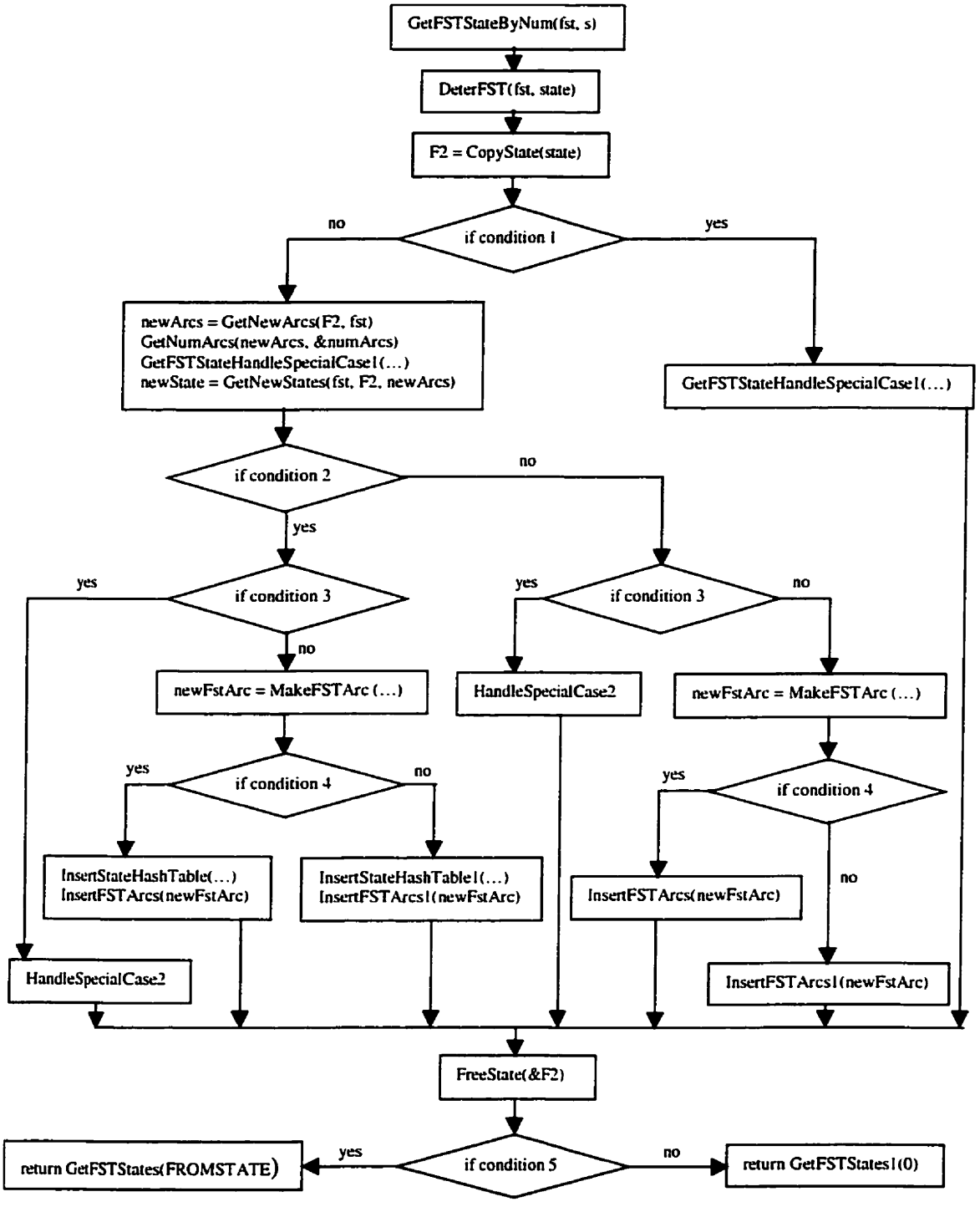


Figure 6.6 The data flow and function call sequence

In Figure 6.6 as an example the method *GetFSTStateByNum* is called by the user. Similar results can be obtained if other method in module *fstdeterlib* such as the method *GetFSTStateAcceptCostByNum*, *GetFSTStateNumArcsByNum*, or *GetFSTStartState*, or *GetFSTStateFirstArcByNum* is called.

The variable F_2 is a subset corresponding to the ordered state. F_2 is transferred to a state of T_2 by the method *GetFSTStateHandleSpecialCase1*.

The variable *newState* is a list of subsets expanded from the subset F_2 (or the ordered state). If the buffer size is not zero these subsets need to be checked before they are put into the hash table.

The variable *newArcs* is a list of intermediate arcs of the ordered state. Each element of *newArcs* will be transferred to an arc of the ordered state.

The varible *newFstArc* is used to create one arc of the ordered state.

More details about the methods in Figure 6.6 can be found in the Appendix. The methods *DeterFST*, *GetNewArcs*, *GetNumArcs*, *GetFSTStateHandleSpecialCase1*, *GetNewStates*, *HandleSpecialCase2* are from module *fstdeter*. The *InsertStatehashTable* method is from module *fstdeterstateHashTable*. The method *InsertFSTArcs* is from module *fstdeterFSTArc*. The method *GetFSTStates* is from module *fstdeterFSTState*. The methods *CopyState* and *FreeState* are defined in module *fstdeterstate*.

In Figure 6.6 there are 5 conditions to be tested. Condition 1 asks that if subset F_2 is generated by special case 1 or special case 2 (see Chapter 3, page 34, 35). Condition 2 asks that if the output of an intermediate arc of the ordered state is symbol. Condition 3 asks that if a subset expanded from the ordered state is a new subset. Condition 4 asks that whether the buffer is full. Condition 5 tests whether the buffer is used.

6.4 Implementation of the SSW_determinization Algorithm

The implementation of the *SSW_determinization* is similar as that of the *DSSW_determinization*. The differences can be seen from the following two points:

1. There is no buffer implemented in the *SSW_determinization* program. All determinized states are kept. The resulting sequential transducer T_2 is always saved to the disk when the determinization terminates.

2. A queue is implemented in the program. This queue is used to keep all subsets not yet be expanded. The first element of the queue is always the subset that should be expanded at the next. Therefore, no search is required to find out the next expanding subset.

Compared with the *DSSW-determinization* program the speed of the *SSW_determinization* program is faster because the implementation of the queue. However, the *SSW_determinization* program always takes more memory because it keeps all the determinized states.

6.5 Implementation of the PSSW_determinization Algorithm

The implementation of the *PSSW_determinization* is similar as that of the *SSW_determinization*. The special case 1 and special case 2 are not considered for the implementation because they have never happened in the partial determinization. See Chapter 4, the Figure 4.1.

In this chapter the implementation of the determinization algorithms for the string-to-string/weight transducers has been introduced. As it is expected the functional test on these programs and the result analysis are carried on in the next chapter.

Chapter 7
Functional Test

Although there is a program used for the determinization of the determinizable string-to-string/weight transducers from the AT&T, no detailed algorithm for the determinization of the string-to-string/weight transducers has been published. In order to check the feasibility of the determinization algorithm *SSW_determinization*, *PSSW_determinization*, and *DSSW_determinization* the functional test on them has to be carried on.

Among these three determinization algorithms the most interesting one is the determinization on the demand because that it can be used for both determinizable and non-determinizable transducers, and it takes little memory during the determinization. Therefore, the functional test introduced in this chapter is mainly focused on the determinization on the demand. Considering the similarity of these three algorithms the functional test results in this chapter can also be used to estimate other two algorithms.

The functional test includes the correctness, the cost of time and space, and other characteristics of the determinization. The correctness is analyzed by comparing the determinization resulting transducer with the transducer obtained from the AT&T determinization program.

Also, in this chapter an application of the Automatic Speech Recognition on a large set of input strings has been designed to further test the determinization results.

7.1 Correctness

The correctness of the determinization algorithms is evaluated by using the *DSSW_determinization* and a commercial software *fsmdeterminize*. The *fsmdeterminize* is provided by AT&T, and is applicable only for the complete determinization of the determinizable string-to-string/weight transducers. To get the whole resulting transducer by *DSSW_determinization* the default buffer and the Breadth First Traversal algorithm are used.

After determinization the resulting transducers are analyzed by applying an AT&T software *fsminfo*. The software *fsminfo* goes through a transducer and collects important information listed in Table 7.1, which the information can be used to analyze and compare the determinization results obtained from the *DSSW_determinization* and *fsmdeterminize*.

Transducer and its property	T_1	T_1'	T_1"
number of states	115959	30637	20005
number of arcs	189800	113962	387284
initial state	0	0	0
number of final states	2	37	1
number of i/o ε	3	37	0
number of input ε	4	51985	
number of output ε	128603	59128	
number of accessible states	115959	30637	20005
number of coaccessible states	115959	30637	20005
number of connected states	115959	30637	20005
number of strongly conn components	1	826	4

Table 7.1 String-to-string/weight transducers used for the test

Table 7.1 lists three determinizable string-to-string/weight transducers T_1, T_1' and T_1" for the test. Specially, T_1" is sequential. The determinization resulting transducers of T_1, T_1' and T_1'' are named T_2, T_2' and T_2'' respectively. The analysis results on these transducers obtained from the *fsmdeterminize* and *DSSW_determinization* are shown in Table 7.2 and Table 7.3. The only difference between Table 7.2 and Table 7.3 is on T_2. The *DSSW_determinization* result has 3 less arcs and 35 more empty input symbols than the *fsmdeterminize* result. This is because the special cases (see Chapter 3, page 34, 35) handled in these two algorithms is different. However, the two results are equivalent because they are both equivalent to the original transducer T_2. Because T_1'' is sequential, therefore its determinization result is same as itself. See T_2'' in Table 7.2 and 7.3.

Transducer and its property	T_2	T_2'	T_2"
number of states	33984	4493	20005
number of arcs	79781	16111	387284
initial state	0	0	0
number of final states	4	260	1
number of i/o ε	3	1127	0
number of input ε	8926	2586	
number of output ε	39623	9609	
number of accessible states	33984	4493	20005
number of coaccessible states	33984	4493	20005
number of connected states	33984	4493	20005
number of strongly conn components	2	116	4

Table 7.2 The determinization results of AT&T fsmdeterminize

Transducer and its property	T_2	T_2'	T_2"
number of states	33984	4493	20005
number of arcs	79778	16111	387284
initial state	0	0	0
number of final states	4	260	1
number of i/o ε	3	1127	0
number of input ε	8961	2586	
number of output ε	39623	9609	
number of accessible states	33984	4493	20005
number of coaccessible states	33984	4493	20005
number of connected states	33984	4493	20005
number of strongly conn components	2	116	4

Table 7.3 The determinization results of DSSW_determinization

Table 7.4 shows the size of each transducer before and after the determinization. One can be seen from this table is that the efficiency of determinization is based on the degree of non-determinism of the original transducer. For example T_1' has a significant reduction on it size, which is up to 85.8%. On the other hand, there is no size reduction if the original transducer is sequential such as T_1''.

Transducer	T_1	T_1'	T_1''
before determinization	4.89 MB	2.19 MB	6.4 MB
after determinization	1.68 MB	0.31 MB	6.4 MB
Percentage (%)	65.5	85.8	0

Table 7.4 Sizes of the transducers in Table 7.2 before and after the determinization

The analysis results of *fsminfo* on the resulting transducers tell us that the determinization results obtained from *DSSW_determinization* and *fsmdeterminize* are equivalent. The following section gives a further test on the equivalency of the resulting transducers by applying these determinization programs to a true ASR process.

7.1.1 Automatic Speech Recognition Test

To further test the determinization results the following ASR procedures are used in the test.

1. Get the distribution-to-phonemes transducer
2. Determinize the transducer
3. Get the complete distributions-to-words recognition transducer
4. Get the compacted recognition transducer
5. Convert the compacted transducer into format required by Viterbi decoder
6. Run recognition (Viterbi decoder) on 300 files

The step 2 needs a determinization program. By changing this program the final recognition results based on *DSSW_determinization* and *fsmdeterminize* are compared. The recognition results on 300 files are same (evaluated by the recognition scores) when *DSSW_determinization* is used instead of *fsmdeterminize*. This means that the resulting

transducers obtained from *DSSW_determinization* and *fsmdeterminize* are equivalent. A typical recognition scores analysis of the recognition result is shown in Figure 7.1.

```
====================== HTK Results Analysis =======================
  Date:  Fri May  5  19:45:52 2000
  Ref :  /misc/speech_data26/aupelf/train/adapt/bref-total-words.mlf
  Rec :  vite_qiu_300t2_t160_p-5_s10.mlf
------------------------ Overall Results --------------------------
SENT: %Correct=5.00  [H=15, S=285, N=300]
WORD: %Corr=70.93, Acc=63.39 [H=3220, D=263, S=1057, I=342, N=4540]
===================================================================
```

Figure 7.1 A typical analysis of the recognition result

In conclusion, as a result of these tests that the AT&T work can be reproduced by using our determinization algorithms.

7.2 Comparison Parameter

In Chapter 6, it is mentioned that the method *SetCompV* in module *fstdeterlib* is used to change the comparison parameter *compV*. The comparison parameter *compV* is a *float* number that is used to decide the equivalence of two residual weights in two triples.

Suppose, there are two triples (q_1', w_1', x_1') and (q_2', w_2', x_2'). If $|x_1' - x_2'| \leq$ *compV* then x_1' equals to x_2'. The value of *compV* is very important. It dramatically influences the size of the resulting transducer since each expanding subset during the determinization has to be checked to see if this subset is a new subset (each unique subset represents a state in the resulting transducer). The default value of *compV* is 0.0003, which is obtained according to the test result of a large number of transducers applied in our ASR research. The transducers in Table 7.3 are the results by using the default value in the determinization.

Figure 7.2 is a typical test result obtained from the determinization of T_1 with different comparison values. It can be seen from this figure that if the value is less than 0.00001 the number of arcs or states of the resulting transducer is increased dramatically. Figure 7.3 is another obtained from the determinization of a large transducer with 6M arcs and 20K states. The number of arcs or states of the resulting transducer in Figure 7.3 is increasing with the decreasing of the comparison value. When the comparison value is 0.0003 the determinization result equals to the result of the *fsmdeterminize*.

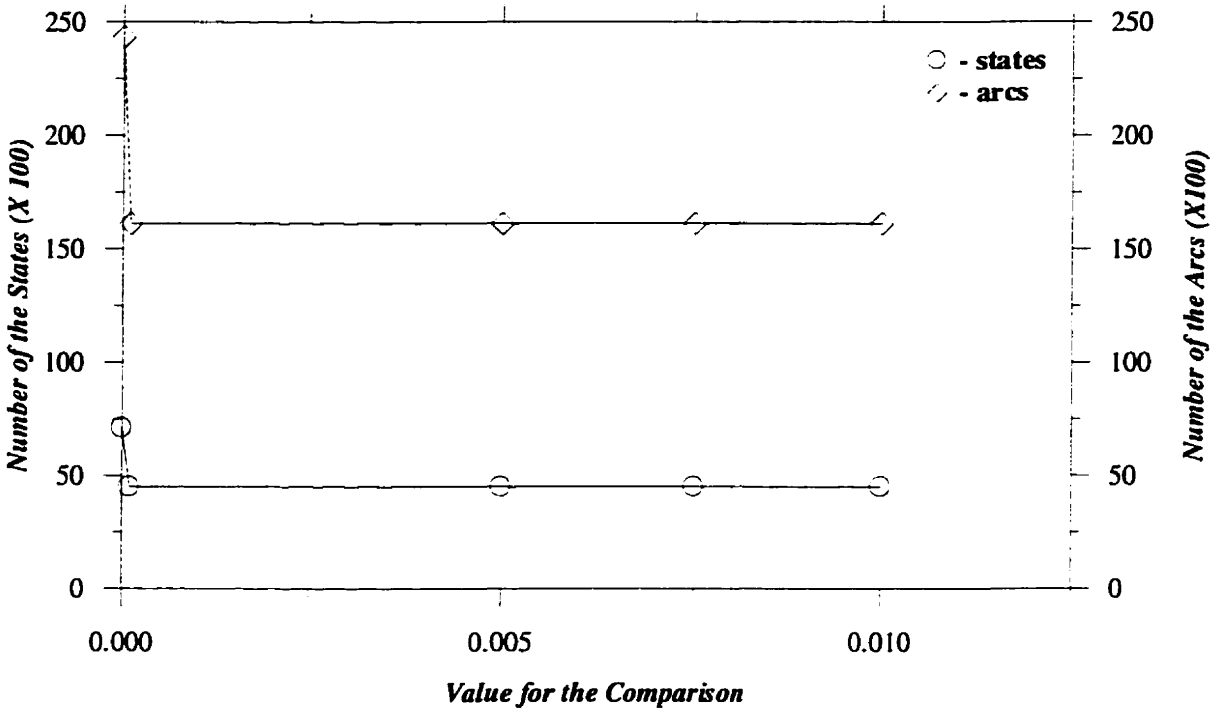


Figure 7.2 The determinization results with different comparison values

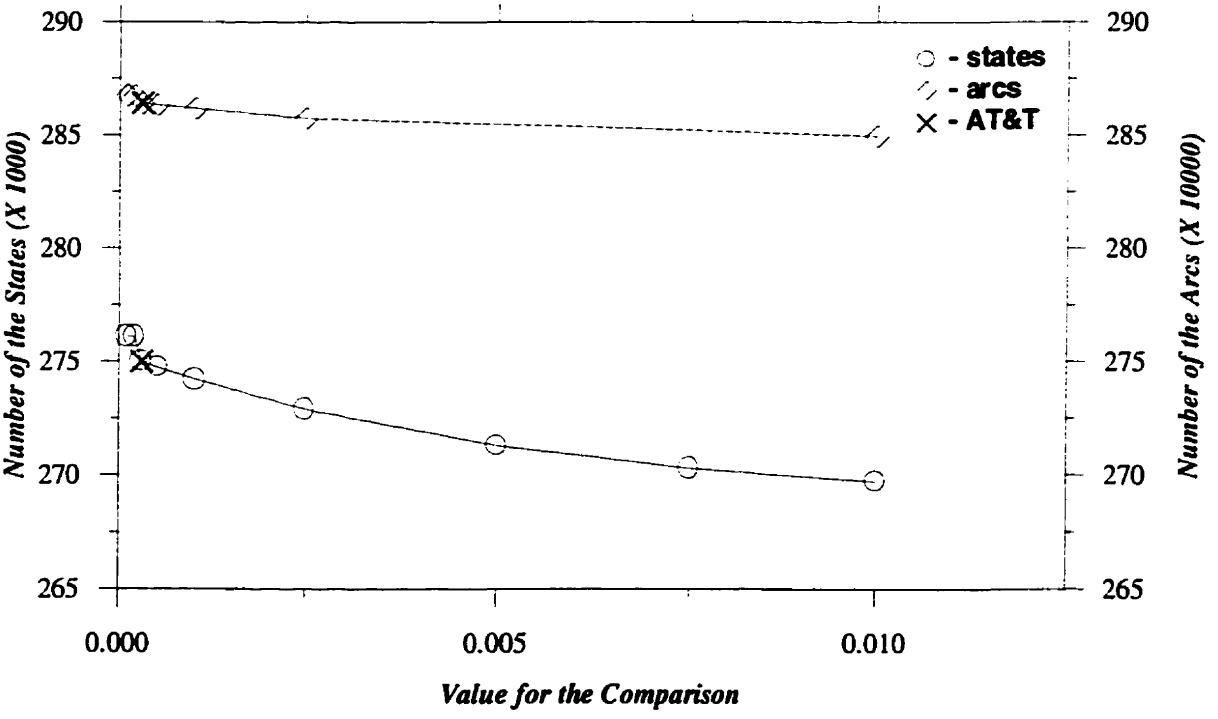


Figure 7.3 The determinization results with different comparison values

7.3 Time and Space

Another important property should be tested for the determinization programs is the requirement of the time and space during the determinization.

To test the time and space a large transducer shown in Table 7.5 is used. The size of this transducer is 90MB. To load it 130MB memory is needed. Table 7.5 also gives the determinization result by *DSSW_determinization* and *fsmdeterminize*. It is interesting that the resulting transducer has more states than the original transducer. But the original transducer has more arcs, which is the reason that the size of the original transducer is larger than (almost 2 times) the resulting transducer because transducer is stored only by its arcs and a few accept states (see Figure 6.1 in Chapter 6).

Transducer and its property	T_1	T_2 (AT&T)	T_2 (CRIM, 0.01)
number of states	192632	275069	269715
number of arcs	5423636	2865945	2849516
initial state	0	0	0
number of final states	1	1	1
number of i/o ε	9562	9374	9373
number of input ε	17223	34783	32828
number of output ε	3078013	2327666	2326223
number of accessible states	192632	275069	269715
number of coaccessible states	192632	275069	269784
number of connected states	192632	275069	269784
number of strongly conn components	2175	4508	7578
transducer size	90 MB	49 MB	47 MB

Table 7.5 A large string-to-string/weight transducer and its determinization results

Figure 7.4 shows the time – memory curves of the determinization on this transducer by using different determinization programs. The memory displayed in Figure

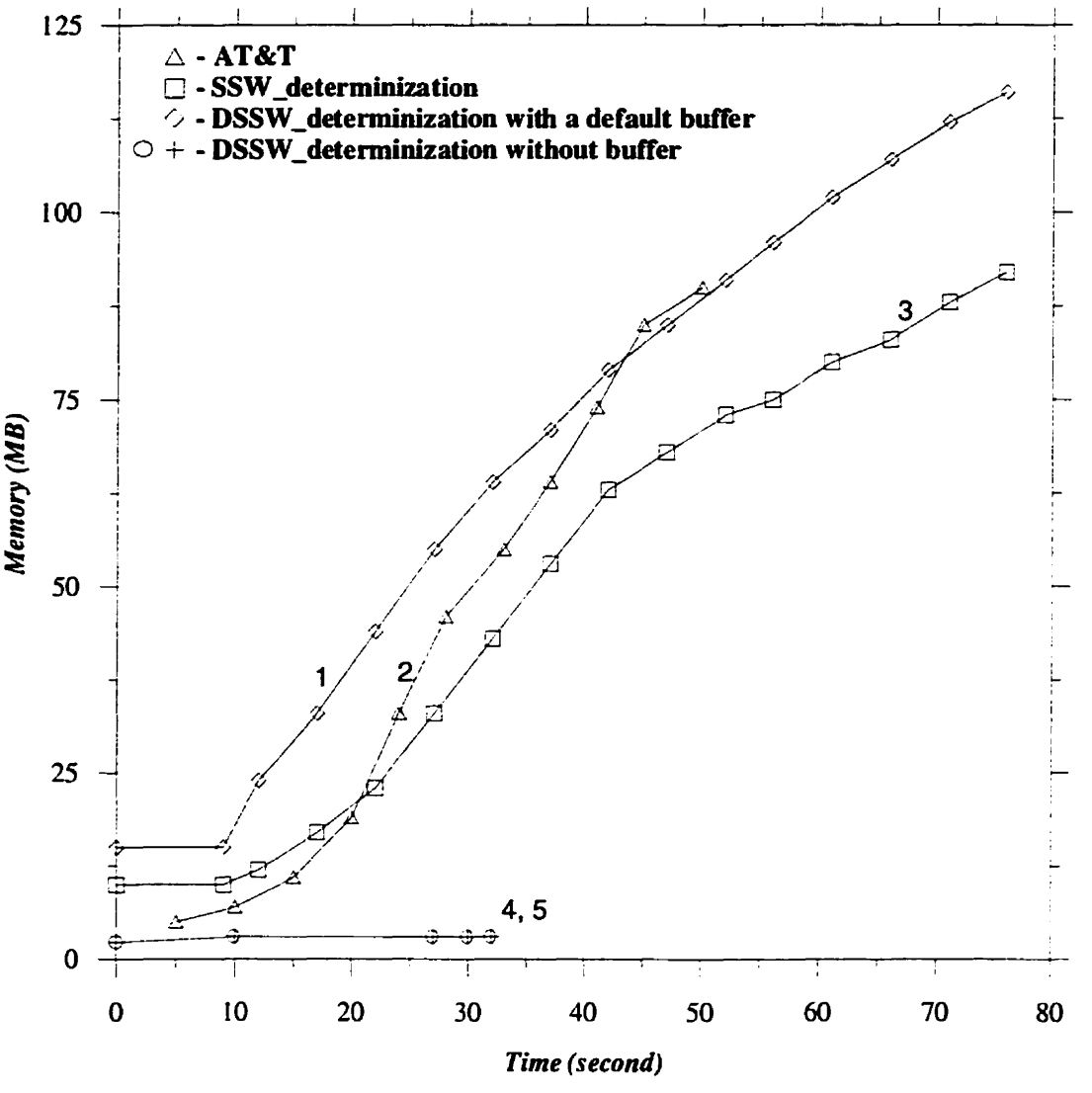


Figure 7.4 Relationship of time and memory during the determinization

7.4 is only the memory required for the determinization since it has been revised by a deduction for the loading.

In Figure 7.4 curve 1, the *DSSW_determinization* needs 116MB memory and 76 seconds. The *fsmdeterminize* (curve 2) needs 90MB and 50 seconds. The reason that *DSSW_determinization* takes more memory is because when the *DSSW_determinization* is used to get the whole resulting transducer it constructs and keeps both all the states and all the arcs during the determinization. In fact, most states are not needed for the resulting

transducer because the arcs have included all the information for the states and the transducer is stored by the arcs and the information on a few accept states. And also the determinization needs only subset but state. Curve 3 is the result obtained from the *SSW_determinization* which only constructs and keeps the accept states during the determinization. It takes 95MB, which is very close to the result of *fsmdeterminize.*

Notice that it is slightly different at the start point and on the shape of curves 1, 2, and 3. For example, the *DSSW_determinization* with the default buffer needs 15MB to declare the state pointer array and the arc pointer array. These differences mean that the memory allocation strategy between our determinization programs and the *fsmdeterminize* is slightly different.

The most interesting information in Figure 7.4 is from curve 4 and curve 5 (overlapped). These two curves are the results of the *DSSW_determinization* without buffer. When the *DSSW_determinization* without buffer is used the determinization returns the ordered state, and only keeps the just determinized (or returned) state with some related information (such as the arcs and the expanding subsets of this state) for the further determinization. At the start point the memory allocated for the state pointer array and the arc pointer array is 2MB. The memory allocated during the determinization is 1MB. The total memory cost is only 3MB. See Figure 7.4 curve 4 and curve 5, which are corresponding to two determinization paths.

Notice that this 1MB memory is allocated only once when the determinization starts, it is supposed to hold all the information for the ordered state. If 1MB memory is not enough for a determinized state during the determinization another 1MB will be added. However, this case never happens because that 1MB allows a determinized state to have 1000 different expands (in an optimal situation [31]), which means that 1MB should be enough for any determinized state of any string-to-string/weight transducer.

Therefore, based on the results in Figure 7.4 it can be concluded that our *SSW_determinization* program takes similar time and memory as the AT&T *fsmdeterminize* does. The *DSSW_determinization* (without buffer is the normal usage of this program) takes the lowest memory among all the determinization programs during the determinization.

7.4 Partial Determinization of the Non Determinizable Transducers

Transducer and its property	T_1	T_1'
Number of states	1444	2198713
Number of arcs	53428	5822490
initial state	0	0
number of final states	37	12
number of i/o ε	37	681763
number of input ε	37	1402275
number of output ε	37	4505275
number of accessible states	1444	2198713
number of coaccessible states	1444	2198713
number of connected states	1444	2198713
number of strongly conn components	76	84137

Table 7.6 String-to-string/weight transducers used for the partial determinization test

Transducer and its property	T_2	T_2'
number of states	1681	2590222
number of arcs	13167	5931825
initial state	0	0
number of final states	93	12
number of i/o ε	2509	1252837
number of input ε	2509	2045142
number of output ε	2509	4876264
number of accessible states	1681	2590222
number of coaccessible states	1681	2590222
number of connected states	1681	2590222
number of strongly conn components	109	123527

Table 7.7 Results of partial determinization of the transducers in Table 7.6

Although there is no tool available for the comparison on the result of the *PSSW_determinization*, the test on *PSSW_determinization* is carried on by using the non-determinizable transducers. T_1 and T_1' in Table 7.6 are two typical non-determinizable transducers. Their partial determinization results are T_2 and T_2' respectively shown in Table 7.7.

The size of T_1 is 0.88MB. After the determinization the number of the arcs of T_2 is reduced from 53K to 13K. As it is expected that the size of T_2 is only 0.23MB. The deduction is up to 74%, which means the partial determinization program is very efficient.

Similarly as the *SSW_determinization* (see Chapter 3, Figure 3.5), the partial determinization of T_1' tells a different story. After the determinization the size of T_2' is slightly increased from 5.8M to 5.9M. However, the advantage is that T_2' is closer to a sequential transducer. This will dramatically increase the searching speed when T_2' is applied to the ASR process instead of T_1'.

In conclusion, this chapter has carried a successful functional test on the determinization algorithms. The results are interesting and exciting. The *DSSW_determinization* only needs 3MB for the determinization.

The final conclusions on the research of the determinization algorithms of the string-to-string transducers in this paper are made in the next chapter.

Chapter 8

Conclusion and Future Work

The purpose of the research carried in this thesis is to develop the determinization algorithms for the string-to-string/weight transducers. The conclusion of this research and the future work can be summarized as follows:

1. The formal definitions of the string-to-string/weight transducers and the sequential string-to-string/weight transducers have been made. These definitions are successfully used to define the transducers concerned in the determinization algorithms.

As a result of this research three determinization algorithms have been developed for the different requirements of the different string-to-string/weight transducers.

The implementation of the determinization algorithms has been introduced. And, the functional test on these determinization programs has been carried on.

By analyzing and comparing the results of our determinization programs with that of AT&T *fsmdeterminize* it can be concluded that these newly developed determinization algorithms and their implementation are successful. AT&T work has been reproduced in this research.

2. The first developed determinization algorithm is a complete determinization algorithm. It is named *SSW_determinization*. This algorithm can only be used to determinize the determinizable string-to-string/weight transducers (same as AT&T software *fsmdeterminize*). Both time complexity and space complexity of the *SSW_determinization* are exponential to the size of the original transducer.

3. To reduce the degree of non-determinism of the non-determinzable string-to-string/weight transducers. A partial determinization algorithm *PSSW_determinization* is developed. This algorithm can be used to get the whole resulting transducer with much low non-determinism degree compared with that of the original non-determinizable transducer. Notice that does not apply this algorithm to the determinizable transducers.

4. The determinization on the demand algorithm *DSSW_determinization* is the last determinization algorithm developed in this thesis. The characteristic of this algorithms is that it only determinizes a state when this state is ordered, and it (without buffer) only

keeps the information for the just determinized state. The next ordered state must be reachable from the last ordered state.

The *DSSW_determinization* has two advantages over the *SSW_determinization* and *PSSW_determinization*. One is that it can be applied to both the determinizable and non-determinizable string-to-string/weight transducers. Another advantage is the very low memory cost during the determinization. For example, in our current implementation *DSSW_determinization* only needs 3MB. Compared with that in some cases the *SSW_determinization*, *PSSW_determinization* or AT&T *fsmdeterminize* needs more than 1000MB memory for the determinization, the *DSSW_determinization* is very useful when memory is scarce.

With a buffer the *DSSW_determinization* can keep some determinized states (of course, with a memory cost). If the buffer size is large enough the whole resulting transducer are kept (normally, a big memory cost). Re-computation can be avoided if the ordered state is kept in the buffer. Also, the size of the buffer can be changed to limit the size of the memory used in the determinization.

5. In most cases both time efficiency and space efficiency (up to 85.8% in our test) are increased after the determinization. However, the space efficiency is slightly decreased (such as 1.0%) in some determinization cases.

6. Future work should focus on two aspects. The first is the algorithm and its implementation for the prediction of the determinizability of the string-to-string/weight transducers. Although we have discussed the principles of the prediction, more details need to be considered for the string-to-string/weight transducers, like the special cases in the determinization algorithms. The second is the optimization of the implementation of the determinization algorithms. Such as the memory allocation strategy, the data structures used for the determinization, and the data structure of the transducer (FST data structure).

References

1. "The HTK Book" (for HTK Version 2.1). Cambridge University, 1997.
2. Graham, S. L., M. A. Harrison, and W.L. Ruzzo. "An Improved Context-Free Recognizer". ACM Transactions on Programming Languages and Systems, 2, 1980.
3. Mohri, Mehryar. "On the Use of Sequential Transducers in Natural Language Processing". Finite-State Language Processing, edited by Emmanuel Roche and Yves Schabes. A Bradford Book, The MIT Press, Cambridge, Massachusetts. London England. 1997.
4. Kaplan, Ronald M. and Martin Kay. "Regular Models of Phonological Rule Systems". Computational Linguistics, 20, 1994.
5. Mohri, Mehryar. "Compact Representations by Finite-State Transducers". Proceedings of the 32nd Meeting of the Association for Computational Linguistics (ACL 94), Las Cruces, New Mexico. ACL.1994.
6. Kimmo Koskenniemi. "Finite-State Parsing and Disambiguation". In proceedings of the 13th International Conference on Computational Linguistics. COLING-90, Vol. 2. Helsinki, Finland. 1992.
7. S. Khoshanfian and B. A. Baker. "Multimedia and Imaging Databases". Morgan Kaufmann Publishers, San Francisco, Calif. 1996.
8. Mohri, Mehryar. "Finite-State Transducers in Language and Speech Processing". Computational Linguistics, 23, 1997.
9. Kimmo Koskenniemi, Pasi Tapanainen and Atro Voutilainen. "Compiling and Using Finite-State Syntactic Rules". In proceedings of the 15th International Conference on Computational Linguistics. COLING-92, Vol. I. Nantes, France. 1992.
10. Mohri, Mehryar and Michael Riley. "Weighted Determinization and Minimization for Large Vocabulary Speech Recognition". In Proceedings of the Eurospeech '97, Rhodes, Greece, 1997.
11. Fernando C. N. Pereira and Michael D. Riley. "Speech Recognition by Composition of Weighted Finite Automata". Finite-State Language Processing, edited by Emmanuel Roche and Yves Schabes. A Bradford Book, The MIT Press, Cambridge, Massachusetts. London England. 1997.

12. Emmanuel Roche and Yves Schabes. "Finite-State Language Processing". A Bradford Book, The MIT Press, Cambridge, Massachusetts. London England. 1997.
13. Bauer, W. "On Minimizing Finite Automata". SATACS Bulletin, 35, 1988.
14. Mohri, Mehryar. http://www.cs.columbia.edu/~mohri/notes.html
15. Gibbon, Dafydd. http://corel.lili.unibielefeld.de/Classes/Winter97
16. Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. "Compilers, Principles, Techniques and Tools". Addison Wesley: Reading, MA. 1986.
17. Aho, Alfred V., John E. Hopcroft, and Jeffrey D. Ullman. "The Design and Analysis of Computer Algorithms". Addison Weley: Reading, MA. 1974.
18. Dominique Revuz. "Minimisation of Acyclic Deterministic Automata in Linear Time". Theoretical Computer Science, 92, 1992.
19. Kleene, Stephen C. "Representation of Events in Nerve Nets and Finite Automata". In C. E. Shannon and J. McCarthy, editors, Automata Studies. Princeton University Press. 1956.
20. Eilenberg, Samuel. "Automata, Languages, and Machines". Volume A. Academic Press, New York. 1974.
21. Eilenberg, Samuel. "Automata, Languages, and Machines". Volume B. Academic Press, New York. 1976.
22. Mehryar Mohri. "On Some Applications of Finite-State Automata Theory to Natural language Processing". Natural Language Engineering, 2, 1996.
23. Perrin, Dominique. "Finite automata". Handbook of Theoretical Computer Science, 1990.
24. Salomaa, A. "Formal Languages". Academic Press, New York, NY. 1973.
25. Michael Siper. "Introduction to the Theory of Computation". PWS Publishing Company, 20 Park Plaza. Boston, MA 02116. 1997.
26. Mohri, Mehryar. "Minimization Algorithms for Sequential Transducers". Theoretical Computer Science, 1998.
27. AT&T Labs-Research. http://www.research.att.com
28. Robert M. Keller. http://www.cs.hmc.edu/~keller/courses/cs60/slides/Acceptors.html
29. Berstel, Jean. "Transductions and Context-Free Languages". B. G. Teubner Stuttgart, 1979.

30. Riley, Pereira and Mohri, Mehryar. "Transducer Composition for Context-Dependent Network Expansion". In Proceedings of the Eurospeech '97, Rhodes, Greece, 1997.
31. Qiu, Jun. Technical Report, CRIM, Jan. 2000.

Appendix

```
/*****************************************************************************/
/*                                                                           */
/* Copyright (c) 2000 Centre de recherche informatique de Montreal           */
/*                                                                           */
/*****************************************************************************/

/**** FILE HEADING ***********************************************************
 * fstdeterlib.h -- Declaration of the interface methods that can be
 * used directly by the user. See the next page on the detailed
 * description of these methods (in fstdeterlib.c).
 */

#ifndef __FSTDETERLIB_H__
#define __FSTDETERLIB_H__

#include <stdio.h>
#include <math.h>
#include "fst.h"
#include "fststate.h"
#include "fstarc.h"

extern void InitFSTDeter();

extern FSTState *GetFSTStartState(FST *fst);

extern FSTState *GetFSTStateByNum(FST *fst, int s);

extern float GetFSTStateAcceptCostByNum(FST *fst, int s);

extern int GetFSTStateNumArcsByNum(FST *fst, int s);

extern FSTArc **GetFSTStateFirstArcByNum(FST *fst, int s);

extern void ViewNewFSTState(FSTState *state);

extern void GetDeterFST();

extern void SetCompA(float value);

extern void SetFSTBufferSize(int number);

#endif /* __FSTDETERLIB_H__ */
```

```
/*****************************************************************/
/*                                                               */
/* Copyright (c) 2000 Centre de recherche informatique de Montreal    */
/*                                                               */
/*****************************************************************/

/**** FILE HEADING ************************************************
 * fstdeterlib.c -- The Implementation of the interface of the
 * determinization algorithm.
 */

#include <stdio.h>
#include "fstdeterlib.h"
#include "fstdeter.h"
#include "fstdeterstate.h"
#include "fstdetersubState.h"
#include "fstdeterrOutSym.h"
#include "fstdeterFSTState.h"
#include "fstdeterFSTArc.h"
#include "fstdeterstateHashTable.h"

/***** FUNCTION HEADING *******************************************
 *
 * NAME: InitFSTDeter()
 *
 * PURPOSE:
 *      Initiate the hash tables, and the global variables. This method
 *      has to be called before the determinization.
 *
 * AUTHOR / MAINTAINED BY:
 *      JUN QIU
 *****************************************************************/

void
InitFSTDeter()
{
     InitStateHashTable();
     InitStateHashTable1();
}

/***** FUNCTION HEADING *******************************************
 *
 * NAME: GetFSTStartState(FST *fst)
 *
 * PURPOSE:
 *      Return a pointer to the first state (_fststate) of a
 *      determinized FST. This method has to be called after the method
 *      InitFSTDeter().
 *
 * AUTHOR / MAINTAINED BY:
 *      JUN QIU
 *****************************************************************/

FSTState *
GetFSTStartState(FST *fst)
{
     int start;
```

```
    LROUTSYM temp1 = NULL;
    LSUBSTATE temp2 = NULL;
    LSTATE F0 = NULL;
    FSTState *fstState;

    /* get the start state number of the original transducer T1 */
    start = GetStartFST(fst);

    temp1 = MakeROutSym(0);
    temp2 = MakeSubState(start, temp1, 0, NULL);
    F0 = MakeState(0, temp2);

    if(GetFSTBufferSize() > 0)
      {
     InsertStateHashTable(F0, 0);
      }

    fstState = DeterFSTState(fst, F0);

    return fstState;
}

/*************************************************************************
* NAME: GetFSTStateByNum(FST *fst, int s)
*
* PURPOSE:
*      Return a pointer to a FST state (_fststate) from
*      the determinized FST.
*
* AUTHOR / MAINTAINED BY:
*      JUN QIU / JUN QIU
*************************************************************************/

FSTState *
GetFSTStateByNum(FST *fst, int s)
{
    int index;
    FSTState *fstState;
    LSTATE state, state1 = NULL;

    /* if the ordered state number is larger than the FROM state
       number, and the FROM state number equals to TO state number,
       the determinization finished */
    if((s > GetFROMSTATE()) && (GetFROMSTATE() == GetTOSTATE()))
      {
       printf("\ndeterminization finished!\n");
       return NULL;
      }

    /* if the ordered state number is larger than the TO state number
       this state is not reachable */
    if(s > (GetTOSTATE()))
      {
       printf("\nstate %i cannot be reached now!\n", s);
       exit(0);
      }
```

```
else
  {
 /* if the buffer is not full or the information on the ordered
    state can be found from the buffer */
 if(s <= GetFSTBufferToStateNum() ||
         GetFSTBufferToStateNum() == -1)
   {
    fstState = GetFSTStates(s);
    if(fstState->acceptCost > -0.5)
      {
       return fstState;
      }
    state = GetStateHashTable(fstState->numArcs, s);

    /* if the buffer is full */
    if(GetFSTBufferToStateNum() != -1)
      {
       FreshStateHashTable1();
       ResetFstArcsIndex1();
       ResetFstStateIndex1();
      }
   }
 else if(s <= GetBeforeOldTOSTATE())
   {
    printf("\nstate %i cannot be reached!\n", s);
    exit(0);
   }
 /* the information on the ordered state must be in state pointer
    array fstStates1 */
 else
   {
    fstState = GetFSTStates1(s - GetBeforeOldTOSTATE());

    state1 = GetStateHashTable1(fstState->numArcs,
                          (s - GetBeforeOldTOSTATE()));

    FreshStateHashTable1();
    ResetFstArcsIndex1();
    ResetFstStateIndex1();

   }

   /* reset the FROM state number to the ordered state number */
   SetFROMSTATE(s);

 if(state1 != NULL)
   {
    fstState = DeterFSTState(fst, state1);
    FreeState(&state1);
   }
 else
   {
    fstState = DeterFSTState(fst, state);
   }
 }

return fstState;
```

```
}

/*****************************************************************************
* NAME: GetFSTStateAcceptCostByNum(FST *fst, int s)
*
* PURPOSE:
*      Get the acceptCost of a FST state from the determinized FST.
*
* AUTHOR / MAINTAINED BY:
*      JUN QIU / JUN QIU
*****************************************************************************/

float
GetFSTStateAcceptCostByNum(FST *fst, int s)
{
     FSTState *fstState;

     fstState = GetFSTStateByNum(fst, s);

     return(fstState->acceptCost);
}

/*****************************************************************************
* NAME: GetFSTStateFirstArcByNum(FST *fst, int s)
*
* PURPOSE:
*      Get the pointer to the first arc of a FST state from the
*      determinized FST.
*
* AUTHOR / MAINTAINED BY:
*      JUN QIU / JUN QIU
*****************************************************************************/

FSTArc **
GetFSTStateFirstArcByNum(FST *fst, int s)
{
     FSTState *fstState;

     fstState = GetFSTStateByNum(fst, s);

     return (fstState->firstArc);
}

/*****************************************************************************
* NAME: GetFSTStateNumArcsByNum(FST *fst, int s)
*
* PURPOSE:
*      Get the number of the total arcs of a FST state from the
*      determinized FST.
*
* AUTHOR / MAINTAINED BY:
*      JUN QIU / JUN QIU
*****************************************************************************/

int
GetFSTStateNumArcsByNum(FST *fst, int s)
{
```

```
    FSTState *fstState;

    fstState = GetFSTStateByNum(fst, s);
    return (fstState->numArcs);
}

/*****************************************************************************
* NAME: ViewNewFSTState(FSTState *state)
*
* PURPOSE:
*      View a FST state.
*
* AUTHOR / MAINTAINED BY:
*      JUN QIU / JUN QIU
*****************************************************************************/

void
ViewNewFSTState(FSTState *state)
{
    int i = 0;
    FSTArc **fstArc;

    if(state == NULL)
      {
       printf("\n state is empty!\n");
       return;
      }
    printf("\n acceptCost = %f\n", state->acceptCost);
    printf("\n numArcs = %i\n", state->numArcs);
    fstArc = state->firstArc;
    while(i < state->numArcs)
      {
       printf("\n %i\t%i\t%i\t%i\t%f\n", fstArc[i]->from,
            fstArc[i]->to,fstArc[i]->sym[ISYM], fstArc[i]->sym[OSYM],
            fstArc[i]->cost);
       i++;
      }
}

/*****************************************************************************
* NAME: GetDeterFST()
*
* PURPOSE:
*      Get the text file of the resulting transducer T2.
*
* AUTHOR / MAINTAINED BY:
*      JUN QIU / JUN QIU
*****************************************************************************/

void
GetDeterFST(char * fsmout)
{
    int i = 0, j = 0, count1 = 0, count2 = 0;
    FILE * f = stdout;
    FSTArc **allFstArcs = GetAllFstArcs();
    FSTState **allFstState = GetAllFstStates();
```

```
    if( fsmout != NULL && *fsmout != '-')
      {
     f = fopen(fsmout, "w");
      }
    count2 = GetFstArcsIndex();
    while(j < count2)
      {
       fprintf(f, "%i\t%i\t%i\t%i\t%f\n", allFstArcs[j]->from,
            allFstArcs[j]->to, allFstArcs[j]->sym[0],
            allFstArcs[j]->sym[1], allFstArcs[j]->cost);
       j++;
      }
    count1 = GetTOSTATE();
    while(i <= count1)
      {
       if(allFstState[i]->acceptCost < WORSTCOST)
       fprintf(f, "%i\t%f\n", i, allFstState[i]->acceptCost);
       i++;
      }
    printf("\n total states and arcs:, %i\t%i\n", i, j);
    fclose(f);
}

/*************************************************************************
* NAME: SetCompA(int value)
*
* PURPOSE:
*      Set the comparison value.
*
* AUTHOR / MAINTAINED BY:
*      JUN QIU / JUN QIU
*************************************************************************/

void
SetCompA(float value)
{
    SetMyCompA(value);
    return;
}

/*************************************************************************
* NAME: SetFSTBufferSize(int number)
*
* PURPOSE:
*      Set the buffer size, the default size is 300000.
*
* AUTHOR / MAINTAINED BY:
*      JUN QIU / JUN QIU
*************************************************************************/

void
SetFSTBufferSize(int number)
{
    SetMyFSTBufferSize(number);
}

/* end of file fstdeterlib.c */
```

```
/*****************************************************************/
/*                                                               */
/* Copyright (c) 2000 Centre de recherche informatique de Montreal    */
/*                                                               */
/*****************************************************************/

/**** FILE HEADING ***********************************************
 * fstdeter.h -- Declaration of the methods used for the
 * determinization.
 */

#ifndef __FSTDETER_H__
#define __FSTDETER_H__

#include <stdio.h>
#include "fst.h"
#include "fstdeterstate.h"
#include "fstdetersubState.h"
#include "fstdeterrOutSym.h"
#include "fstdeternewArc.h"
#include "fstdeterFSTState.h"
#include "fstdeterFSTArc.h"
#include "fstdeterstateHashTable.h"

/* determinize a finite state of the string-to-string/weight transducer
   T1 */
extern FSTState *DeterFSTState(FST *fst, LSTATE state);

/* get the TO state number */
extern int GetTOSTATE();

/* get the FROM state number */
extern int GetFROMSTATE();

/* update the FROM state number */
extern void SetFROMSTATE(int number);

/* set the buffer size, the default size is 300000 */
extern void SetMyFSTBufferSize(int number);

/* get the buffer size */
extern int GetFSTBufferSize();

/* get the TO state number when the buffer is full */
extern int GetFSTBufferToStateNum();

extern int GetBeforeOldTOSTATE();

#endif /* __FSTDETER_H__ */
```

```
/*****************************************************************************/
/*                                                                           */
/* Copyright (c) 2000 Centre de recherche informatique de Montreal           */
/*                                                                           */
/*****************************************************************************/

/**** FILE HEADING ************************************************
 * fstdetersubState.h -- Definition of the _substate data structure and
 * declaration of the utility methods. The _substate data structure is
 * used to represent the subsets.
 */

#ifndef __FSTDETERSUBSTATE_H__
#define __FSTDETERSUBSTATE_H__

#include <stdio.h>
#include "fstdeterrOutSym.h"

typedef struct _subState SUBSTATE, *LSUBSTATE;

struct _subState {
     int s; /* a state number in the original transducer T1 */
     float c; /* residual weight */
     LROUTSYM loutSym; /* residual string */
     LSUBSTATE next;
};

/* set the initial _subState pool size, the default size is 1000 */
extern void SetSubStateNumber(int number);

/* get a _subState data structure from the _subState pool.
   If the pool is empty, create a new pool */
extern LSUBSTATE MakeSubState(int s, LROUTSYM loutSym, float c,
                              LSUBSTATE next);

extern LSUBSTATE CopySubState(LSUBSTATE subState);

/* return a _subState to the _subState pool */
extern void FreeSubState(LSUBSTATE * subState);

#endif /* __FSTDETERSUBSTATE_H__ */
```

```
/*****************************************************************************/
/*                                                                           */
/* Copyright (c) 2000 Centre de recherche informatique de Montreal           */
/*                                                                           */
/*****************************************************************************/

/**** FILE HEADING ************************************************
 * fstdeterrOutSym.h -- Definition of the _rOutSym data structure and
 * declaration of the utility methods. the _rOutSym data structure is
 * used to describe the residual string.
 */

#ifndef __FSTDETERROUTSYM_H__
#define __FSTDETERROUTSYM_H__

#include <stdio.h>

typedef struct _rOutSym ROUTSYM, *LROUTSYM;

struct _rOutSym {
     int outSym; /* represent a symbol in the residual string */
     LROUTSYM next;
};

/* set the initial _rOutSym pool size, the default size is 1000 */
extern void SetROutSymNumber(int number);

/* get a _rOutSym from the _rOutSym pool. If the pool is empty,
   create a new pool */
extern LROUTSYM MakeROutSym(int outSym);

extern LROUTSYM CopyOutSym(LROUTSYM rOutSym);

/* append rOutSym2 to rOutSym1 */
extern LROUTSYM OutSymPlus(LROUTSYM rOutSym1, LROUTSYM rOutSym2);

/* remove rOutSym2 from the head of rOutSym1 */
extern LROUTSYM OutSymSubstract(LROUTSYM rOutSym1, LROUTSYM rOutSym2);

/* compare whether rOutSym1 and rOutSym2 are equivalent */
extern int CompareOutSym(LROUTSYM rOutSym1, LROUTSYM rOutSym2);

/* get the common prefix of rOutSym1 and rOutSym2 */
extern LROUTSYM GetPrefix(LROUTSYM rOutSym1, LROUTSYM rOutSym2);

/* return a _rOutSym to the _rOutSym pool */
extern void FreeROutSym(LROUTSYM *rOutSym);

#endif /* __FSTDETERROUTSYM_H__ */
```

```
/*****************************************************************************/
/*                                                                           */
/* Copyright (c) 2000 Centre de recherche informatique de Montreal           */
/*                                                                           */
/*****************************************************************************/

/**** FILE HEADING ************************************************
 * fstdeternewArc.h -- Definition of the _newArc data structure and
 * declaration of the utility methods. The _newArc data structure is
 * used to represent the intermediate arcs during the determinization.
 */

#ifndef __FSTDETERNEWARC_H__
#define __FSTDETERNEWARC_H__

#include <stdio.h>
#include "fstdeterrOutSym.h"

typedef struct _newArc NEWARC, *LNEWARC;

struct _newArc {
     int inSym; /* the input symbol */
     float cost; /* the cost of the transition */
     LROUTSYM loutSym; /* the output string */
     LNEWARC next;
};

/* set the initial _newArc pool size, the default size is 1000 */
extern void SetNewArcNumber(int number);

/* get a _newArc from the _newArc pool. If the pool is empty,
   create a new pool */
extern LNEWARC MakeNewArc(int inSym, LROUTSYM loutSym, float cost);

/* insert a _newArc into a _newArc list */
extern void InsertNewArc(LNEWARC *newArcs, LNEWARC newArc);

/* return a -newArc to the _newArc pool */
extern void FreeNewArc(LNEWARC *newArc);

#endif /* __FSTDETERNEWARC_H__ */
```

```
/*****************************************************************************/
/*                                                                           */
/* Copyright (c) 2000 Centre de recherche informatique de Montreal           */
/*                                                                           */
/*****************************************************************************/

/**** FILE HEADING ************************************************************
 * fstdeterFSTState.h -- Declaration of the utility methods used for the
 * _fststate data structure. The _fststate data structure represents the
 * states of the string-to-string/weight Finite-State Transducers.
 */

#ifndef __FSTDETERFSTSTATE_H__
#define __FSTDETERFSTSTATE_H__

#include <stdio.h>
#include "fststate.h"
#include "fstarc.h"

/* get a _fststate from the _fststate pool. If the pool is empty,
   create a new pool */
extern FSTState *MakeFSTState(float AccepCost, int numArcs,
                        FSTArc **firstArc);

extern void UpdateFSTStates(int stateNumber, int index);

extern void UpdateFSTStates1(int stateNumber, int index);

extern void InsertFSTStates(float acceptCost, int numArcs,
                      FSTArc **firstArc, int flag, int stateNumber);

extern void InsertFSTStates1(float acceptCost, int numArcs,
                      FSTArc **firstArc, int flag, int stateNumber);

/* find a _fststate in the state pointer array fstStates, and return
   a pointer to it */
extern FSTState *GetFSTStates(int stateNumber);

/* find a _fststate in the state pointer array fstStates1, and return
   a pointer to it */
extern FSTState *GetFSTStates1(int stateNumber);

/* return the pointer to fstStates */
extern FSTState **GetAllFstStates();

/* let the _fststate pool can be reused */
extern void ResetFstStateIndex1();

#endif /* __FSTDETERFSTSTATE_H__ */
```

```
/*****************************************************************************/
/*                                                                           */
/* Copyright (c) 2000 Centre de recherche informatique de Montreal           */
/*                                                                           */
/*****************************************************************************/

/**** FILE HEADING ************************************************
 * fstdeterFSTArc.h -- Declaration of utility methods used for _fstarc
 * data structure. The _fstarc data structure represents the arcs of the
 * string-to-string/weight Finite-State Transducers.
 */

#ifndef __FSTDETERFSTARC_H__
#define __FSTDETERFSTARC_H__

#include <stdio.h>
#include "fstarc.h"

/* get a _fstarc from the _fstarc pool. If the pool is empty,
   create a new pool */
extern FSTArc * MakeFSTArc(int inSym, int outSym, int from, int to,
                    float cost);

/* insert a _fstarc into the arc pointer array fstArcs */
extern void InsertFSTArcs(FSTArc *fstArc);

/* insert a _fstarc into the arc pointer array fstArcs1 */
extern void InsertFSTArcs1(FSTArc *fstArc);

/* get the pointer to the pointer (in the arc pointer array fstArcs) of
   the first arc of a _fststate */
extern FSTArc **GetFirstArc(int index);

/* get the pointer to the pointer (in the arc pointer array fstArcs1) of
   the first arc of a _fststate */
extern FSTArc **GetFirstArc1(int index);

/* return the pointer to fstArcs */
extern FSTArc **GetAllFstArcs();

/* this index is used to save a new _fstarc in fstArcs */
extern int GetFstArcsIndex();

/* this index is used to save a new _fstarc in fstArcs1 */
extern int GetFstArcsIndex1();

/* let fstArcs1 and the _fstarc pool can be reused */
extern void ResetFstArcsIndex1();

#endif /* __FSTDETERFSTARC_H__ */
```

```
/*****************************************************************************/
/*                                                                           */
/* Copyright (c) 2000 Centre de recherche informatique de Montreal           */
/*                                                                           */
/*****************************************************************************/

/**** FILE HEADING ************************************************
 * fstdeterstateHashTable.h -- Declaration of the methods used for the
 * hash tables of the _state data structure. These hash tables are used
 * to keep the subsets.
 */

#ifndef __FSTDETERSTATEHASHTABLE_H__
#define __FSTDETERSTATEHASHTABLE_H__

#include <stdio.h>
#include "fstdeterstate.h"

/* initiate hash table stateHashTable, which is used to keep all subsets
   expanded from the determinized states in the buffer */
extern void InitStateHashTable();

/* initiate hash table stateHashTable1, which is used to keep all
subsets
   expanded from the just determinized kept by fstStates1[0] */
extern void InitStateHashTable1();

/* insert a _state into stateHashTable */
extern void InsertStateHashTable(LSTATE state, int stateNumber);

extern void InsertStateHashTable1(LSTATE state, int stateNumber);

/* check whether a _state exists in stateHashTable. If it exists, return
   its new state number in the resulting transducer T2 */
extern int FindStateHashTable(LSTATE state);

/* find a _state from stateHashTable */
extern LSTATE GetStateHashTable(int index, int stateNumber);

/* find and return a copy of a _state from stateHashTable1 */
extern LSTATE GetStateHashTable1(int index, int stateNumber);

/* set the number of buckets for stateHashTable */
extern void SetBuckets(int number);

/* set the number of buckets for stateHashTable1 */
extern void SetBuckets1(int number);

/* set the comparison value, the default value is 0.0003 */
extern void SetMyCompA(float value);

/* reinitiate hash table statehashTable1 */
extern void FreshStateHashTable1();

#endif /* __FSTDETERSTATEHASHTABLE_H__ */
```