ELSEVIER

Theoretical Computer Science 231 (2000) 17–32

Theoretical Computer Science

www.elsevier.com/locate/tcs

The design principles of a weighted finite-state transducer library

Mehryar Mohri *, Fernando Pereira, Michael Riley

AT&T Labs-Research, 180 Park Avenue, Florham Park, NJ 07932-0971, USA

Abstract

We describe the algorithmic and software design principles of an object-oriented library for weighted finite-state transducers. By taking advantage of the theory of rational power series, we were able to achieve high degrees of generality, modularity and irredundancy, while attaining competitive efficiency in demanding speech processing applications involving weighted automata of more than 10^7 states and transitions. Besides its mathematical foundation, the design also draws from important ideas in algorithm design and programming languages: dynamic programming and shortest-paths algorithms over general semirings, object-oriented programming, lazy evaluation and memoization.

Keywords: Weighted automata; Finite-state transducers; Rational power series; Speech recognition

1. Introduction

Finite-state techniques have proven valuable in a variety of natural-language processing applications [5–11, 14, 16, 18, 19, 28, 32, 33, 36, 38, 39]. However, speech processing imposes requirements that were not met by any existing finite-state library. In particular, speech recognition requires a general means for managing *uncertainty*: all levels of representation, and all mappings between levels, involve alternatives with different probabilities, since there is uncertainty in the interpretation of the speech signal at all levels. Previous speech recognition algorithms and systems relied on "ad hoc" methods for combining finite-state representations with uncertainty. However, by taking advantage of the theory of rational power series, we were able to develop a library for building and applying weighted finite-state transducers that can represent together all

* Corresponding author. Tel.: +1-973-360-8536; fax: +1-973-360-8092.
E-mail address: mohri@research.att.com (M. Mohri)

0304-3975/00/$ - see front matter
PII: S0304-3975(99)00014-6

the finite-state and uncertainty management operations in speech recognition while creating the opportunity for hitherto unrecognized optimizations and achieving competitive or superior performance in many speech recognition tasks [23, 24, 30].

This paper focuses on the overall design of the library starting from its mathematical foundation, rather than on specific algorithms or applications, which have been described elsewhere [18, 21, 23–25, 27, 30]. Although our initial motivation was to improve the tools available for speech recognition, we aimed always for the highest degree of generality compatible with the mathematical foundation and with the efficiency demands of the application. By basing our datatypes on the least restrictive algebraic structures compatible with the desired algorithms, we were able to avoid redundant implementations of the same generic algorithm on related but distinct datatypes, thus creating a design with a minimal, highly modular core of algorithms. In addition, by using mathematically defined datatypes, we can abstract away from implementation details in most of the user-visible parts of the library, while being able to support a variety of implementations with different performance characteristics for datatypes and operations.

One of the central steps of program design is to factor the task under study into algorithm and data structures. We suggest here a mathematical analogue of that principle: the separation of algebra and algorithms. In other words, our algorithms should be designed to work in as general an algebraic structure as possible.

We start by outlining the mathematical foundation for the library in Section 2. Operating at the higher level of generality of weighted finite-state transducers requires new algorithms that are not always straightforward extensions of the corresponding classical algorithms for unweighted automata, as discussed in Section 3.1. In particular, we use the example of ε-removal in Section 3.2 to illustrate how that higher level of generality can be attained efficiently by using general shortest-paths computations over semirings.

The efficiency of the library in some applications depends crucially on delaying the full computation of operation results until they are needed. While this idea had been used in previous finite-state tools, for instance the on-demand determinization in `egrep` [2], our library uses lazy evaluation for all operations satisfying certain locality constraints, as explained in Section 3.3.

These mathematical and algorithmic considerations led to a set of general operations on a simple and general automaton datatype with a range of possible implementations, which are discussed in Section 4.

Finally, in Section 5, we present in more detail the requirements and current status of our main application, speech recognition, and illustrate with an application of the library to a simplified version of a typical speech-processing task.

2. Mathematical foundations

The generality of our library derives from the algebraic concepts of *rational power series* and *semiring*. A semiring $(K, \oplus, \otimes, \bar{0}, \bar{1})$ is a set K equipped with two binary

operations $\oplus$ and $\otimes$ such that $(K, \oplus, \bar{0})$ is a commutative monoid, $(K, \otimes, \bar{1})$ is a (possibly non-commutative) monoid, $\otimes$ distributes over $\oplus$, and $\bar{0} \otimes x = x \otimes \bar{0} = \bar{0}$ for any $x \in K$. Informally, a semiring is a ring that may lack negation. In the following, we will often call *weights* the elements of a semiring.

A formal power series $S : x \mapsto (S, x)$ is a function from a free monoid Σ^* to a semiring K. Rational power series are those formal power series that can be built by rational operations (concatenation, sum and Kleene closure) from the *singleton* power series given by $(S, x) = k$, $(S, y) = \bar{0}$ if $x \neq y$ for $x \in \Sigma^*$, $k \in K$. The rational power series are exactly those formal power series that can be represented by weighted automata [35].

Weighted automata are a generalization of the notion of automaton: each transition of a weighted automaton is assigned a weight in addition to the usual label(s). More formally, a weighted *acceptor* over a finite alphabet Σ and a weight semiring K is a finite directed graph with nodes representing states and arcs representing transitions in which each transition t is labeled with an input $i(t) \in \Sigma$ and a weight $w(t) \in K$. Furthermore, each state q has an *initial weight* $\lambda(q) \in K$ and a *final weight* $\rho(q) \in K$. In a weighted transducer, each transition t has also an output label $o(t) \in \Delta^*$ where Δ is the transducer's output alphabet. A state q is *initial* if $\lambda(q) \neq \bar{0}$, and *final* if $\rho(q) \neq \bar{0}$. [1]

A weighted acceptor A defines a rational power series $S(A)$ as follows. For each input string x, let $P(x)$ be the set of transition paths $p = t_1 \cdots t_{n_p}$ from an initial state i_p to a final state f_p such that $x = i(t_1) \cdots i(t_{n_p})$. Each such path assigns x the weight $w(p) = \lambda(i_p) \otimes (\bigotimes_j w(t_j)) \otimes \rho(f_p)$. A similar definition can be given for a weighted transducer T, except that $S(T)$ is now a rational power series over a semiring of rational power series, those mapping transducer output strings to weights [34].

Most of the algorithms of our library work with arbitrary semirings or with semirings from mathematically defined subclasses (closed semirings, K-closed semirings [20]). To instantiate the library for a particular semiring K, we just need to give computational representations for the semiring elements and operations. Library algorithms, for instance composition, ε-removal, determinization and minimization, work without change over different semirings because of their foundation in the theory of rational power series [18].

For example, the same power series determinization algorithm and code [18] can be used to determinize transducers [17], weighted transducers, weighted automata encountered in speech processing [23] and weighted automata using the probability operations. To do so, one just needs to use the algorithm with the string semiring $(\Sigma^* \cup \{\infty\}, \wedge, \cdot, \infty, \varepsilon)$ [21] in the case of transducers, with the semirings $(\mathbb{R}, +, \cdot, 0, 1)$ and $(\mathbb{R}_+, \min, +, \infty, 0)$ in the other cases, and with the cross product of the string

[1] For convenience of implementation, and without loss of generality (initial weights can be simulated with ε transitions), the automata supported by the library have a single initial state, with initial weight $\bar{1}$. Also, we allow the input label of a transition to be ε and restrict output labels to $\Delta \cup \{\varepsilon\}$ for practical reasons related to the efficient implementation of rational operations and composition. As is well known, the theory can be extended to cover those cases.

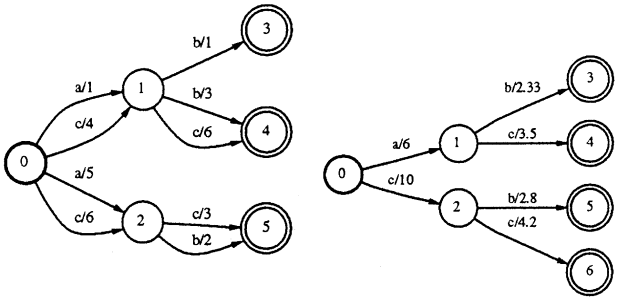

Fig. 1. Determinization over $(\mathbb{R}, +, \cdot, 0, 1)$.

semiring and one of these semirings in the case of weighted transducers. Fig. 1 shows a weighted acceptor over $(\mathbb{R}, +, \cdot, 0, 1)$ and its determinization.

3. Algorithms

3.1. Weighted automata algorithms

Although algorithms for weighted automata are closely related to their better-known unweighted counterparts, they differ in crucial details. One of the important features of our finite-state library is that most of its algorithms operate on general weighted automata and transducers.

We briefly outlined in the previous section the mathematical foundation for weighted automata, and how it allows us to write general algorithms that are independent of the underlying algebra. Owing to this generality, weights may be numbers, but also strings, sets, or even regular expressions. Depending on the algorithms, some restrictions apply to the semirings used. For instance, some algorithms require *commutative* semirings, meaning that $\otimes$ is commutative; others require *closed* semirings, in which infinite addition is defined and behaves like finite addition with respect to multiplication.

Shortest-paths algorithms play an essential role in applications, being used to find the "best" solution in the set of possible solutions represented by an automaton (for instance, the best string alignment or the best recognition hypothesis), as we shall see in Section 5.1. Therefore, we developed a general framework for single-source shortest-paths algorithms based on semirings that leads to a single generic algorithm [20]. This generic algorithm computes the single-source shortest distance when weights are numbers, strings, or subsets of a set. These different cases are related to the computation of minimal deterministic weighted automata [21].

Since the general framework for solving all pairs shortest-paths problems – closed semirings – is compatible with the abstract notion of weights we use, we were able to include an efficient version of the generic algorithm of Floyd-Warshall [1, 3] in our library. Using the same algorithm and code, we can provide the all-pairs shortest

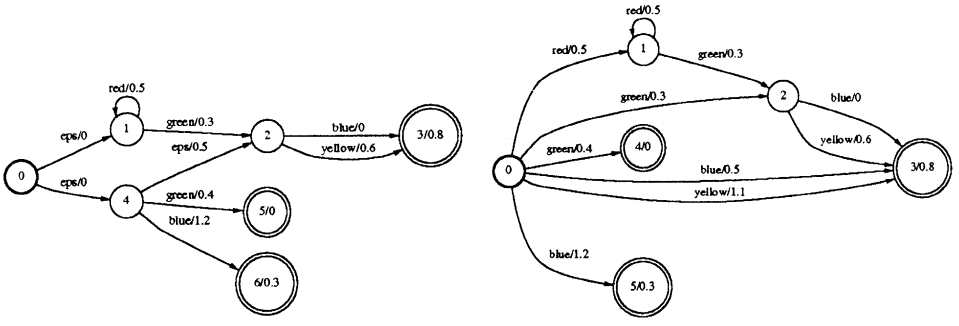

Fig. 2. Weighted automaton and its ε-removal.

distances when weights are real numbers representing, for example, probabilities, but also when they are strings or regular expressions. This last case is useful to generate efficiently a regular expression equivalent to a given automaton. The Floyd–Warshall algorithm is also useful in the general ε-removal algorithm we will now present as an example.

3.2. Example: ε-removal

Fig. 3 shows the pseudocode of a generic ε-removal algorithm for weighted automata. Given a weighted automaton M_i, the algorithm returns an equivalent weighted automaton M_o without ε-transitions. $\text{Trans}_M[s]$ denotes the set of transitions leaving state s in automaton M, $\text{Next}(t)$ denotes the destination state of transition t, $i(t)$ denotes its input label, and $w(t)$ its weight. Lines 1 and 2 extract from M_i the subautomaton M_ε containing all ε transitions in M_i and the subautomaton M_o containing all the non-ε transitions. Line 3 applies the general all-pairs shortest distance algorithm CLOSURE to M_ε to derive the ε-closure G_ε. The nested loops starting in lines 4, 5 and 6 iterate over all pairs of an ε-closure transition e and a non-ε transition t such that the destination of e is the source of t. Line 7 looks in M_o for a transition t' with label $i(t)$ from e's source from t's destination if it exists, or creates a new one with weight $\bar{0}$ if it does not. This transition is the result of extending t "backwards" with the M_i ε-path represented by ε-closure transition e. Its weight, updated in line 8, is the semiring sum of such extended transitions with a given source, destination and label.

In most speech-processing applications, the appropriate weight algebra is the *tropical semiring* [37]. Weights are positive real numbers representing negative logarithms of probabilities. Weights along a path are added; when several paths correspond to the same string, the weight of the string is the minimum of the weights of those paths. Fig. 2 illustrates the application of ε-removal to weighted automata over the tropical semiring. The example shows that the new algorithm generalizes the classical unweighted algorithm by ensuring that the weight of any string accepted by the automaton is preserved in the ε-free result.

```
1  M_ε ← M_i|_{ε}
2  M_o ← M_i|_{Σ*−{ε}}
3  G_ε ← CLOSURE(M_ε)
4  for p ← 1 to |V|
5  do for each e ∈ Trans_{G_ε}[p]
6     do for each t ∈ Trans_{M_i}[Next(e)] ∧ i(t) ≠ ε
7        do t' ← FINDTRANS(i(t), Next(t), Trans_{M_o}[p])
8           w(t') ← w(t') ⊕ w(t) ⊗ w(e)
```

Fig. 3. Pseudocode of the general ε-removal algorithm.

As noted before, the computation of the ε-closure requires the computation of the all-pairs shortest distances in M_ε. In the case of idempotent semirings such as the tropical semiring, the most efficient algorithm available is Johnson's algorithm which is based on the algorithms of Dijkstra and Bellman-Ford [3]. The running time complexity of Johnson's algorithm is $O(|Q|^2 \log |Q| + |Q||E|)$ when using Fibonacci heaps, but we use instead the more general but less efficient Floyd–Warshall algorithm because it supports non-idempotent closed semirings. When M_ε is acyclic, we use the linear time topological-sort algorithm, which also works with non-idempotent semirings [20].

Our implementation of the algorithm is in fact somewhat more complex: we first decompose M_ε into strongly connected components, apply the Floyd–Warshall algorithm to each component, and then apply the acyclic algorithm to the component graph of M_ε to compute the final result.

Our choice of the most general implementation was also guided by experimentation: in practice, each strongly connected component of M_ε is small relative to M_ε's overall size, and therefore the use of the Floyd–Warshall algorithm does not seriously impact efficiency.

3.3. Lazy algorithms

Most of the library's main functions have lazy implementations, meaning that their results are computed only as required by the operations using those results. Lazy execution is very advantageous when a large intermediate automaton is constructed in an application but only a small part of the automaton needs to be visited for any particular input to the application. For instance, in a speech recognizer, several weighted transducers – the language model, the dictionary, the context-dependent acoustic models – are composed into a potentially huge transducer, but only a very small part of it is searched when processing a particular utterance [30].

The main precondition for a function to have a lazy implementation is that the function be expressible as a *local* computation rule, in the sense that the transitions leaving a particular state in the result be determined solely by their source state and information from the function's arguments associated to that state. For instance, composition has a lazy implementation, as we will see in Section 3.4 below. Similarly,

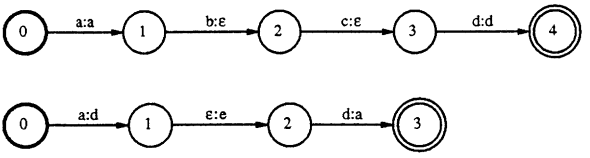

Fig. 4. Composition inputs.

union, concatenation and Kleene closure can be computed on demand, and so can determinization.

3.4. Example: lazy composition

Composition generalizes acceptor intersection. States in the composition $T_1 \circ T_2$ of T_1 and T_2 are identified with pairs of a state of T_1 and a state of T_2. Leaving aside transitions with ε inputs or outputs for the moment, the following rule specifies how to compute a transition of $T_1 \circ T_2$ from appropriate transitions of T_1 and T_2

$$(q_1 \xrightarrow{a:b/w_1} q_1' \quad \text{and} \quad q_2 \xrightarrow{b:c/w_2} q_2') \Rightarrow (q_1, q_2) \xrightarrow{a:c/(w_1 \otimes w_2)} (q_1', q_2'),$$

where $s \xrightarrow{x:y/w} t$ represents a transition from s to t with input x, output y and weight w. Clearly, this computation is local, and can thus be used in a lazy implementation of composition.

Transitions with ε labels in T_1 or T_2 add some subtleties to composition. In general, output and input ε's can be aligned in several different ways: an output ε in T_1 can be consumed either by staying in the same state in T_2 or by pairing it with an input ε in T_2; an input ε in T_2 can be handled similarly. For instance, the two transducers in Fig. 4 can generate all the alternative paths in Fig. 5. However, the single bold path is sufficient to represent the composition result, shown separately in Fig. 6. The problem with redundant paths is not only that they increase unnecessarily the size of the result, but also they fail to preserve *path multiplicity*: each pair of compatible paths in T_1 and T_2 may yield several paths in $T_1 \circ T_2$. If the weight semiring is not idempotent, that leads to a result that does not satisfy the algebraic definition of composition:

$$[\![T_1 \circ T_2]\!](u, w) = \bigoplus_v [\![T_1]\!](u, v) \otimes [\![T_2]\!](v, w).$$

We solve the path-multiplicity problem by mapping the given composition into a new composition

$$T_1 \circ T_2 \to T_1' \circ F \circ T_2'$$

in which F is a special *filter transducer* and the T_i' are versions of the T_i in which the relevant ε labels are replaced by special "silent transition" symbols ε_i. The bold path in Fig. 5 is the only one allowed by the filter in Fig. 7 for the input transducers in Fig. 4.

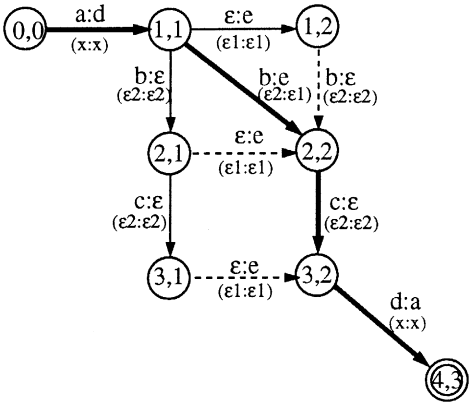

Fig. 5. Redundant composition paths.

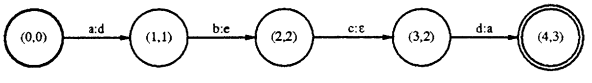

Fig. 6. Composition output.

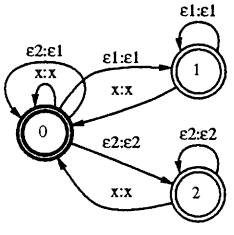

Fig. 7. Composition filter.

Clearly, all the operations involved in the filtered composition are local, therefore they can be performed on demand, without needing to perform explicitly the replacement of T_i by T_i'. More details on filtered composition can be found elsewhere [22, 27].

4. Software design

Our library was designed to meet two important requirements:

- Algorithms that operate on automata should do so only through abstract accessor and mutator operations, which in turn operate on the internal representations of those automata.

- Algorithms that operate on weights should do so solely through abstract operations that implement the weight semiring.

We motivate and describe these two requirements below. Furthermore, the demanding nature of our applications imposes the constraint that these abstractions add little computational burden compared to more specialized architectures.

4.1. Finite-state objects

Requiring algorithms to operate on automata solely through abstract accessors and mutators has three benefits: it allows the internal representation of automata to be hidden, it allows *generic* algorithms that operate on multiple finite-state representations and it provides the mechanism for creating and using lazy implementations of algorithms. To illustrate these points, consider the core accessors supported by all automata classes in the library:

- *fsm*.`start()`, which returns the initial state of *fsm*;
- *fsm*.`final`(*state*), which returns the final weight of *state* in *fsm*;
- *fsm*.`arcs`(*state*), which returns an iterator over the transitions leaving *state* in *fsm*. The iterator is itself an object supporting the `next` operation, which returns (a pointer to) each transition from *state* in turn.

A state is specified by an integer index; a transition is specified by a structure containing an input label, an output label, a weight and a next state index.[2]

Clearly, a variety of automata implementations meet this core interface. As a simple example, the transitions leaving a state could be stored in arrays or in linked lists. By hiding the automaton's implementation from its user we gain the usual advantages of separating interfaces from implementations: we can change the representation as we wish and, so long as we do not change the object interface, the code that uses it still runs.

In fact, it proves very useful to have multiple automata implementations in the same library. For example, one class of automata in the library provides mutating operations such as adding states and arcs, by using an extensible vector representation of states and transitions that supports efficient appends. Another class, for read-only automata, uses fixed state and transition arrays that can be efficiently memory-mapped from files. A third class, also read-only, stores states and transitions in a compressed form to save space, and uncompresses them on demand when they are accessed.

Our algorithms are written generically, in that they assume that automata support the core operations above and as little else as necessary. For example, some classes of automata support the *fsm*.`numstates()` operation that returns *fsm*'s number of states, while others do not (we will see an example in a moment). Where possible and reasonably efficient, we write our algorithms to avoid using such optional operations. In this way, they will work on any automaton class. On the other hand, if it is really

[2] Using integer indices allows referring to states that may not have yet been constructed in automata being created by lazy algorithms.

necessary to use *fsm*.`numstates()`, then at least all automata classes that support that operation operation will work.[3,4]

The restricted set of core operations above was motivated by the need to support lazy implementations of algorithms. In particular, the operations are local if we accept the convention that no state should be visited that has not been discovered from the start state. Thus the automaton object that lies behind this interface need not have a static representation. For example, we can implement the result of the composition of two automata A and B as a delayed composition automaton $C = \texttt{FSMCompose}(A, B)$. When C.`start()` is called, the start state can be constructed on demand by first calling A.`start()` and B.`start()` and then pairing these states and hashing the pair to a new constructed state index, which C.`start()` returns. Similarly, C.`final()` and C.`arcs()` can be computed on-demand by first calling these operations on A and B and then constructing the appropriate result for C to return. If we had included `numstates` as a core operation, the composition would have to be fully expanded immediately to count its number of states. Since a user might do this inadvertently, we do not provide that operation for automata objects resulting from composition.[5] The core operations, in fact, can support lazy automata with an infinite number of states, so long as only a finite portion of such automata is traversed.

To achieve the required efficiency for the above interface, we ensure that each call to the transition iterator involves nothing more than a pointer increment in the automata classes intended for demanding applications such as speech recognition. Since most of the time used for automata operations in those applications is spent iterating over the transitions leaving various states, that representation is usually effective.

4.2. Weight objects

As mentioned earlier, many of the algorithms in our library will work with a variety of weight semirings. Our design encourages writing algorithms over the most general semiring by making the weights an abstract type with suitable addition and multiplication operations and identity elements. In this way, we can switch between, say, the tropical semiring and the probability semiring by just using a different implementation of the abstract type. For efficiency, the weight operations are represented by macros in our C version and by inline member functions in the C++ version under development.

[3] For those that do not, our current C implementation will issue a run-time error, while run-time type-checking can be used to circumvent such errors. In our new C++ version, we will use compile-time type-checking where possible

[4] This design philosophy has some similarities with that of other modern software toolkits such as the C++ Standard Template Library [26]

[5] The user can always copy this lazy automaton into an instance of a static automata class that supports the `numstates` operation. In other words, we favor explicit conversions to implicit ones.

4.3. Coverage

The library operates on weighted transducers; weighted acceptors are represented as restrictions of the identity transducer to the support of the acceptor. In our chosen representation, weighted automata have a single initial state; whether a state is accepting or not is determined by the state's final weight. The library includes:

Rational operations: union, concatenation, Kleene closure, reversal, inversion and projection.

Composition: transducer composition [22], and acceptor intersection, as well as taking the difference between a weighted acceptor and an unweighted DFA.

Equivalence transformations: ε-elimination, determinization [17, 18] and minimization for unweighted (both the general case [1] and the more efficient acyclic case [28]) and weighted acceptors and transducers [15, 18], removal of inaccessible states and transitions.

Search: best path [20], n-best paths, pruning (remove all states and transitions that occur only on paths of weight greater by a given threshold than the best path).

Representation and storage management: create and convert among automata representations with different performance tradeoffs; also, as discussed in Section 3.3, many of the library functions can have their effects delayed for lazy execution, and functions are provided to cache and force delayed objects, inspired by similar features in lazy functional programming.

In addition, a comprehensive set of support functions is provided to manipulate the internal representations of automata (for instance, topological sorting), for converting between internal and external representations, and for accessing and mutating the components of an automaton (states, transitions, initial state and accepting weights).

For convenient experimentation, each of the library's main functions has a Unix shell-level counterpart that operates between external automata representations, allowing the expression of complex operations on automata as shell pipelines. The concrete example in the next section is presented in terms of those commands for simplicity.

These Unix shell-level commands are available for download for a variety of computer architectures from the AT&T Labs-Research web site [40] along with documentation, tutorials, and exercises.

5. Language processing applications

As noted in Section 1, finite-state methods have been used very successfully in a variety of language-processing applications. However, until we developed our library, those applications had not included speech recognition.

Current speech-recognition systems rely on a variety of probabilistic finite-state models, for instance n-gram language models [29], multiple-pronunciation dictionaries [13], and context-dependent acoustic models [12]. However, most speech recognizers do not take advantage of the shared properties of the information sources they use. Instead, they rely on special-purpose algorithms for specific representations. That means that the

recognizer has to be rewritten if representations are changed for a new application or for increased accuracy or performance. Experiments with different representations are therefore difficult, as they require changing or even completely replacing fairly intricate recognition programs.

This situation is not too different from that in programming-language parsing before `lex` and `yacc` [2]. Furthermore, specialized representations and algorithms preclude certain global optimizations based on the general properties of finite-state models. Again, the situation is similar to the lack of general methods in programming-language parsing before the development of the theory of deterministic context-free languages and of general grammar optimization techniques based on it.

As noted in Section 1, in speech recognition it is essential that alternative ways of generating or transforming a string be weighted by the likelihood of that generation or transformation. Therefore, the crucial step in applying general finite-state techniques to speech recognition problems was to move from regular languages to rational power series, and from unweighted to weighted automata.[6] The main challenges in this move have been the generalization of core algorithms to the weighted case, and their implementation with the degree of efficiency required in speech recognition.

5.1. Simple example: alignment

As a simple example of the use of the library in speech processing, we show how to find the best alignment between two strings using a weighted edit distance, which can be applied for instance to finding the best alignment between the dictionary phonetic transcription of a word string and the acoustic (phone) realization of the same word string, as exemplified in Fig. 8. Fig. 9 shows a domain-dependent table of insertion, deletion and substitution weights between phonemes and phones. In a real application, those weights would be derived automatically from aligned examples using a suitable machine-learning method [13, 31]. The minimum edit distance between two strings can be simply defined by the recurrences

$$
\begin{aligned}
&d(\mathbf{a}^0, \mathbf{b}^0) = 0,\\
&d_s(\mathbf{a}^i, \mathbf{b}^j) = d(\mathbf{a}^{i-1}, \mathbf{b}^{j-1}) + w(a_i, b_j) && \text{(substitution)},\\
&d_d(\mathbf{a}^i, \mathbf{b}^j) = d(\mathbf{a}^{i-1}, \mathbf{b}^j) + w(a_i, \varepsilon) && \text{(deletion)},\\
&d_i(\mathbf{a}^i, \mathbf{b}^j) = d(\mathbf{a}^i, \mathbf{b}^{j-1}) + w(\varepsilon, b_j) && \text{(insertion)},\\
&d(\mathbf{a}^i, \mathbf{b}^j) = \min\{d_s(\mathbf{a}^i, \mathbf{b}^j), d_d(\mathbf{a}^i, \mathbf{b}^j), d_i(\mathbf{a}^i, \mathbf{b}^j)\}\ .
\end{aligned}
$$

The possible one symbol edits (insertion, deletion or substitution) and their weights can be readily represented by a one-state weighted transducer. If the transducer is in file `T.fst` and the strings to be aligned are represented by acceptors `A.fsa` and `B.fsa`, the best alignment is computed simply by the shell command

```
fsmcompose A.fsa T.fst B.fsa | fsmbestpath >C.fst
```

[6] Weighted acceptors and transducers have also been used in image processing [4].

Baseform	Phone	Word
p	pr	purpose
er	er	
p	pcl	
-	pr	
ax	ix	
s	s	
ae	eh	and
n	n	
d	-	
r	r	respect
ih	ix	
s	s	
p	pcl	
-	pr	
eh	eh	
k	kcl	
t	tr	

Fig. 8. String alignment.

Baseform a_i	Phone b_j	Weights $w(a_i, b_j)$	Type
ae	eh	1	substitution
d	ε	2	deletion
ε	pr	1	insertion

Fig. 9. Weighted edit distance.

Abbreviated examples of the inputs and outputs to this command are shown in Fig. 10.

The correctness of this implementation of minimum edit distance alignment depends on the use of suitable weight combination rules in automata composition, specifically those of the tropical semiring, which was discussed in Section 3.2.

Alignment by transduction can be readily extended to situations in which edits involve longer strings or are context-dependent, as those shown in Fig. 11. In such cases, states in the edit transducer encode appropriate context conditions. Furthermore, a set of weighted edit rules like those in Fig. 11 can be directly compiled into an appropriate weighted transducer [25].

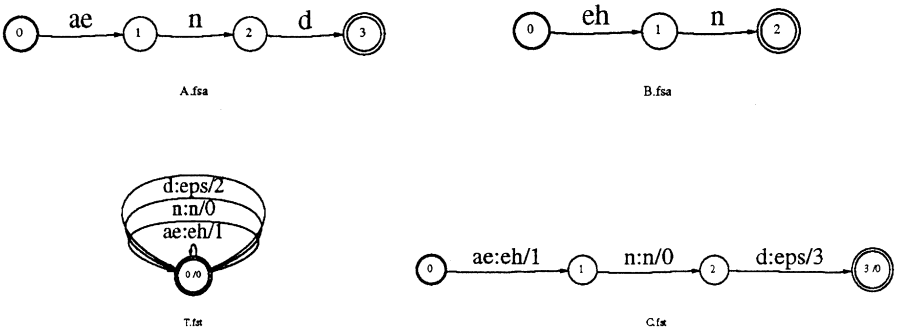

Fig. 10. Alignment automata.

Baseform(s) a_i	Phone(s) b_j	Weights $w(a_i, b_j)$	Type
p	pcl pr	1	expansion
eh m	em	3	contraction
r eh	ax r	2	transposition
t/V'___V	dx	0	context-dependency

Fig. 11. Generalized weighted edit distance.

6. Conclusion

We presented a very general finite-state library based on the notions of semiring and of rational power series, which allowed us to use the same code for a variety of different applications requiring different semirings. The current version of the library is written in C, with the semiring operations defined as macros. Our new version is being written in C++ to take advantage of templates to support more general transition labels and multiple semirings in a single application.

Our experience shows that it is possible and in fact sometimes easier to implement efficient generic algorithms for a class of semirings than to implement specialized algorithms for particular semirings. Similarly, lazy versions of algorithms are often easier to implement than their traditional counterparts.

We tested the efficiency of our library by building competitive large-vocabulary speech recognition applications involving very large automata ($>10^6$ states, $>10^7$ transitions) [23, 24]. The library is being used in a variety of speech recognition and speech synthesis projects at AT&T Labs and at Lucent Bell Laboratories.

References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974

[2] A.V. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques and Tools, Addison-Wesley: Reading, MA, 1986.
[3] T. Cormen, C. Leiserson, R. Rivest, Introduction to Algorithms, The MIT Press, Cambridge, MA, 1992.
[4] K. Culik II, J. Kari, Digital images and formal languages, in: G. Rozenberg, A. Salomaa (Eds.), Handbook of Formal Languages, Springer, Berlin, 1997, pp. 599–616,
[5] M. Gross, The Use of Finite Automata in the Lexical Representation of Natural Language, Lecture Notes in Computer Science, vol. 377, Springer, Berlin, 1989.
[6] M. Gross, D. Perrin (Eds.), Electronic Dictionnaries and Automata in Computational Linguistics, Lecture Notes in Computer Science, vol. 377, Springer, Berlin, 1989.
[7] R.M. Kaplan, M. Kay, Regular Models of Phonological Rule Systems, Comput. Linguistics 20 (3) (1994) 331–378.
[8] F. Karlsson, A. Voutilainen, J. Heikkila, A. Anttila, Constraint Grammar, A language-Independent System for Parsing Unrestricted Text, Mouton de Gruyter, 1995.
[9] L. Karttunen, The Replace Operator, 33rd Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, 1995, pp. 16–23, Distributed by Morgan Kaufmann Publishers, San Francisco, CA.
[10] L. Karttunen, R.M. Kaplan, A. Zaenen, Two-level Morphology with Composition, in Proc. 15th Internat. Conf. on Computational Linguistics (COLING'92), Nantes, France, COLING, 1992.
[11] K. Koskenniemi, Finite-state parsing and disambiguation, Proc. 13th Internat. Conf. on Computational Linguistics (COLING'90), Helsinki, Finland, COLING, 1990.
[12] K.-F. Lee, Context dependent phonetic hidden Markov models for continuous speech recognition, IEEE Trans. Acouts. Speech Signal Process. 38 (4) (1990) 599–609.
[13] A. Ljolje, M.D. Riley, Optimal speech recognition using phone recognition and lexical access, in Proc. ICSLP, Banff, Canada, October 1992, pp. 313–316.
[14] M. Mohri, Compact representations by finite-state transducers, in 32nd Meeting of the Association for Computational Linguistics (ACL 94), Proc. Conf. Las Cruces, NM, ACL, 1994.
[15] M. Mohri, Minimization of Sequential Transducers, Lecture Notes in Computer Science, vol. 807, Springer, Berlin, 1994.
[16] M. Mohri, Syntactic Analysis by local grammars automata: an efficient algorithm, in Proc. Internat. Conf. on Computational Lexicography (COMPLEX 94), Linguistic Institute, Hungarian Academy of Science, Budapest, Hungary, 1994.
[17] M. Mohri, On some applications of finite-state automata theory to natural language processing, J. Natural Language Eng. 2 (1996) 1–20.
[18] M. Mohri, Finite-State Transducers in Language and Speech Processing, Comput. Linguistics 23 (2) (1997) 269–311.
[19] M. Mohri, On the use of sequential transducers in natural language processing, in: E. Roche, Y. Schabes (Eds.), Finite-State Language Processing, MIT Press, Cambridge, MA, 1997, 355–382.
[20] M. Mohri, General Algebraic Framework and Algorithms for Shortest Distance Problems. Technical Memorandum, AT&T Labs-Research, 981210-TM, 1998.
[21] M. Mohri, Minimization algorithms for sequential transducers, Theoret. Comput. Sci. (2000) to appear.
[22] M. Mohri, F.C.N Pereira, M. Riley, Weighted automata in text and speech processing, in ECAI-96 Workshop, Budapest, Hungary, ECAI, 1996.
[23] M. Mohri, M. Riley, Weighted determinization and minimization for large vocabulary speech recognition, in Eurospeech'97, Rhodes, Greece, 1997.
[24] M. Mohri, M. Riley, D. Hindle, A. Ljolje, F.C. N. Pereira, Full expansion of context-dependent networks in large vocabulary speech recognition, in Proc. ICASSP'98, IEEE, New York, 1998.
[25] M. Mohri, R. Sproat, An efficient compiler for weighted rewrite rules, in 34th Meeting of the Association for Computational Linguistics (ACL 96), Proc. Conf., Santa Cruz, Ca, ACL, 1996.
[26] D. Musser, A. Saini, STL Tutorial and Reference Guide, Addison-Wesley, Reading, MA, 1996.
[27] F.C.N. Pereira, M.D. Riley, Speech recognition by composition of weighted finite automata, in: E. Roche, Y. Schabes (Eds.), Finite-State Language Processing, MIT Press, Cambridge, Ma, 1997, pp. 431–453.
[28] D. Revuz, Minimisation of acyclic deterministic automata in linear time, Theoret. Comput. Sci. 92 (1992) 181–189.

[29] G. Riccardi, E. Bocchieri, R. Pieraccini, Non-deterministic stochastic language models for speech recognition, in Proc. IEE Internat. Conf. on Acoustics, Speech and Signal Processing, vol. 1, IEEE, New York, 1995, pp. 237–240
[30] M. Riley, F. Pereira, M. Mohri, Transducer composition for context-dependent network expansion, in Eurospeech'97, Rhodes, Greece, 1997.
[31] E. Ristad, P. Yianilos, Finite growth models, Technical Report CS-TR-533-96, Department of Computer Science, Princeton University, 1996.
[32] E. Roche, Analyse Syntaxique Transformationnelle du Français par Transducteurs et Lexique-Grammaire, Ph.D. Thesis, Université Paris 7, 1993.
[33] E. Roche, Two parsing methods by means of finite state tansducers, in Proc. 16th Internat. Conf. on Computational Linguistics (COLING'94), Kyoto, Japan, COLING, 1994.
[34] A. Salomaa, M. Soittola, Automata-Theoretic Aspects of Formal Power Series, Springer, New York, 1978.
[35] M.P. Schützenberger, On the definition of a family of automata, Inform. Control 4 (1961).
[36] M. Silberztein, Dictionnaires électroniques et analyse automatique de textes: le système INTEX, Masson, Paris, France, 1993.
[37] I. Simon, Limited subsets of a free monoid, in Proc. 19th Annual Symp. on Foundation of Computer Science, 1978, pp. 143–150.
[38] R. Sproat, Morphology and Computation, The MIT Press, Cambridge, MA, 1992.
[39] R. Sproat, A finite-state architecture for tokenization and grapheme-to-phoneme conversion in multilingual text analysis, in Proc. ACL SIGDAT Workshop, Dublin, Ireland, ACL, 1995.
[40] M. Mohri, F. Pereira, M. Riley, FSM Library – General-Purpose Finite. State Machine Software tools, http://www.research.att.com/tools/fsm, 1998.