A Rational Design for a Weighted Finite-State Transducer Library

Mehryar Mohri, Fernando Pereira, and Michael Riley

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

1 Overview

We describe the design principles and main algorithms for an object-oriented library for *weighted finite-state transducers,* which are finite automata in which each transition has an output and a weight as well as the more familiar input. The main goal of the library is to provide algorithms and representations for all the symbolic processing components (language models, dictionaries, acoustic realization rules, word lattices) in large-vocabulary speech recognition systems. This goal leads to several requirements: generality, to support the representation and use of the various information sources in speech recognition; modularity, to allow rapid experimentation with different representations of speech recognition tasks; and efficiency, to support competitive large-vocabulary recognition. Rational power series provide the theoretical foundation for the library by giving the semantics for the objects and operations in the library and by creating the opportunity for optimizations (on-demand composition, determinization and minimization) that are not available in more "ad hoc" speech recognition frameworks. The generality of the library has made it also valuable in other language-processing applications, such as word segmentation for Chinese text [25].

1.1 Design Rationale

Current speech-recognition systems rely on a variety of probabilistic finite-state models, for instance n-gram language models [21], multiple-pronunciation dictionaries [11], and context-dependent acoustic models [10]. 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.

In some other areas of language processing, especially dictionaries, morphology and local parsing, finite-state techniques have been used with great success [7,13]. Complex finite-state mappings between strings can be represented with regular expressions over elementary *transductions* (input-output mappings) together with relational *composition* of transductions. Convenient notation extending regular expressions with composition and useful derived forms have also been developed [8]. It might thus be thought that those finite-state tools would apply directly to the problem of specifying and combining the multiple information sources used in a speech recognizer. However, 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. That is, instead of languages and transductions we have to work with weighted languages and weighted transductions, and thus with finite automata with weighted transitions.[1] Weighted languages are just *formal power series* over appropriate *semirings,* and weighted transductions can be interpreted in a similar way [23, 6, 9, 3]. We were thus led to develop a library supporting the representation and use of weighted finite-state representations of speech-recognition models. Using the library, each weighted language or transduction is represented as an appropriate weighted acceptor or transducer, and model combination is done by calls to library functions.

The adjective "rational" in the title of this paper is used ambiguously to refer both to the use of the theory of rational power series as a foundation for the library, and to the design approach, in which each object and function has a well-defined mathematical semantics. At the level of the rational operations and composition, the library is *compositional* — the meaning as a language or transduction of the result of a function application depends only on the meanings of the function's arguments. Other functions in the library depend on or manipulate representations — weighted automata — but they are compositional at the automata level, in that they operate uniformly on any automata objects providing suitable accessor and mutator methods.

Algorithms on weighted automata have strong similarities with their better-known unweighted counterparts, but the proper treatment of weights introduces various subtleties that we shall describe later. Furthermore, the size of symbol sets and automata arising in large-vocabulary recognition require careful implementation techniques even for standard algorithms. For example, iterating over the symbol set in the standard DFA minimization algorithm [1] is impractical for sparse DFAs if the symbol set is large, as is the case in language models for large-vocabulary recognition.

[1] Weighted acceptors and transducers have also been used in image processing [5].

1.2 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 *accepting weight*. The library includes:

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

Composition: transducer composition [17], and acceptor intersection, as well as taking the difference between a weighted acceptor and an unweighted DFA;

Equivalence transformations: ϵ-elimination, determinization [14,15] and minimization for unweighted (both the general case [1] and the more efficient acyclic case [20]) and weighted acceptors and transducers [12,15], removal of inaccessible states and transitions;

Search: best path, 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 we will discuss later, many of the library functions can have their effects *delayed* for on-demand execution, and functions are provided to cache and force delayed objects, inspired by similar functions for lazy evaluation in functional programming languages.

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, including graphical ones, 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 example that follows is presented in terms of those commands for simplicity.

1.3 Simple Example: Alignment

As a simple example of the use of the library, 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. Figure 2 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

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. 1. String Alignment

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

Fig. 2. Weighted Edit Distance

suitable machine-learning method [11, 22]. The minimum edit distance between two strings can be simply defined by the recurrences

$$d(\mathbf{a}^0, \mathbf{b}^0) = 0$$

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

$$d(\mathbf{a}^i, \mathbf{b}^j) = \min\{d_s(\mathbf{a}^i, \mathbf{b}^j) + d_d(\mathbf{a}^i, \mathbf{b}^j) + d_i(\mathbf{a}^i, \mathbf{b}^j)\}$$

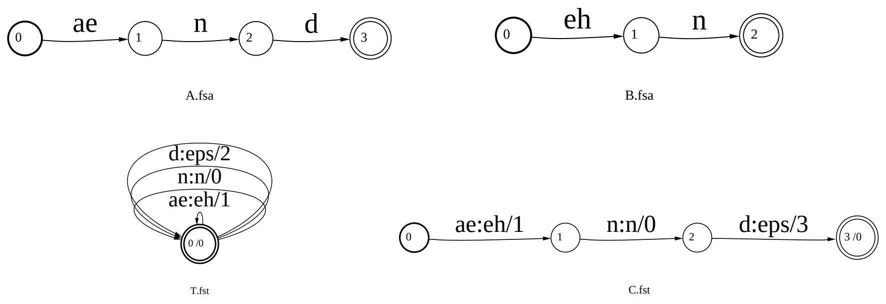

Fig. 3. Alignment Automata

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
```

Abbreviated examples of the inputs and outputs to this command are shown in Figure 3.

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* [24], which will be discussed more fully in the next section.

Alignment by transduction can be readily extended to situations in which edits involve longer strings or are context-dependent, as those shown in Figure 4. In such cases, states in the edit transducer encode appropriate context conditions. Furthermore, a set of weighted edit rules like those in Figure 4 can be directly compiled into an appropriate weighted transducer [18].

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. 4. Generalized Weighted Edit Distance

2 Weighted 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 shall outline later the mathematical foundation for weighted automata, and how it allows us to write general algorithms that are independent of the underlying algebra. For the moment, we note that weights may not be numbers, but they may instead be strings, sets, or even regular expressions. In each case, the additive and multiplicative operations $\oplus$ and $\otimes$ on weights need to be chosen to form a *semiring*: $\oplus$ must be commutative with a neutral element $\mathbf{0}$, $\otimes$ must have a neutral element $\mathbf{1}$, $\otimes$ must distribute on the left and on the right with respect to $\oplus$, and $\mathbf{0} \otimes a = a \otimes \mathbf{0} = \mathbf{0}$. For some algorithms the semiring must be *commutative*, meaning that $\otimes$ is commutative. Finally, some algorithms require *closed* semirings, in which infinite addition is defined and behaves as 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 saw in the alignment example given earlier. 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 that generic algorithm [1,4] in our library. Using the same algorithm and code, we can provide the all-pairs shortest 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.

In a similar way we defined a general framework for single-source shortest-paths algorithms based on semirings that leads to a generic algorithm [16]. This generic algorithm computes the single-source shortest distance when weights are numbers, strings, or subsets of a set. These different cases are useful in computing minimal deterministic weighted automata. In most speech-processing applications, the appropriate weight algebra is the *tropical semiring*. Weights are positive real numbers, representing negative logarithms of probabilities. Weights

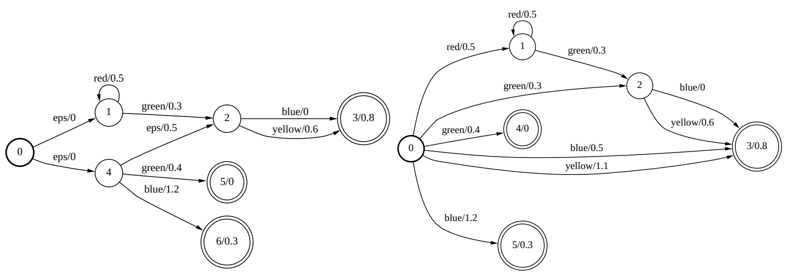

Fig. 5. Weighted automaton and its ϵ-removal

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.

2.1 Example: ϵ Removal

```
1 $M_\epsilon \leftarrow M_i|_{\{\epsilon\}}$
2 $M_o \leftarrow M_i|_{\Sigma-\{\epsilon\}}$
3 $G_\epsilon \leftarrow$ CLOSURE$(M_\epsilon)$
4 for $p \leftarrow 1$ to $|V|$
5   do for each $t_\epsilon \in \mathrm{Trans}_{G_\epsilon}[p]$
6        do for each $t_i \in \mathrm{Trans}_{M_i}[\mathrm{Next}(t_\epsilon)] \wedge l(t_i) \neq \epsilon$
7             do $t_o \leftarrow$ FINDTRANS$(l(t_i), \mathrm{Next}(t_i), \mathrm{Trans}_{M_o}[p])$
8                  $\lambda(t_o) \leftarrow \lambda(t_o) \oplus \lambda(t_i) \otimes \lambda(t_\epsilon)$
```

Fig. 6. Pseudocode of the general ϵ-removal algorithm.

Figure 6 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. $\mathrm{Trans}_M[s]$ denotes the set of transitions leaving state s in automaton M, $\mathrm{Next}(t)$ denotes the destination state of transition t, $l(t)$ denotes its label, and $\lambda(t)$ 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 t_ϵ and a non-ϵ transition t_i such that the destination of t_ϵ is the source of t_i. Line 7 looks in M_o for a transition t_o with label $l(t_i)$ from t_ϵ's source from t_i's destination if it exists, or creates a new one with weight **0** if it does not. This transition is the result of extending t_i "backwards" with the M_i ϵ-path represented by ϵ-closure transition t_ϵ. Its weight, updated in line 8, is the semiring sum of such extended transitions with a given source, destination and label.

Figure 5 illustrates the application of ϵ-removal to weighted automata over the tropical semiring. The example shows that the new algorithm is more general than the classical one because ϵ-transitions with non-zero weights are dealt with correctly.

3 On-Demand Algorithms

As noted earlier, most of the library's main functions have *lazy* implementations, meaning that their results are computed *on demand*, only as required by the operations using those results. On-demand 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.

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 3.1 below. Similarly, union, concatenation and Kleene closure can be computed on demand, and so does determinization. On the other hand, algorithms that require traversing an automaton both in the forward and in the backward directions, such as the standard DFA minimization algorithm or any algorithm based on relaxation techniques (for instance, strongly connected components, coaccessibility, best path through single-source shortest paths methods) do not have lazy implementations.

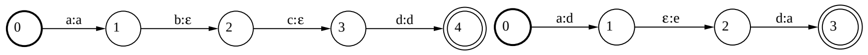

Fig. 7. Composition Inputs

3.1 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

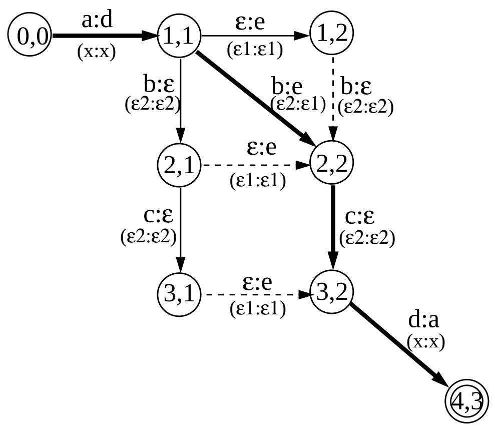

Fig. 8. Redundant Composition Paths

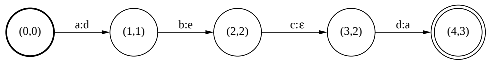

Fig. 9. Composition Output

specifies how to compute a transition of $T_1 \circ T_2$ from appropriate transitions of T_1 and T_2

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

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 Figure 7 can generate all the alternative paths in Figure 8. However, the single bold path is sufficient to represent the composition result, shown separately in Figure 9. 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:

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

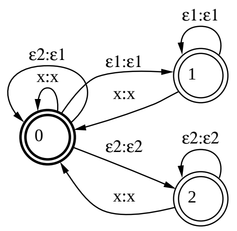

Fig. 10. Composition Filter

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

$$T_1 \circ T_2 \rightarrow T_1' \circ F \circ T_2'$$

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 Figure 8 is the only one allowed by the filter in Figure 10 for the input transducers in Figure 7.

Clearly, all the operations involved in the filtered composition are local, therefore they can be performed on demand, without needing perform explicitly the replacement of T_i by T_i'.

4 Software Design

Our library was designed to meet two important requirements:

1. Algorithms that operate on automata should do it only through abstract accessor and mutator operations, which in turn operate on the internal representations of those automata.
2. Algorithms that operate on weights should do it 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: (a) it allows the internal representation of automata to be hidden, (b) it allows *generic* algorithms that operate on multiple finite-state representations and (c) it provides the mechanism for creating and using on-demand 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, there are a variety of automata implementations that 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 advantage – 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 automaton in the library, which supports mutating operations such as adding states and arcs, uses an extensible vector representation of states and transitions that admits efficient appends. Another class, which is immutable, has a fixed array of states and transitions that admits memory-mapping from files. A third class, also immutable, 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, 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 necessary to use *fsm*.`numstates`(), then at least all automata classes that support that operation will work.[3] This design philosophy is similar in some ways to other modern software toolkits such as the C++ Standard Template Library [19].

The restricted set of core operations above was motivated by the need to support on-demand implementations of algorithms. In particular, all of those 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

[2] Using integer indices allow referring to states that may not have yet been constructed in automata being created by on-demand algorithms.

[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.

example, we can implement the result of the composition of two automata A and B as a delayed composition automaton C = `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.[4] The core operations, in fact, can support on-demand 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 insure that each call to the transition iterator involves nothing more than a pointer increment for the automata classes intended to be used in the core of demanding applications such as speech recognition. Since most of the time operating on an automaton in those applications is spent sequencing through the transitions leaving various states, that 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 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 replacing the weight definitions. For efficiency, these operations are represented by macros in our C version and by inline member functions in the C++ version under development.

5 Generality

In our design, we followed two major principles: generality and minimality. Our motivation is to provide the most general algorithms that can be useful in a variety of applications, and to avoid duplication of code that would increase the chance of errors and make the code harder to understand.

One of the main aspects of software engineering consists of separating data and programs. We suggest a mathematical analogue of this principle: the separation of algebra and algorithms. In other words, our algorithms are designed to work in as general an algebraic structure as possible.

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

This generality is possible thanks to the algebraic concepts of *rational power series* and semirings. We have already introduced semirings. Rational power series are functions that map a free monoid to a semiring. As shown by [23], rational power series are exactly those functions that can be represented by weighted automata. 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. More formally, a weighted automaton A is a 6-tuple $A = (\Sigma, K, Q, E, \lambda, \rho)$ in which Σ is a finite alphabet, $(K, \oplus, \otimes, 1, 0)$ is a semiring, Q is a finite set of states, $E \subseteq Q \times \Sigma \times K \times Q$ is a finite set of transitions, $\lambda : I \to K$ is the initial weight function and $\rho : F \to K$ the final weight function.[5]

A weighted automaton is used as follows. Given an input string x, each path from the initial state to a final state whose label sequence matches x will assign to x the semiring product $\otimes$ of the weights of the path transitions and the final weight of the final state of the path. The total weight for the input string x is the semiring sum $\oplus$ of the weights assigned to x by all the paths matching x.

We define the morphism lab from E^* to $\Sigma^* \times \Delta^*$ by

$$\forall t = (p, x, y, q) \in E,\ lab(t) = (x, y)$$

$lab(t)$ is the *label* of a transition $t \in E$. We denote by $\Pi(i, f)$ the set of all paths from a state i to f. Then, a weighted automaton A realizes the rational power series $S(A)$ defined by:

$$\forall x \in \Sigma^*, (S(A), x) = \bigoplus_{(x,y) \in lab(\Pi(i,f)), (i,f) \in I \times F} (\lambda(i) \otimes y \otimes \rho(f))$$

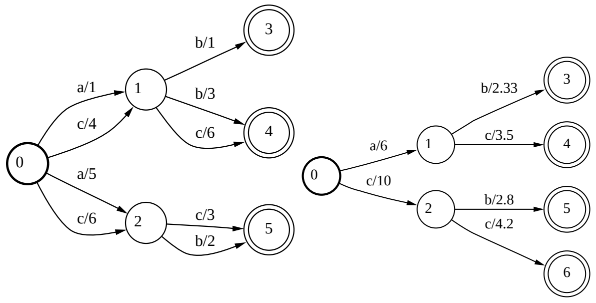

Fig. 11. Determinization over $(\mathcal{R}, +, \cdot, 0, 1)$.

[5] For historical and practical reasons, we use single initial states rather than an initial weight function; it's easy to see that this makes no substantive difference.

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

The same power-series determinization algorithm and code [15] can be used to determinize transducers, weighted automata encountered in speech processing and weighted automata using the probability operations. To do so, one just needs to use the algorithm with the string semiring $(\Sigma^* \cup \{\infty\}, \wedge, \cdot, \infty, \epsilon)$ in the case of transducers, and with the semirings $(\mathcal{R}, +, \cdot, 0, 1)$ and $(\mathcal{R}_+, \min, +, \infty, 0)$ in the other cases. For example, Figure 11 shows a weighted acceptor over $(\mathcal{R}, +, \cdot, 0, 1)$ and its determinization.

6 Conclusion

We presented a very general finite-state library based on the notions of semiring and of rational power series. We were thus able 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). 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, and J. D. Ullman. *The design and analysis of computer algorithms.* Addison Wesley: Reading, MA, 1974.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley: Reading, MA, 1986.
3. J. Berstel and C. Reutenauer. *Rational Series and Their Languages.* Springer-Verlag: Berlin-New York, 1988.
4. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* The MIT Press: Cambridge, MA, 1992.
5. K. Culik II and J. Kari. Digital images and formal languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, pages 599–616. Springer, 1997.

6. S. Eilenberg. *Automata, Languages and Machines*, volume A. Academic Press, 1974-1976.
7. R. M. Kaplan and M. Kay. Regular models of phonological rule systems. *Computational Linguistics*, 20(3), 1994.
8. L. Karttunen. The replace operator. In *33rd Annual Meeting of the Association for Computational Linguistics*, pages 16–23. Association for Computational Linguistics, 1995. Distributed by Morgan Kaufmann Publishers, San Francisco, California.
9. W. Kuich and A. Salomaa. *Semirings, Automata, Languages.* Number 5 in EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 1986.
10. K.-F. Lee. Context dependent phonetic hidden Markov models for continuous speech recognition. *IEEE Trans. ASSP*, 38(4):599–609, Apr. 1990.
11. A. Ljolje and M. D. Riley. Optimal speech recognition using phone recognition and lexical access. In *Proceedings of ICSLP*, pages 313–316, Banff, Canada, Oct. 1992.
12. M. Mohri. Minimization of sequential transducers. *Lecture Notes in Computer Science*, 807, 1994.
13. M. Mohri. Syntactic analysis by local grammars automata: an efficient algorithm. In *Proceedings of the International Conference on Computational Lexicography (COMPLEX 94).* Linguistic Institute, Hungarian Academy of Science: Budapest, Hungary, 1994.
14. M. Mohri. On some applications of finite-state automata theory to natural language processing. *Journal of Natural Language Engineering*, 2:1–20, 1996.
15. M. Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23, 1997.
16. M. Mohri. A general framework for shortest distance problems, 1997. In preparation.
17. M. Mohri, F. C. N. Pereira, and M. Riley. Weighted automata in text and speech processing. In *ECAI-96 Workshop, Budapest, Hungary.* ECAI, 1996.
18. M. Mohri and R. Sproat. An efficient compiler for weighted rewrite rules. In *34th Meeting of the Association for Computational Linguistic s (ACL 96), Proceedings of the Conference, Santa Cruz, California.* ACL, 1996.
19. D. Musser and A. Saini. *STL Tutorial and Reference Guide.* Addison-Wesley, 1996.
20. D. Revuz. Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92:181–189, 1992.
21. G. Riccardi, E. Bocchieri, and R. Pieraccini. Non-deterministic stochastic language models for speech recognition. In *Proceedings IEE International Conference on Acoustics, Speech and Signal Processing*, volume 1, pages 237–240. IEEE, 1995.
22. E. Ristad and P. Yianilos. Finite growth models. Technical report CS-TR-533-96, Department of Computer Science, Princeton University, 1996.
23. M. P. Schützenberger. On the definition of a family of automata. *Information and Control*, 4, 1961.
24. I. Simon. Limited subsets of a free monoid. In *Proceedings of the 19th Annual Symposium on Foundation of Computer Science*, pages 143–150, 1978.
25. R. Sproat, C. Shih, W. Gale, and N. Chang. A stochastic finite-state word-segmentation algorithm for Chinese. In *32nd Annual Meeting of the Association for Computational Linguistics*, pages 66–73, San Francisco, California, 1994. New Mexico State University, Las Cruces, New Mexico, Morgan Kaufmann.