WFSC – A New Weighted Finite State Compiler

André Kempe[1], Christof Baeijs[1], Tamás Gaál[1]
Franck Guingne[1,2], Florent Nicart[1,2]

[1] Xerox Research Centre Europe – Grenoble Laboratory
6 chemin de Maupertuis – 38240 Meylan – France
firstname.lastname@xrce.xerox.com – http://www.xrce.xerox.com

[2] Laboratoire d'Informatique Fondamentale et Appliquée de Rouen
Faculté des Sciences et des Techniques – Université de Rouen
76821 Mont-Saint-Aignan – France
firstname.lastname@dir.univ-rouen.fr – http://www.univ-rouen.fr/LIFAR/

Abstract. This article presents a new tool, WFSC, for creating, manipulating, and applying weighted finite state automata. It inherits some powerful features from Xerox's non-weighted XFST tool and represents a continuation of Xerox's work in the field of finite state automata over two decades. The design is generic: algorithms work on abstract components of automata and on a generic abstract semiring, and are independent of their concrete realizations. Applications can access WFSC's functions through an API or create automata through an end-user interface, either from an enumeration of their states and transitions or from rational expressions.

1 Introduction

Finite state automata (FSAs) are mathematically well defined and offer many practical advantages. They allow for fast processing of input data and are easily modifiable and combinable by well defined operations. Therefore, FSAs are widely used in Natural Language Processing (NLP) (Kaplan and Kay, 1981; Koskenniemi, Tapanainen, and Voutilainen, 1992; Sproat, 1992; Karttunen et al., 1997; Mohri, 1997; Roche and Schabes, 1997; Sproat, 2000) and in many other fields. There are several toolkits that support the creation and use of FSAs, such as XFST (Karttunen et al., 1996-2003; Beesley and Karttunen, 2003), FSA Utilities (van Noord, 2000), FIRE Lite (Watson, 1994), INTEX (Silberztein, 1999), and many more.

Weighted finite state automata (WFSAs) combine the advantages of ordinary FSAs with those of statistical models, such as Hidden Markov Models (HMMs), and hence have a potentially wider scope of application than FSAs. Some toolkits support the work with WFSAs, such as the pioneering implementation FSM (Mohri, Pereira, and Riley, 1998), Lextools on top of FSM (Sproat, 2003), and FSA Utilities (van Noord, 2000).

WFSC (Weighted Finite State Compiler) is our new tool for creating, manipulating, and applying WFSAs. It inherits some powerful features from Xerox's

non-weighted XFST tool, that are crucial for many practical applications. For example, the "unknown symbol" allows us to assign the infinite set of all unknown symbols to a single transition rather than declaring in advance all symbols that potentially could occur and assigning each of them to a separate transition. This saves a considerable amount of memory and processing time. Flag diacritics, another feature proposed by Xerox, can also reduce the size of FSAs. They are extensively used in the analysis of morphologically rich languages such as Finnish and Hungarian. WFSC represents a continuation of Xerox's work in the field of FSAs, spanning over two decades (Kaplan and Kay, 1981; Karttunen, Kaplan, and Zaenen, 1992; Karttunen et al., 1996-2003; Beesley and Karttunen, 2003).

This article is structured as follows: Section 2 explains some of the mathematical background of WFSAs. Section 3 gives an overview of the modular generic design of WFSC, describing the system architecture (3.1), the central role and the implementation of sets (3.2), and the approach for programming the algorithms (3.3). Section 4 presents WFSC from the users' perspective, describing the end-user interface (4.1) and an example of application (4.2). Section 5 concludes the article.

2 Preliminaries

In this section we recall the basic definitions of our framework: algebraic structures such as monoid and semiring, as well as weighted automata and transducers (Eilenberg, 1974; Kuich and Salomaa, 1986).

2.1 Semirings

A *monoid* consists of a set M, an associative binary operation $\circ$ on M, and a *neutral* element $\bar{1}$ such that $\bar{1} \circ a = a \circ \bar{1} = a$ for all $a \in M$. A monoid is called *commutative* iff $a \circ b = b \circ a$ for all $a, b \in M$.

The set K with two binary operations $\oplus$ and $\otimes$ and two elements $\bar{0}$ and $\bar{1}$ is called a *semiring*, if it satisfies the following properties:

1. $\langle K, \oplus, \bar{0} \rangle$ is a commutative monoid
2. $\langle K, \otimes, \bar{1} \rangle$ is a monoid
3. $\otimes$ is *left-* and *right-distributive* over $\oplus$:
   $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)\,, \quad (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)\,, \quad \forall a, b, c \in K$
4. $\bar{0}$ is an annihilator for $\otimes$: $\quad \bar{0} \otimes a = a \otimes \bar{0} = \bar{0}\,, \quad \forall a \in K$

We denote a generic semiring $\mathcal{K}$ as $\langle K, \oplus, \otimes, \bar{0}, \bar{1} \rangle$.

Some automaton algorithms require semirings to have specific properties. For example, composition as proposed by (Pereira and Riley, 1997; Mohri, Pereira, and Riley, 1998) requires a semiring to be commutative, and ε-removal as proposed by (Mohri, 2002) requires it to be *k-closed*. These properties are defined as follows:

1. commutativity: $a \otimes b = b \otimes a\ ,\ \ \forall a, b \in K$
2. k-closedness: $\bigoplus_{n=0}^{k+1} a^n = \bigoplus_{n=0}^{k} a^n\ ,\ \ \forall a \in K$

The following well-known examples are all commutative semirings:

1. $\langle \mathbb{B}, +, \times, 0, 1 \rangle$: boolean semiring, with $\mathbb{B} = \{0, 1\}$ and $1 + 1 = 1$
2. $\langle \mathbb{N}, +, \times, 0, 1 \rangle$: integer semiring with the usual addition and multiplication
3. $\langle \mathbb{R}^+, +, \times, 0, 1 \rangle$: real positive sum times semiring
4. $\langle \overline{\mathbb{R}}^+, min, +, \infty, 0 \rangle$: a real tropical semiring where $\overline{\mathbb{R}}^+$ denotes $\mathbb{R}^+ \cup \{\infty\}$

A number of algorithms require semirings to be equipped with an order or partial order denoted by $<_{\mathcal{K}}$ (example in Section 3.3). Each idempotent semiring $\mathcal{K}$ (i.e., $\forall a \in \mathcal{K} : a \oplus a = a$) has a natural partial order defined by $a <_{\mathcal{K}} b \Leftrightarrow a \oplus b = a$. In the above examples, the boolean and the real tropical semiring are idempotent, and hence have a natural partial order.

2.2 Weighted Automata and Transducers

A *weighted automaton* A over a semiring $\mathcal{K}$ is defined by the 6-tuple $\langle \Sigma, Q, I, F, E_A, \mathcal{K} \rangle$, and a *weighted transducer* T by the 7-tuple $\langle \Sigma, \Omega, Q, I, F, E_T, \mathcal{K} \rangle$, where

Σ, Ω	are finite alphabets
Q	is the finite set of states
$I \subseteq Q$	is the set of initial states
$F \subseteq Q$	is the set of final states
$E_A \subseteq Q \times \Sigma \cup \{\varepsilon\} \times Q$	is the set of transitions of A
$E_T \subseteq Q \times \Sigma \cup \{\varepsilon\} \times \Omega \cup \{\varepsilon\} \times Q$	is the set of transitions of T
$\mathcal{K}$	is a semiring

In the following, both automata and transducers will be referred to as *networks*. By convention, our networks have only one initial state $i \in I$ without loss of generality since for any network with multiple start states there exists a network with a single start state accepting the same language. For any state $q \in Q$, we denote by

$\lambda(q)$	$\lambda : I \to \mathcal{K}$	the initial weight function with $\lambda(q) = \bar{0}\ ,\ \ \forall q \notin I$
$\varrho(q)$	$\varrho : F \to \mathcal{K}$	the final weight function with $\varrho(q) = \bar{0}\ ,\ \ \forall q \notin F$

and for any transition $e \in E$

$w(e)$	$w : E \to \mathcal{K}$	the weight of e with $w(e) \neq \bar{0}\ ,\ \ \forall e \in E$
$p(e)$	$p : E \to Q$	the source state of e
$n(e)$	$n : E \to Q$	the target state of e
$a(e)$	$a : E \to \Sigma \cup \{\varepsilon\} \times \Omega \cup \{\varepsilon\}$	the label of e

A *path* π of length $l = |\pi|$ is a sequence of transitions $e_1 e_2 \cdots e_l$ such that $n(e_i) = p(e_{i+1})$ for all $i \in [\![1, l-1]\!]$. A path is said to be *successful* iff $p(e_1) \in I$ and $n(e_l) \in F$. For any successful path π, the accepting weight $w(\pi)$ is given by

$$w(\pi) \quad = \quad \lambda(p(e_1)) \otimes \left(\bigotimes_{j=[\![1,l]\!]} w\left(e_j\right) \right) \otimes \varrho(n\left(e_l\right)) \tag{1}$$

We denote by $\Pi(s)$ the set of successful paths for the input string s. Thus, the accepting weight for any input string s is defined by

$$w(s) \quad = \quad \bigoplus_{\pi \in \Pi(s)} w(\pi) \tag{2}$$

Composition of transducers T_i is expressed either by the $\diamond$ or the $\circ$ operator. However, $T_1 \diamond T_2 = T_2 \circ T_1$ which corresponds to $T_2(\, T_1(\ )\,)$ in functional notation (Birkhoff and Bartee, 1970).

3 Modular Generic Design

3.1 Layers of the WFSC Library

WFSC has been designed in a modular and generic way in several layers to facilitate its maintenance and the implementation of new algorithms (Figure 1) : the bottom layer contains different physical realizations of automaton components such as many different types of states and transitions. It is followed by a layer of abstract automaton components such as one single abstract type of transitions or states. The next higher layer contains basic automaton algorithms, and is followed by a layer of more complex algorithms. Algorithms work only on abstract components, and are independent of their concrete physical realizations in the lowest layer. Physical components can change their form (to optimally adapt to a situation) without disturbing the correct functioning of the algorithms. C++ was chosen as the implementation language, as a compromise between expressive power, efficiency, and modularity.

Semirings, which are themselves modular theoretical concepts, are implemented in WFSC in a modular way. An algorithm always works on a generic (abstract) semiring, and functions correctly no matter which actual semiring is behind it (provided the semiring has all properties required by the algorithm).

Programmers of algorithms will work only with abstract automaton components (low-level interface) and do not have to deal with their concrete physical realizations. Programmers of practical applications can use WFSC's function library through an API or an end-user interface (Section 4.1).

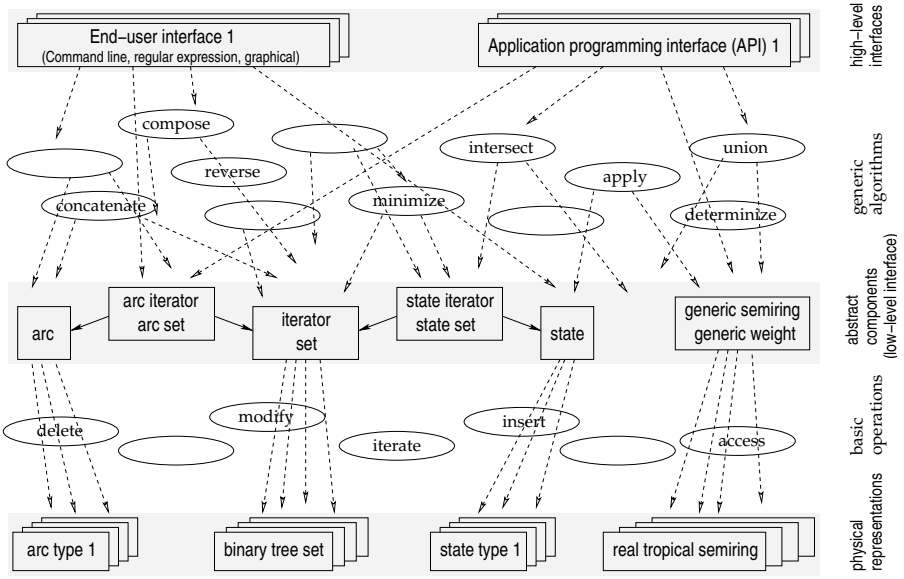

Fig. 1. Layers of the WFSC library (simplified extract).

3.2 Central Role of Sets

Sets play an important role in automata theory. An automaton has a state set Q and a transition set E and each of its states $q \in Q$ has a set of outgoing transitions $E(q)$. Automaton algorithms make extensive use of set operations. In addition to the above sets they may manipulate sets of auxiliary "objects" such as state pairs or pairs of a state and some related weight. For example, the pseudocode in Figure 3, showing a modified version of the Viterbi algorithm (Viterbi, 1967), contains 7 out of 18 lines with set operations (lines: 4, 6, 7, 8, 12, 15, 17).

To facilitate an efficient and easy implementation of algorithms, we consider it crucial to provide WFSC with a generic and flexible implementation of sets supporting a large number of basic set operations, alternative internal structures such as vector, list, binary tree, etc. We designed sets so that they have a compact representation since each of the (possibly many millions of) states of an automaton has a set of outgoing transitions, and since algorithms like composition can dramatically increase that number.

Since the default implementation of sets in C++ does not meet these requirements, a special technique has been developed, called Bitwise Virtuality, that allows class abstraction and polymorphism at little cost in memory (Nicart, 2003). Furthermore, this mechanism allows on-the-fly changing of type and methods of existing objects and hence on-the-fly conversion among set structures for runtime optimization in different steps of an algorithm.

3.3 Algorithm Programming Style

The layer of abstract automaton components (low-level interface, Figure 1) in the WFSC library allows us to write algorithms in a style close to pseudocode. Low-level operations (such as keeping a list linked) are hidden inside the C++ classes that implement the abstract components. This approach facilitates the implementation and maintenance of algorithms by allowing programmers to fully concentrate on the algorithms themselves rather than on low-level operations. Numerous versions of a new algorithm can be tested in relatively short time.

We illustrate this programming style on the example of a modified version of the Viterbi algorithm. The "classical" Viterbi algorithm is used for estimating in linear time the most likely path through a Hidden Markov Model (HMM), given a sequence of observation symbols (Viterbi, 1967; Rabiner, 1990; Manning and Schütze, 1999).

We use our modified version of the algorithm for identifying the "best" path of bounded length in a WFSA, ignoring the symbols. Conceptually, the algorithm uses a trellis of nodes where each row corresponds to one state in the WFSA, each column to one step in the traversal (Figure 3). For example, the node in row 2 and column 3 represents the fact of reaching state 2 after traversing 3 transitions.

In the pseudocode of the algorithm, we describe each trellis node (in column t) by a 4-tuple $m_t = \langle q_t, \psi_t, e_{t-1}, m_{t-1} \rangle$ with q_t being the state of m_t, ψ_t the weight of the best path from the initial state i to q_t, e_{t-1} the last transition on this path, and m_{t-1} a back-reference to the trellis node of the source state of e_{t-1} (Figure 2). Absent elements are expressed by $\perp$. The sets M_t and M_{t+1} describe the columns t and $t+1$ respectively. Given two weights w and w', we write $w \succ w'$ and $w \prec w'$ to express that w is "better" or "worse" than w' respectively, meaning $w > w'$ or $w < w'$ according to maximum or minimum search.

The algorithm first checks a property of the semiring $\mathcal{K}$, namely whether continuing on a path π beyond some state q can lead to a better weight than the one compiled up to q (line 1). It then initializes M_0 with a single m_0 corresponding to the initial state i (Figure 3b and Figure 2 lines 3, 4). It inserts into each following set M_{t+1} one element m_{t+1} for each state q' that can be reached by a transition e from some state q having an element $m_t \in M_t$ (lines 8 to 17): each newly created m_{t+1} (line 14) is compared to a previously created m'_{t+1} of the same state $n(e)$ which either exists in M_{t+1} or is $\perp$ (lines 15, 16). Only the best of the two is kept in M_{t+1} (lines 16, 17) so that at any time there is at most one m_{t+1} for a given state q' in M_{t+1}. A transition e is not taken if it cannot be on the best path (lines 1, 13). Whether an m is better than another one depends on the following conventions: $\psi(m) \succ \psi(m') \Rightarrow m \succ m'$, $m = \perp \Rightarrow (\psi(m) = \bar{0} \ \wedge \ \varrho(q(m)) = \bar{0})$, and $(m \neq \perp \ \wedge \ m' = \perp) \Rightarrow m \succ m'$. When a final state q_t is reached, we have identified a complete path whose weight is $\psi_t \otimes \varrho(q_t)$ (since $\lambda(i) = \bar{1}$). The variable $\widehat{m}$ refers to the m of the final state q of the best complete path found so far (lines 9 and 10).

In the C++ program, m_t is represented by `m0={q,psi,e_prev,m_prev}`, m_{t+1} by `m1`, m'_{t+1} by `m1a`, and $\widehat{m}$ by `mBest`. The sets M_t and M_{t+1} are denoted by `M0`

```
ViterbiBestPath(A, maxlength) :
1   κ ← (∀w1,w2∈K: w1⊗w2 ⊁ w1)                          (improvement impossible)
2   m̂ ← ⊥
3   m0 ← ⟨i, Ī, ⊥, ⊥⟩
4   M0 ← {m0}
5   t ← 0
6   while  (t ≤ maxlength) ∧ (Mt ≠ ∅)  do
7        Mt+1 ← ∅
8        for ∀mt ∈ Mt  do
9             if  ψ(m̂) ⊗ ϱ(q(m̂)) ≺ ψ(mt) ⊗ ϱ(q(mt))
10                 then m̂ ← mt
11            if  t < maxlength
12                 for ∀e ∈ E(qt)  do
13                      if  ¬( κ ∧ ((p(e)=n(e)) ∨ (ϱ(p(e)) ≻ w(e))) )
14                           then mt+1 ← ⟨n(e), ψt⊗w(e), e, mt⟩
15                                m't+1 ← ⟨n(e),  ,  ,  ⟩ ∈ Mt+1          (possibly ⊥)
16                                if  m't+1 ≺ mt+1
17                                     then Mt+1 ← {Mt+1 − {m't+1}} ∪ {mt+1}
18       t ← t+1
19  return BuildPath(m̂)
```

```
Wfsa ViterbiBestPath (Wfsa* A, int maxlength, bool min_search)
{
  bool (*better_weight)(Weight, Weight, Semiring*) = (min_search) ? lower_weight : higher_weight;
  Set<m> M0(), M1();
 ..... ;
1:  bool improvement_imposs = (min_search) ? A->K->is_monAscending() : A->K->is_monDescending();
2:  m* mBest = 0;
3:  m* m0 = new m (A->i, A->K->_1, 0, 0);
4:  M0.insert (m0);
5:  int t = 0;
6:  while ((t <= maxlength) && (M0.size() > 0)) {
7:    M1.clear();
8:    for (M0_Iterator.connect (M0); !M0_Iterator.end(); M0_Iterator++) {
        m0 = M0_Iterator.item();
9:      if (better_m (m0, mBest, A->K, better_weight))
10:       mBest = m0;
11:     if (t < maxlength) {
12:       for (E_Iterator.connect (m0->q->arcSet); !E_Iterator.end(); E_Iterator++) {
            e = E_Iterator.item();
13:         if (! ( improvement_imposs &&
                    (m0->q == e->target || better_weight (m0->rho(), e->weight, A->K)) ) ) {
14:           m1 = new m (e->target, A->K->extension (m0->psi, e->weight), e, m0);
              M1_Iterator.connect (M1);
15:           m1a = M1_Iterator.search (m1, compare_function);
16:           if (better_m (m1, m1a, A->K, better_weight)) {
17:             M1_Iterator.replace (m1a, m1);
                delete m1a;  }
              else
                delete m1;  } } } }
18:   t ++;
      swap_M (M0, M1);
    }
19: return BuildPath (mBest);
}
```

Fig. 2. Illustration of the similarity between pseudocode and C++ program through a modified version of the Viterbi algorithm (corresponding lines have equal numbers).

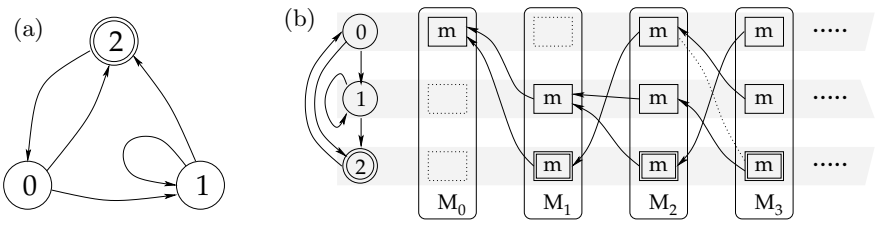

Fig. 3. Illustration of a modified Viterbi algorithm through (a) a WFST and (b) the corresponding trellis (labels and weights are omitted).

and `M1` respectively and use our own implementation of sets (Section 3.2). Null pointers indicate absent elements.

For the purpose of optimization (in the C++ program) we add a reference counter to each node m_t and delete m_t (and possibly some of its predecessors m_{t-k}) when it is no longer referenced by any successor node m_{t+j}. All sets M_{t-k} preceding M_t are deleted (without deleting all of their members m_{t-k}), which allows us to keep only two sets permanently, M_t and M_{t+1}, that are swapped after each step of iteration.

4 Creating Applications With WFSC

4.1 End-User Interface

WFSC is both a compiler, creating weighted automata from different descriptions, and an interactive programming and testing environment. Easy, intuitive definition and manipulation of networks, as in Xerox's non-weighted XFST toolkit (Karttunen et al., 1996-2003; Beesley and Karttunen, 2003), are vital to the success of an application (Aït-Mokhtar and Chanod, 1997; Grefenstette, Schiller, and Aït-Mokhtar, 2000).

A network can be described either through an enumeration of its states and transitions, including weights, or through a rational expression (i.e., a regular expression with weights).

The interactive WFSC interface provides commands for reading, writing, optimizing, exploring, visualizing, and applying networks to input. One can also create new networks from existing ones by explicitly calling operations such as union and composition. WFSC commands can be executed interactively or written to a batch file and executed as a single job. Using WFSC it is possible to read legacy non-weighted networks, created by XFST, and add weights to their states and transitions. Conversely, weights can be stripped from a weighted network to produce a non-weighted network compatible with XFST.

A new finite-state programming language is also under development (Beesley, 2003). In addition to the compilation of regular-expression and phrase-structure notations, it will provide boolean tests, imperative control structures, Unicode support, and a graphical user interface.

4.2 An Implemented Application

Optical Character Recognition (OCR) converts the bitmap of a scanned page of text into a sequence of symbols (characters) equal to the text. Post-OCR Correction attempts to reduce the number of errors in a text generated by OCR, using language models and other statistical information. This task can be performed with WFSTs (Abdallahi, 2002).

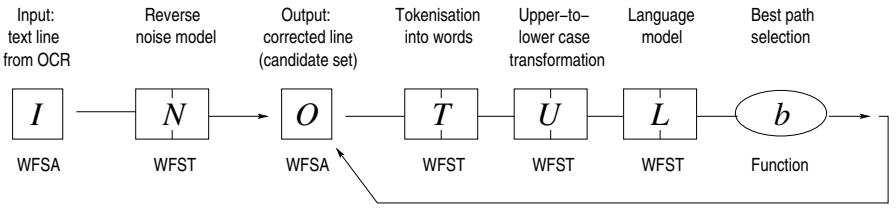

Fig. 4. Block diagram of the post-OCR correction of one text line, using WFSTs.

The task consists in finding the most likely corrected output text line $\hat{o}$ in the set of all possible output lines O, given an input line i generated by OCR:

$$\hat{o} \;=\; \arg\max_{o \in O} \; p(\, o \mid i \,) \tag{3}$$

The implementation of this task with WFSC uses some basic automaton algorithms: composition, best-path search, and projection of either the input or output tape of a transducer (Figure 4) :

$$\hat{o} \;=\; project_i(\, bestpath(\, project_o(\, I \diamond N \,) \diamond T \diamond U \diamond L \,) \,) \tag{4}$$

First, we build a WFSA I representing the input line. Each transition of I is labeled with one (possibly incorrect) symbol of this line. Then, we construct the *output-side* projection of the composition of I with a WFST N representing a reverse *noise model*: $project_o(\, I \diamond N \,)$. The language of the resulting WFSA contains all lines of text that could have generated the (possibly incorrect) OCR output. To find the most likely from among those lines, we compose them with a WFST T, that introduces separator symbols between words, a WFST U, that transforms all upper-case letters into lower-case, and a WFST L, that represents a *language model*: $(\, \ldots \diamond T \diamond U \diamond L \,)$. Finally, we take the *input-side* projection of the best path: $project_i(\, bestpath(\, \ldots \,) \,)$.

Note that N evaluates the probability of letter sequences and L the probability of word sequences.

5 Conclusion

The article presented a new tool, WFSC, for creating, manipulating, and applying weighted finite state automata. WFSC inherits some powerful features from Xerox's non-weighted XFST tool, such as the "unknown symbol" and flag diacritics.

In WFSC, all algorithms work on abstract components of automata and on a generic abstract semiring, and are independent of their concrete realizations. Algorithm programmers can write in a style close to pseudocode which allows for fast prototyping. Since automaton algorithms make extensive use of set operations, special care has been given to a generic and flexible implementation of sets supporting a large number of basic operations and alternative internal structures that are inter-changeable on-the-fly.

Programmers of applications can either access WFSC's function library through an API or create weighted automata through an end-user interface. The interface has a basic set of commands for network creation, input and output, operations on networks, network optimization, inspection, display, etc. Automata are built either from an enumeration of their states and transitions or from regular expressions that are extended to allow for specification of weights.

WFSC can be used in large-scale real-life applications. It does, however, not yet have all features initially planned. The implementation work is continuing, and due to WFSC's generic and modular design new features and algorithms can be added easily.

Acknowledgments. We would like to thank Jean-Marc Champarnaud and Kenneth R. Beesley for their advice, and Lemine Abdallahi for his help in implementing the described application.

References

Abdallahi, Lemine. 2002. Ocr postprocessing. Internal technical report, Xerox Research Centre Europe, Meylan, France.

Aït-Mokhtar, Salah and Jean-Pierre Chanod. 1997. Incremental finite-state parsing. In *Proceedings of Applied Natural Language Processing*, Washington, DC.

Beesley, Kenneth R. 2003. A language for finite state programming. *In preparation.*

Beesley, Kenneth R. and Lauri Karttunen. 2003. *Finite State Morphology.* CSLI Publications, Palo Alto, CA, USA. URL: http://www.fsmbook.com/.

Birkhoff, Garrett and Thomas C. Bartee. 1970. *Modern Applied Algebra.* McGraw-Hill, New York, USA.

Eilenberg, Samuel. 1974. *Automata, Languages, and Machines*, volume A. Academic Press, San Diego, CA, USA.

Grefenstette, Greg, Anne Schiller, and Salah Aït-Mokhtar. 2000. Recognizing lexical patterns in text. In F. Van Eynde and D. Gibbon, editors, *Lexicon Development for Speech and Language Processing*. Kluwer Academic Publishers, pages 431–453.

Kaplan, Ronald M. and Martin Kay. 1981. Phonological rules and finite state transducers. In *Winter Meeting of the Linguistic Society of America*, New York, USA.

Karttunen, Lauri, Jean-Pierre Chanod, Greg Grefenstette, and Anne Schiller. 1997. Regular expressions for language engineering. *Journal of Natural Language Engineering*, 2(4):307–330.

Karttunen, Lauri, Tamás Gaál, Ronald M. Kaplan, André Kempe, Pasi Tapanainen, and Todd Yampol. 1996-2003. *Xerox Finite-State Home Page*. Xerox Research Centre Europe, Grenoble, France. URL: http://www.xrce.xerox.com/competencies/content-analysis/fst/.

Karttunen, Lauri, Ronald M. Kaplan, and Annie Zaenen. 1992. Two-level morphology with composition. In *Proceedings of COLING'92*, pages 141–148, Nantes, France.

Koskenniemi, Kimmo, Pasi Tapanainen, and Atro Voutilainen. 1992. Compiling and using finite-state syntactic rules. In *Proceedings of COLING'92*, volume 1, pages 156–162, Nantes, France.

Kuich, Werner and Arto Salomaa. 1986. *Semirings, Automata, Languages*. Number 5 in EATCS Monographs on Theoretical Computer Science. Springer Verlag, Berlin, Germany.

Manning, Christopher D. and Hinrich Schütze. 1999. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA.

Mohri, Mehryar. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–312.

Mohri, Mehryar. 2002. Generic epsilon-removal and input epsilon-normalization algorithms for weighted transducers. *International Journal of Foundations of Computer Science*, 13(1):129–143.

Mohri, Mehryar, Fernando C. N. Pereira, and Michael Riley. 1998. A rational design for a weighted finite-state transducer library. Number 1436 in Lecture Notes in Computer Science. Springer Verlag, Berlin, Germany, pages 144–158.

Nicart, Florent. 2003. Toward scalable virtuality in C++. *In preparation*.

Pereira, Fernando C. N. and Michael D. Riley. 1997. Speech recognition by composition of weighted finite automata. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*. MIT Press, Cambridge, MA, USA, pages 431–453.

Rabiner, Lawrence R. 1990. A tutorial on hidden markov models and selected applications in speech recognition. In Alex Waibel and Kai-Fu Lee, editors, *Readings in Speech Recognition*. Morgan Kaufmann, pages 267–296.

Roche, Emmanuel and Yves Schabes. 1997. *Finite-State Language Processing*. MIT Press, Cambridge, MA, USA.

Silberztein, Max. 1999. INTEX: a finite state transducer toolbox. volume 231 of *Theoretical Computer Science*. Elsevier Science, pages 33–46.

Sproat, Richard. 1992. *Morphology and Computation*. MIT Press, Cambridge, MA.

Sproat, Richard. 2000. *A Computational Theory of Writing Systems*. Cambridge University Press, Cambridge, MA.

Sproat, Richard. 2003. *Lextools Home Page*. AT&T Labs – Research, Florham Park, NJ, USA. URL: http://www.research.att.com/sw/tools/lextools/.

van Noord, Gertjan. 2000. *FSA6 – Finite State Automata Utilities Home Page*. Alfa-informatica, University of Groningen, The Netherlands. URL: http://odur.let.rug.nl/ vannoord/Fsa/.

Viterbi, Andrew J. 1967. Error bounds for convolutional codes and an asymptotical optimal decoding algorithm. In *Proceedings of the IEEE*, volume 61, pages 268–278. Institute of Electrical and Electronics Engineers.

Watson, Bruce W. 1994. *The Design and Implementation of the FIRE engine: A C++ Toolkit for Finite Automata and Regular Expressions*. Computing science note 94/22, Eindhoven University of Technology, The Netherlands.