FSA: An Efficient and Flexible C++ Toolkit for Finite State Automata Using On-Demand Computation

Stephan Kanthak and Hermann Ney
Lehrstuhl für Informatik VI, Computer Science Department
RWTH Aachen – University of Technology
52056 Aachen, Germany
{kanthak,ney}@informatik.rwth-aachen.de

Abstract

In this paper we present the RWTH FSA toolkit – an efficient implementation of algorithms for creating and manipulating weighted finite-state automata. The toolkit has been designed using the principle of on-demand computation and offers a large range of widely used algorithms. To prove the superior efficiency of the toolkit, we compare the implementation to that of other publically available toolkits. We also show that on-demand computations help to reduce memory requirements significantly without any loss in speed. To increase its flexibility, the RWTH FSA toolkit supports high-level interfaces to the programming language Python as well as a command-line tool for interactive manipulation of FSAs. Furthermore, we show how to utilize the toolkit to rapidly build a fast and accurate statistical machine translation system. Future extensibility of the toolkit is ensured as it will be publically available as open source software.

1 Introduction

Finite-state automata (FSA) methods proved to elegantly solve many difficult problems in the field of natural language processing. Among the most recent ones are full and lazy compilation of the search network for speech recognition (Mohri et al., 2000a), integrated speech translation (Vidal, 1997; Bangalore and Riccardi, 2000), speech summarization (Hori et al., 2003), language modelling (Allauzen et al., 2003) and parameter estimation through EM (Eisner, 2001) to mention only a few. From this list of different applications it is clear that there is a high demand for generic tools to create and manipulate FSAs.

In the past, a number of toolkits have been published, all with different design principles. Here, we give a short overview of toolkits that offer an almost complete set of algorithms:

- The FSM Library^{TM} from AT&T (Mohri et al., 2000b) is judged the most efficient implementation, offers various semirings, on-demand computation and many algorithms, but is available only in binary form with a proprietary, non commercial license.
- FSA6.1 from (van Noord, 2000) is implemented in Prolog. It is licensed under the terms of the (GPL, 1991).
- The WFST toolkit from (Adant, 2000) is built on top of the Automaton Standard Template Library (LeMaout, 1998) and uses C++ template mechanisms for efficiency and flexibility, but lacks on-demand computation. Also licensed under the terms of the (GPL, 1991).

This paper describes a highly efficient new implementation of a finite-state automata toolkit that uses on-demand computation. Currently, it is being used at the Lehrstuhl für Informatik VI, RWTH Aachen in different speech recognition and translation research applications. The toolkit will be available under an open source license ((GPL, 1991)) and can be obtained from our website `http://www-i6.informatik.rwth-aachen.de`.

The remaining part of the paper is organized as follows: Section 2 will give a short introduction to the theory of finite-state automata to recall part of the terminology and notation. We will also give a short explanation of composition which we use as an exemplary object of study in the following sections. In Section 2.3 we will discuss the locality of algorithms defined on finite-state automata. This forms the basis for implementations using on-demand computations. Then the RWTH FSA toolkit implementation is detailed in Section 3. In Section 4.1 we will compare the efficiency of different toolkits. As a showcase for the flexibility we show how to use the toolkit to build a statistical machine translation system in Section 4.2. We conclude the paper with a short summary in Section 5 and discuss some possible future extensions in Section 6.

2 Finite-State Automata

2.1 Weighted Finite-State Transducer

The basic theory of weighted finite-state automata has been reviewed in numerous papers (Mohri, 1997; Allauzen et al., 2003). We will introduce the notation briefly.

A *semiring* $(K, \oplus, \otimes, \overline{0}, \overline{1})$ is a structure with a set K and two binary operations $\oplus$ and $\otimes$ such that $(K, \oplus, \overline{0})$ is a commutative monoid, $(K, \otimes, \overline{1})$ is a monoid and $\otimes$ distributes over $\oplus$ and $\overline{0} \otimes x = x \otimes \overline{0} = \overline{0}$ for any $x \in K$. We will also associate the term *weights* with the elements of a semiring. Semirings that are frequently used in speech recognition are the *positive real semiring* $(\mathbb{R} \cup \{-\infty, +\infty\}, \oplus_{log}, +, +\infty, 0)$ with $a \oplus_{log} b = -log(e^{-a} + e^{-b})$ and the *tropical semiring* $(\mathbb{R} \cup \{-\infty, +\infty\}, \min, +, +\infty, 0)$ representing the well-known sum and maximum weighted path criteria.

A *weighted finite-state transducer* $(Q, \Sigma \cup \{\epsilon\}, \Omega \cup \{\epsilon\}, K, E, i, F, \lambda, \rho)$ is a structure with a set Q of states[1], an alphabet Σ of input symbols, an alphabet Ω of output symbols, a weight semiring K (we assume it k-closed here for some algorithms as described in (Mohri and Riley, 2001)), a set $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Omega \cup \{\epsilon\}) \times K \times Q$ of arcs, a single initial state i with weight λ and a set of final states F weighted by the function $\rho : F \to K$. To simplify the notation we will also denote with Q_T and E_T the set of states and arcs of a transducer T. A *weighted finite-state acceptor* is simply a weighted finite-state transducer without the output alphabet.

2.2 Composition

As we will refer to this example throughout the paper we shortly review the composition algorithm here. Let $T_1 : \Sigma^* \times \Omega^* \to K$ and $T_2 : \Omega^* \times \Gamma^* \to K$ be two transducers defined over the same semiring K. Their composition $T_1 \circ T_2$ realizes the function $T : \Sigma^* \times \Gamma^* \to K$ and the theory has been described in detail in (Pereira and Riley, 1996).

For simplification purposes, let us assume that the input automata are ϵ-free and $S = (Q_1 \times Q_2, \leftarrow, \rightarrow, \text{empty})$ is a stack of state tuples of T_1 and T_2 with push, pop and empty test operations. A non lazy version of composition is shown in Figure 1.

Composition of automata containing ϵ labels is more complex and can be solved by using an intermediate filter transducer that also has been described in (Pereira and Riley, 1996).

[1] we do not restrict this to be a finite set as most algorithms of the lazy implementation presented in this paper also support a virtually infinite set

$T = T_1 \circ T_2 :$

$i = (i_1, i_2)$
$S \leftarrow (i_1, i_2)$
while not S empty
  $(s_1, s_2) \leftarrow S$
  $Q_T = Q_T \cup (s_1, s_2)$
  foreach $(s_1, i_1, o_1, w_1, t_1) \in E_{T_1}$
    foreach $(s_2, i_2, o_2, w_2, t_2) \in E_{T_2}$ with $o_1 = i_2$
      $E_T = E_T \cup ((s_1, s_2), i_1, o_2, w_1 \otimes w_2, (t_1, t_2))$
      if $(t_1, t_2) \notin Q_T$ then $S \leftarrow (t_1, t_2)$

Figure 1: Simplified version of composition (assumes ϵ-free input transducers).

What we can see from the pseudo-code above is that composition uses tuples of states of the two input transducers to describe states of the target transducer. Other operations defined on weighted finite-state automata use different *abstract states*. For example transducer determinization (Mohri, 1997) uses a set of pairs of states and weights. However, it is more convenient to use integers as state indices for an implementation. Therefore algorithms usually maintain a mapping from abstract states to integer state indices. This mapping has linear memory requirements of $O(|Q_T|)$ which is quite attractive, but that depends on the structure of the abstract states. Especially in case of determinization where the size of an abstract state may vary, the complexity is no longer linear in general.

2.3 Local Algorithms

Mohri and colleagues pointed out (Mohri et al., 2000b) that a special class of transducer algorithms can be computed on demand. We will give a more detailed analysis here. We focus on algorithms that produce a single transducer and refer to them as *algorithmic transducers*.

Definition: Let θ be the input configuration of an algorithm $A(\theta)$ that outputs a single finite-state transducer T. Additionally, let $M : S \to Q_T$ be a one-to-one mapping from the set of abstract state descriptions S that A generates onto the set of states of T. We call A *local* iff for all states $s \in Q_T$ A can generate a state s of T and all outgoing arcs $(s, i, o, w, s') \in E_T$, depending only on its abstract state $M^{-1}(s)$ and the input configuration θ.

With the preceding definition it is quite easy to prove the following lemma:

Lemma: An algorithm A that has the *local* property can be built on demand starting with the initial state i_{T_A} of its associated algorithmic transducer T_A.

Proof: For the proof it is sufficient to show that we can generate and therefore reach all states of T_A.

Let S be a stack of states of T_A that we still have to process. Due to the one-to-one mapping M we can map each state of T_A back to an abstract state of A. By definition the abstract state is sufficient to generate the complete state and its outgoing arcs. We then push those target states of all outgoing arcs onto the stack S that have not yet been processed. As T_A is finite the traversal ends after all states of T_A as been processed exactly once. □

Algorithmic transducers that can be computed on-demand are also called *lazy* or *virtual transducers*. Note, that due to the *local* property the set of states does not necessarily be finite anymore.

3 The Toolkit

The current implementation is the second version of this toolkit. For the first version – which was called *FSM* – we opted for using C++ templates to gain efficiency, but algorithms were not lazy. It turned out that the implementation was fast, but many operations wasted a lot of memory as their resulting transducer had been fully expanded in memory. However, we plan to also make this initial version publically available.

The design principles of the second version of the toolkit, which we will call *FSA*, are:

- decoupling of data structures and algorithms,
- on-demand computation for increased memory efficiency,
- low computational costs,
- an abstract interface to alphabets to support lazy mappings from strings to indices for arc labels,
- an abstract interface to semirings (should be k-closed for at least some algorithms),
- implementation in C++, as it is fast, ubiquitous and well-known by many other researchers,
- easy to use interfaces.

3.1 The C++ Library Implementation

We use the lemma from Section 2.3 to specify an interface for lazy algorithmic transducers directly. The code written in pseudo-C++ is given in Figure 2. Note that all lazy algorithmic transducers are derived from the class `Automaton`.

The lazy interface also has disadvantages. The virtual access to the data structure might slow computations down, and obtaining global information about the automaton becomes more complicated. For example the size of an automaton can only be computed by traversing it. Therefore central algorithms of the RWTH FSA toolkit are the *depth-first search (DFS)* and the computation of *strongly connected components (SCC)*. Efficient versions of these algorithms are described in (Mehlhorn, 1984) and (Cormen et al., 1990).

```
class Automaton {
public:
  struct Arc {
    StateId target();
    Weight weight();
    LabelId input();
    LabelId output();
  };
  struct State {
    StateId id();
    Weight weight();
    ConstArcIterator arcsBegin();
    ConstArcIterator arcsEnd();
  };
  virtual R<Alphabet> inputAlphabet();
  virtual R<Alphabet> outputAlphabet();
  virtual StateId initialState();
  virtual R<State> getState(StateId);
};
```

Figure 2: Pseudo-C++ code fragment for the abstract datatype of transducers. Note that `R<T>` refers to a smart pointer of `T`.

It is very costly to store arbitrary types as arc labels within the arcs itself. Therefore the RWTH FSA toolkit offers alphabets that define mappings between strings and label indices. Alphabets are implemented using the abstract interface shown in Figure 4. With alphabets arcs only need to store the abstract label indices. The interface for alphabets is defined using a single constant: for each label index an alphabet reports it must ensure to always deliver the same symbol on request through `getSymbol()`.

```
class Alphabet {
public:
  virtual LabelId begin();
  virtual LabelId end();
  virtual LabelId next(LabelId);
  virtual string getSymbol(LabelId);
};
```

Figure 4: Pseudo-C++ code fragment for the abstract datatype of alphabets.

3.2 Algorithms

The current implementation of the toolkit offers a wide range of well-known algorithms defined on weighted finite-state transducers:

```
compose(T1, T2) = simple-compose(   cache(sort-output(map-output(T1, A_{T2,I}))),
                                    cache(sort-input(T2)))
```

Figure 3: Optimized composition where $A_{T_2,I}$ denotes the input alphabet of T_2. Six algorithmic transducers are used to gain maximum efficiency. Mapping of arc labels is necessary as symbol indices may differ between alphabets.

- **basic operations**
  sort (by input labels, output labels or by total arc), *map-input* and *-output* labels symbolically (as the user expects that two alphabets match symbolically, but their mapping to label indices may differ), *cache* (helps to reduce computations with lazy implementations), *topologically-sort* states
- **rational operations**
  project-input, *project-output*, *transpose* (also known as reversal: calculates an equivalent automaton with the adjacency matrix being transposed), *union*, *concat*, *invert*
- **classical graph operations**
  depth-first search (DFS), *single-source shortest path* (SSSP), *connect* (only keep accessible and coaccessible state), *strongly connected components* (SCCs)
- **operations on relations of sets**
  compose (filtered), *intersect*, *complement*
- **equivalence transformations**
  determinize, *minimize*, *remove-epsilons*
- **search algorithms**
  best, *n-best*
- **weight/probability-based algorithms**
  prune (based on forward/backward state potentials), *posterior*, *push* (push weights toward initial/final states), *failure* (given an acceptor/transducer defined over the tropical semiring converts ϵ-transitions to failure transitions)
- **diagnostic operations**
  count (counts states, final states, different arc types, SCCs, alphabet sizes, . . .)
- **input/output operations**
  supported input and/or output formats are: `AT&T` (currently, ASCII only), `binary` (fast, uses fixed byte-order), `XML` (slower, any encoding, fully portable), `memory-mapped` (also on-demand), `dot` (AT&T graphviz)

We will discuss some details and refer to the publication of the algorithms briefly. Most of the basic operations have a straigthforward implementation.

As arc labels are integers in the implementation and their meaning is bound to an appropriate symbolic alphabet, there is the need for symbolic mapping between different alphabets. Therefore the toolkit provides the lazy *map-input* and *map-output* transducers, which map the input and output arc indices of an automaton to be compatible with the indices of another given alphabet.

The implementations of all classical graph algorithms are based on the descriptions of (Mehlhorn, 1984) and (Cormen et al., 1990) and (Mohri and Riley, 2001) for *SSSP*. The general graph algorithms *DFS* and *SCC* are helpful in the realisation of many other operations, examples are: *transpose*, *connect* and *count*. However, counting the number of states of an automaton or the number of symbols of an alphabet is not well-defined in case of an infinite set of states or symbols.

SSSP and *transpose* are the only two algorithms without a lazy implementation. The result of *SSSP* is a list of state potentials (see also (Mohri and Riley, 2001)). And a lazy implementation for *transpose* would be possible if the data structures provide lists of both successor and predecessor arcs at each state. This needs either more memory or more computations and increases the size of the abstract interface for the lazy algorithms, so as a compromise we omitted this.

The implementations of *compose* (Pereira and Riley, 1996), *determinize* (Mohri, 1997), *minimize* (Mohri, 1997) and *remove-epsilons* (Mohri, 2001) use more refined methods to gain efficiency. All use at least the lazy *cache* transducer as they refer to states of the input transducer(s) more than once. With respect to the number of lazy transducers involved in computing the result, *compose* has the most complicated implementation. Given the implementations for the algorithmic transducers *cache*, *map-output*, *sort-input*, *sort-output* and *simple-compose* that assumes arc labels to be compatible and sorted in order to perform matching as fast as possible, the final implementation of *compose* in the RWTH FSA toolkit is given in figure 3. So, the current implementation of *compose* uses 6 algorithmic transducers in addition to the two input automata. *Determinize* additionally uses lazy *cache* and *sort-input* transducers.

The search algorithms *best* and *n-best* are based on (Mohri and Riley, 2002), *push* is based on (Mohri and Riley, 2001) and *failure* mainly uses ideas from (Allauzen et al., 2003). The algorithms *posterior* and *prune* compute arc posterior probabilities and prune arcs with respect to them. We believe they are standard algorithms defined on probabilistic networks and they were simply ported to the framework of weighted finite-state automata.

Finally, the RWTH FSA toolkit can be loosely interfaced to the AT&T FSM Library^{TM} through its ASCII-based input/output format. In addition, a new XML-based file format primarly designed as being human readable and a fast binary file format are also supported. All file formats support optional on-the-fly compression using *gzip*.

3.3 High-Level Interfaces

In addition to the C++ library level interface the toolkit also offers two high-level interfaces: a Python interface, and an interactive command-line interface.

The Python interface has been built using the SWIG interface generator (Beazley et al., 1996) and enables rapid development of larger applications without lengthy compilation of C++ code. The command-line interface comes handy for quickly applying various combinations of algorithms to transducers without writing any line of code at all. As the Python interface is mainly identical to the C++ interface we will only give a short impression of how to use the command-line interface.

The command-line interface is a single executable and uses a stack-based execution model (postfix notation) for the application of operations. This is different from the pipe model that AT&T command-line tools use. The disadvantage of using pipes is that automata must be serialized and get fully expanded by the next executable in chain. However, an advantage of multiple executables is that memory does not get fragmented through the interaction of different algorithms.

With the command-line interface, operations are applied to the topmost transducers of the stack and the results are pushed back onto the stack again. For example,

```
> fsa A B compose determinize draw -
```

reads `A` and `B` from files, calculates the determinized composition and writes the resulting automaton to the terminal in `dot` format (which may be piped to `dot` directly). As you can see from the examples some operations like *write* or *draw* take additional arguments that must follow the name of the operation. Although this does not follow the strict postfix design, we found it more convenient as these parameters are not automata.

4 Experimental Results

4.1 Comparison of Toolkits

A crucial aspect of an FSA toolkit is its computational and memory efficiency. In this section we will compare the efficiency of four different implementations of weighted-finite state toolkits, namely:

- RWTH FSA,
- RWTH FSM (predecessor of RWTH FSA),
- AT&T FSM Library^{TM} 4.0 (Mohri et al., 2000b),
- WFST (Adant, 2000).

We opted to not evaluate the FSA6.1 from (van Noord, 2000) as we found that it is not easy to install and it seemed to be significantly slower than any of the other implementations. RWTH FSA and the AT&T FSM Library^{TM} use on-demand computations whereas FSM and WFST do not. As the algorithmic code between RWTH FSA and its predecessor RWTH FSM has not changed much except for the interface of lazy transducers, we can also compare lazy versus non lazy implementation. Nevertheless, this direct comparison is also possible with RWTH FSA as it provides a static storage class transducer and a traversing deep copy operation.

Table 1 summarizes the tasks used for the evaluation of efficiency together with the sizes of the resulting transducers. The exact meaning of the different transducers is out of scope of this comparison. We simply focus on measuring the efficiency of the algorithms. Experiment 1 is the full expansion of the static part of a speech recognition search network. Experiment 2 deals with a translation problem and splits words of a “bilanguage” into single words. The meaning of the transducers used for Experiment 2 will be described in detail in Section 4.2. Experiment 3 is similar to Experiment 1 except for that the grammar transducer is exchanged with a translation transducer and the result represents the static network for a speech-to-text translation system.

All experiments were performed on a PC with a 1.2GHz AMD Athlon processor and 2 GB of memory using Linux as operating system. Table 2 summarizes the peak memory usage of the different toolkit implementations for the given tasks and Table 3 shows the CPU usage accordingly.

As can be seen from Tables 2 and 3 for all given tasks the RWTH FSA toolkit uses less memory and

Table 1: Tasks used for measuring the efficiency of the toolkits. Sizes are given for the resulting transducers (VM = Verbmobil).

Experiment		states	arcs
1	VM, $HCL \circ G$	12,203,420	37,174,684
2	VM, $C_1 \circ A \circ C_2$	341,614	832,225
3	Eutrans, $HCL \circ T$	1,201,718	3,572,601

computational power than any of the other toolkits. However, it is unclear to the authors why the AT&T Library^{TM} is a factor of 1800 slower for experiment 2. The numbers also do not change much after additionally connecting the composition result (as in RWTH FSA `compose` does not connect the result by default): memory usage rises to 62 MB and execution time increases to 9.7 seconds. However, a detailed analysis for the RWTH FSA toolkit has shown that the composition task of experiment 2 makes intense use of the lazy `cache` transducer due to the loop character of the two transducers C_1 and C_2.

It can also be seen from the two tables that the lazy implementation RWTH FSA uses significantly less memory than the non lazy implementation RWTH FSM and less than half of the CPU time. One explanation for this is the poor memory management of RWTH FSM as all intermediate results need to be fully expanded in memory. In contrast, due to its lazy transducer interface, RWTH FSA may allocate memory for a state only once and reuse it for all subsequent calls to the `getState()` method.

Table 2: Comparison of peak memory usage in MB (* aborted due to exceeded memory limits).

Exp.	FSA	FSM	AT&T	WFST
1	360	1700	1500	$> 1850^*$
2	59	310	69	$> 1850^*$
3	48	230	176	550

Table 3: Comparison of CPU time in seconds including I/O using a 1.2GHz AMD Athlon processor (* exceeded memory limits: given time indicates point of abortion).

Exp.	FSA	FSM	AT&T	WFST
1	105	203	515	$> 40^*$
2	6.5	182	11760	$> 64^*$
3	6.6	21	28	3840

4.2 Statistical Machine Translation

Statistical machine translation may be viewed as a weighted language transduction problem (Vidal, 1997). Therefore it is fairly easy to build a machine translation system with the use of weighted finite-state transducers.

Let f_1^J and e_i^I be two sentences from a source and target language respectively. Also assume that we have word level alignments $\mathcal{A}$ of all sentences from a bilingual training corpus. We denote with $e_{p_1}^{p_J}$ the segmentation of a target sentence e_1^I into phrases such that f_1^J and $e_{p_1}^{p_J}$ can be aligned monotoneously. This segmentation can be directly calculated from the alignments $\mathcal{A}$. Then we can formulate the problem of finding the best translation $\hat{e}_1^I$ of a source sentence as follows:

$$\begin{aligned}
\hat{e}_1^I &= \operatorname*{argmax}_{e_1^I} Pr(f_1^J, e_1^I) \\
&\approx \operatorname*{argmax}_{\mathcal{A}, e_{p_1}^{p_J}} Pr(f_1^J, e_{p_1}^{p_J}) \\
&= \operatorname*{argmax}_{\mathcal{A}, e_{p_1}^{p_J}} \prod_{f_j : j=1..J} Pr(f_j, e_{p_j} | f_1^{j-1}, e_{p_1}^{p_{j-1}}) \\
&\approx \operatorname*{argmax}_{\mathcal{A}, e_{p_1}^{p_J}} \prod_{f_j : j=1..J} Pr(f_j, e_{p_j} | f_{j-n}^{j-1}, e_{p_{j-n}}^{p_{j-n}})
\end{aligned}$$

The last line suggests to solve the translation problem by estimating a language model on a bilanguage (see also (Bangalore and Riccardi, 2000; Casacuberta et al., 2001)). An example of sentences from this bilanguage is given in Figure 5 for the translation task Vermobil (German $\rightarrow$ English). For technical reasons, ϵ-labels are represented by a \$ symbol. Note, that due to the fixed segmentation given by the alignments, phrases in the target language are moved to the last source word of an alignment block.

So, given an appropriate alignment which can be obtained by means of the pubically available GIZA++ toolkit (Och and Ney, 2000), the approach is very easy in practice:

1. Transform the training corpus with a given alignment into the corresponding bilingual corpus
2. Train a language model on the bilingual corpus
3. Build an acceptor A from the language model

The symbols of the resulting acceptor are still a mixture of words from the source language and phrases from the target language. So, we additionally use two simple transducers to split these bilingual words (C_1 maps source words f_j to bilingual words that start with f_j and C_2 maps bilingual words with the target sequence e_{p_j} to the sequences of target words the phrase was made of):

```
dann|$ melde|$ ich|I_am_calling mich|$ noch|$ einmal|once_more .|.
11U|eleven Uhr|o'clock ist|is hervorragend|excellent .|.
ich|I bin|have da|$ relativ|quite_a_lot_of frei|free_days_then .|.
```

Figure 5: Example corpus for the bilanguage (Verbmobil, German → English).

Table 4: Translation results for different tasks compared to similar systems using the alignment template (AT) approach (Tests were performed on a 1.2GHz AMD Athlon).

Task	System	Translation	WER [%]	PER [%]	100-BLEU	Memory [MB]	Time/Sentence [ms]
Eutrans	FSA	Spanish → English	8.12	7.64	10.7	6-8	20
	AT		8.25	-	-	-	-
FUB	FSA	Italian → English	27.0	21.5	37.7	3-5	22
	AT		23.7	18.1	36.0	-	-
Verbmobil	FSA	German → English	48.3	41.6	69.8	65-90	460
	AT		40.5	30.1	62.2	-	-
PF-Star	FSA	Italian → English	39.8	34.1	58.4	12-15	35
	AT		36.8	29.1	54.3	-	-

4. Split the bilingual phrases of A into single words:

$$T = C_1 \circ A \circ C_2$$

Then the translation problem from above can be rewritten using finite-state terminology:

$$e' = \texttt{project-output}(\texttt{best}(f \circ T))$$

Translation results using this approach are summarized in Table 4 and are being compared with results obtained using the alignment template approach (Och and Ney, 2000). Results for both approaches were obtaining using the same training corpus alignments. Detailed task descriptions for Eutrans/FUB and Verbmobil can be found in (Casacuberta et al., 2001) and (Zens et al., 2002) respectively. We use the usual definitions for word error rate (WER), position independent word error rate (PER) and BLEU statistics here.

For the simpler tasks Eutrans, FUB and PF-Star, the WER, PER and the inverted BLEU statistics are close for both approaches. On the German-to-English Verbmobil task the FSA approach suffers from long distance reorderings (captured through the fixed training corpus segmentation), which is not very surprising.

Although we do not have comparable numbers of the memory usage and the translation times for the alignment template approach, resource usage of the finite-state approach is quite remarkable as we only use generic methods from the RWTH FSA toolkit and full search (i.e. we do not prune the search space). However, informal tests have shown that the finite-state approach uses much less memory and computations than the current implementation of the alignment template approach.

Two additional advantages of finite-state methods for translation in general are: the input to the search algorithm may also be a word lattice and it is easy to combine speech recognition with translation in order to do speech-to-speech translation.

5 Summary

In this paper we have given a characterization of algorithms that produce a single finite-state automaton and bear an on-demand implementation. For this purpose we formally introduced the local property of such an algorithm.

We have described the efficient implementation of a finite-state toolkit that uses the principle of lazy algorithmic transducers for almost all algorithms. Among several publically available toolkits, the RWTH FSA toolkit presented here turned out to be the most efficient one, as several tests showed. Additionally, with lazy algorithmic transducers we have reduced the memory requirements and even increased the speed significantly compared to a non lazy implementation.

We have also shown that a finite-state automata toolkit supports rapid solutions to problems from the field of natural language processing such as statistical machine translation. Despite the genericity of the methods, statistical machine translation can be done very efficiently.

6 Shortcomings and Future Extensions

There is still room to improve the RWTH FSA toolkit. For example, the current implementation of determinization is not as general as described in (Allauzen and Mohri, 2003). In case of ambiguous input the algorithm still produces an infinite transducer. At the moment this can be solved in many

cases by adding disambiguation symbols to the input transducer manually.

As the implementation model is based on virtual C++ methods for all types of objects in use (semirings, alphabets, transducers and algorithmic transducers) it should also be fairly easy to add support for dynamically loadable objects to the toolkit.

Other semirings like the expectation semiring described in (Eisner, 2001) are supported but not yet implemented.

7 Acknowledgment

The authors would like to thank Andre Altmann for his help with the translation experiments.

References

Alfred V. Aho and Jeffrey D. Ullman, 1972, *The Theory of Parsing, Translation and Compiling*, volume 1, Prentice-Hall, Englewood Cliffs, NJ, 1972.

Arnaud Adant, 2000, *WFST: A Finite-State Template Library in C++*, http://membres.lycos.fr/adant/tfe/.

Cyril Allauzen, Mehryar Mohri, and Brian Roark, 2003, *Generalized Algorithms for Constructing Statistical Language Models*, In Proc. of the 41st Meeting of the Association for Computational Linguistics, Sapporo, Japan, July 2003.

Cyril Allauzen and Mehryar Mohri, 2003, *Generalized Optimization Algorithm for Speech Recognition Transducers*, In Proc. of the IEEE Int. Conf. on Acoustics, Speech, and Signal Processing, pp. , Hong Kong, China, April 2003.

Srinivas Bangalore and Giuseppe Riccardi, 2000, *Stochastic Finite-State models for Spoken Language Machine Translation*, In Proc. of the Workshop on Embedded Machine Translation Systems, pp. 52–59, 2000.

David Beazley, William Fulton, Matthias Köppe, Lyle Johnson, Richard Palmer, 1996, *SWIG - Simplified Wrapper and Interface Generator*, Electronic Document, http://www.swig.org, February 1996.

F. Casacuberta, D. Llorens, C. Martinez, S. Molau, F. Nevado, H. Ney, M. Pasto, D. Pico, A. Sanchis, E. Vidal and J.M. Vilar, 2001, *Speech-to-Speech Translation based on Finite-State Transducer*, In Proc. IEEE Int. Conf. on Acoustics, Speech and Signal Processing, pp. 613-616, Salt Lake City, Utah, May 2001.

Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest, 1990, *Introductions to Algorithms*, The MIT Press, Cambridge, MA, 1990.

Jason Eisner, 2001, *Expectation Semirings: Flexible EM for Finite-State Transducers*, In Proc. of the ESSLLI Workshop on Finite-State Methods in NLP (FSMNLP), Helsinki, August 2001.

Free Software Foundation, 1991, *GNU General Public License, Version 2*, Electronic Document, http://www.gnu.org/copyleft/gpl.html, June 1991.

Takaaki Hori, Chiori Hori and Yasuhiro Minami, 2003, *Speech Summarization using Weighted Finite-State Transducers*, In Proc. of the European Conf. on Speech Communication and Technology, Geneva, Switzerland, September 2003.

Vincent Le Maout, 1998, *ASTL: Automaton Standard Template Library*, http://www-igm.univ-mlv.fr/l̃emaout/.

Kurt Mehlhorn, 1984, *Data Structures and Efficient Algorithms*, Chapter 4, Springer Verlag, EATCS Monographs, 1984, also available from http://www.mpi-sb.mpg.de/m̃ehlhorn/DatAlgbooks.html.

Mehryar Mohri, 1997, *Finite-State Transducers in Language and Speech Processing*, Computational Linguistics, 23:2, 1997.

Mehryar Mohri, Fernando C.N. Pereira, and Michael Riley, 2000, *Weighted Finite-State Transducers in Speech Recognition*, In Proc. of the ISCA Tutorial and Research Workshop, Automatic Speech Recognition: Challenges for the new Millenium (ASR2000), Paris, France, September 2000.

Mehryar Mohri, Fernando C.N. Pereira, and Michael Riley, 2000, *The Design Principles of a Weighted Finite-State Transducer Library*, Theoretical Computer Science, 231:17-32, January 2000.

Mehryar Mohri and Michael Riley, 2000, *A Weight Pushing Algorithm for Large Vocabulary Speech Recognition*, In Proc. of the European Conf. on Speech Communication and Technology, pp. 1603–1606, Åalborg, Denmark, September 2001.

Mehryar Mohri, 2001, *Generic Epsilon-Removal Algorithm for Weighted Automata*, In Sheng Yu and Andrei Paun, editor, 5th Int. Conf., CIAA 2000, London Ontario, Canada. volume 2088 of Lecture Notes in Computer Science, pages 230-242. Springer-Verlag, Berlin-NY, 2001.

Mehryar Mohri and Michael Riley, 2002, *An Efficient Algorithm for the N-Best-Strings Problem*, In Proc. of the Int. Conf. on Spoken Language Processing, pp. 1313–1316, Denver, Colorado, September 2002.

Franz J. Och and Hermann Ney, 2000, *Improved Statistical Alignment Models*, In Proc. of the 38th Annual Meeting of the Association for Computational Linguistics, pp. 440-447, Hongkong, China, October 2000.

Fernando C.N. Pereira and Michael Riley, 1996, *Speech Recognition by Composition of Weighted Finite Automata*, Available from http://xxx.lanl.gov/cmp-lg/9603001, Computation and Language, 1996.

Gertjan van Noord, 2000, *FSA6 Reference Manual*, http://odur.let.rug.nl/v̈annoord/Fsa/.

Enrique Vidal, 1997, *Finite-State Speech-to-Speech Translation*, In Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing, pp. 111–114, Munich, Germany, 1997.

Richard Zens, Franz J. Och and H. Ney, 2002, *Phrase-Based Statistical Machine Translation*, In: M. Jarke, J. Koehler, G. Lakemeyer (Eds.) : KI - 2002: Advances in artificial intelligence. 25. Annual German Conference on AI, KI 2002, Vol. LNAI 2479, pp. 18-32, Springer Verlag, September 2002.