HFST Runtime Format - A Compacted Transducer Format Allowing for Fast Lookup

Miikka Silfverberg and Krister Lindén

University of Helsinki
Helsinki, Finland
miikka.silfverberg,krister.linden@helsinki.fi

Abstract. Lexical transducers form part of most language-aware applications, which means that less time spent on lexical lookup will have wide-ranging effects. The efficiency of a morphological analyzer stems mainly from the properties of the underlying transducer, but the way its transition sets are represented also plays a large role, since this determines how efficiently transitions can be accessed. We consider three principal ways to represent transition sets leading to three different transducer formats. We call them the linked list model, the ordered array model and the random access model. As test material for evaluating the speed allowed by the formats, we use two full-fledged morphological descriptions: *Morphalou for French*, which is a full-form morphological lexicon resulting in an acyclic transducer, and the *Divvun lexicon for Northern Sámi*, which is a morphological description with a productive compounding mechanism resulting in a cyclic transducer. Both have sufficient coverage to be used in real applications. The random access model using Liang compaction implemented in the HFST−*Helsinki Finite-State Technology* runtime is the fastest and achieves a throughput of 119 000-408 000 w/s on running text with a moderate memory consumption.

1 Introduction

Lexical transducers form part of most language-aware applications which means that less time spent on lexical lookup will have wide-ranging effects. Especially when continuously processing large amounts of data, the speed of lexical access becomes essential, e.g. indexing large amounts of text for information retrieval, or for on line processing of document collections for data-mining purposes. The efficiency of a morphological analyzer stems mainly from the properties of the underlying transducer (e.g. whether its input side is deterministic), but the way its transition sets are represented also plays a large role, since this determines how efficiently transitions can be accessed.

We consider three principal ways to represent transition sets leading to three different transducer formats. We call them the linked list model, the ordered array model and the random access model according to the kind of access they allow. All are well known in literature, see e.g. [5]. The linked list model represents transition sets as linked lists. This allows for easy modification of the transducer, but it makes lookup slow requiring linear time according to the number of transitions of a state. The ordered array model represents the transitions of each state in an array sorted according to the input symbol. This allows using binary search to find transitions requiring only logarithmic lookup time. The random access model makes it possible to find transitions for a given input symbol in a state in constant time. A drawback of this format is that the transducers may require a lot of space unless properly compacted[1]. Further properties of the formats are discussed in Section 2.

Kiraz [4] describes both the ordered array and random access models for finite-state automata (FSAs), but he does not consider any method for space reduction for the the random access model. Liang [7] describes a compacted random access model for FSAs when storing lexicons for the hyphenation algorithm of the TEX82 typesetting system. Tarjan and Yao [11] and Aho and Ullman [1] describe similar compactifications. Liang's compaction for FSAs is not directly applicable to finite-state transducers (FSTs). We can, however, modify the random access model for FSAs by factoring the transducer into a deterministic input acceptor (i.e. an FSA) and a (possibly non-deterministic) array of transitions. This forms the basis for the HFST runtime format. The space requirement of the input acceptor in random access format can be reduced considerably using the Liang compaction. We note that the transitions in the transition array can be stored densely as they are indexed by the input acceptor and each output always needs to be considered for a particular input. We make one additional optimization, i.e. states with transitions with one single input symbol only need to be represented in the transition array. Additional details of the HFST runtime format are presented in Section 3.

As all HFST tools the runtime format and drivers are utf-8 compatible.

We evaluate the speed allowed by the formats using two full-fledged morphological descriptions described in section 4: the full-form lexicon *Morphalou for French*, and the *Divvun lexicon for Northern Sámi*, which has a productive compounding mechanism. In evaluation HFST runtime format represents the random access format. The HFST standard format used in the HFST finite-state calculus tools represents the linked list format with unordered transition lists. The compact format of *Stuttgart Finite-State Transducer Tools (*SFST*)* [10] represents an ordered array transducer similar to the one presented by Kiraz [4]. For a comparison of their speed and memory consumption, see Section 5

2 Sketches of Three Different Transducer Formats

We consider three different transducers formats: *the linked list format*, *the ordered array format* and *the random access format.* All of the transducer formats share two aspects, which are central for our needs. They can easily represent

[1] We use the terms *packing* for sequentially storing items in an archive, *compacting* for storing by removing empty or redundant space and *compressing* for storing by encoding with fewer bits

both deterministic and non-deterministic transducers and they require minimal changes in order to allow for weights in transitions and final states.

Linked list transducers are easy to modify and to maintain. Adding and removing states and transitions is fast. Hence their main use is during construction of transducers, whereas they are comparatively slow for runtime lookup purposes. The HFST *standard format* represents the linked list format.

In comparison to linked list transducers, ordered array transducers are rigid. Since ordered array transducers build on an array structure, adding or removing transitions and states may require some extra work. Adding and removing transitions and states is efficient only near the end of the array. Instead transducers in the ordered array format allow for fast load times. Analysis is fast as well, since suitable transitions may be found using a binary search algorithm. The *Stuttgart Finite-State Transducer Tools (*SFST*) compact format* (Schmidt [10]) represents the ordered array format with some additional optimizations, which reduce the size of the transducer binaries. The ordered array format and its possible optimizations are more thoroughly described by Kiraz [4].

The random access format represents a further optimization for lookup time in transducers. Transducers in the random access format represent all states as equal length random access arrays. The arrays may be indexed, using the symbols in the input alphabet of the transducer. Indexing the array with an input symbol gives an address to the set of transitions of the state, with that input symbol. A special address $\emptyset$ is reserved to signify an empty transition set in case the state doesn't have transitions with a given input symbols. The relevant transitions for a given input symbol may always be found in constant time regardless of the number of transitions in a state. The down-side of constant time transition recovery is the huge size of the transducer, since every input symbol in the input alphabet requires an entry in every state of the transducer. Fortunately, the space requirement may be dramatically reduced with little or no effect on lookup speed using a technique first introduced by Liang [7] for compacting lexicon automata used in hyphenation. The HFST runtime format[2] represents a compacted version of the random access format, where empty space in one state may be reused for storing input symbol entries of another state under certain conditions. We describe the compacted format in section 3.

2.1 The Linked List Format

Transducers in the linked list format are sets of states. Each state is a structure having a unique address, a bit indicating finality and a set of transitions. If the transducer is weighted, the binary finality-bit is replaced by a final weight from the relevant weight semi-ring. Each transition consists of a label and a target state address. If the transducer is weighted, the transitions also include a transition weight.

The transition sets are represented as a linked list. Since it is fast to add new transitions to a linked structures, the transition sets in linked list transducer can

[2] https://kitwiki.csc.fi/twiki/bin/view/KitWiki/HfstRuntimeBinaryFormat

easily be modified either by insertion or deletion. Lookup among the transitions requires iteration through all transitions until a transition with the correct input symbol is found. Even if the transitions are sorted, all of them need to be checked in the worst case. Lookup in a state with n transitions is $\mathrm{O}(n)$.

An optimized version of the linked list format represents the transitions sets of each state as vectors, which allows for binary search of transitions. Hence lookup in a state with n transitions in the optimized version is $\mathrm{O}(\log n)$. Removing transitions can be done by marking them as deleted, but adding new transitions either requires shifting the old transitions towards the end of the transition vector in order to maintain the sorting for an efficient lookup, or adding new transitions at the end of the transition vector and periodically sorting.

2.2 The Ordered Array Format

The ordered array format represents transducers in a few large arrays. The transition array, stores all transitions in the transducers. The transitions for each state occupy a dense sub-array in the transition array. The transitions are sorted according to input symbol in order to allow binary search. Lookup in a state with n transitions is $\mathrm{O}(\log n)$. Transitions consist of input and output symbols as well as pointers to a second array, the state array, which stores the index of the first transition of each state in the transition array. A third array stores the finality bits of the states.

Modifying an ordered array transducer in any way is generally inefficient but both loading the transducer into memory and lookup of forms are fast. Loading is fast, since the entire transducer can be read into memory in one read operation and it can be used directly. The ordered array format is mainly used for runtime lookup applications.

Bit-level compression techniques may be used to reduce the size of ordered array transducers on hard-drive as explained by Kiraz [4]. Weights may be added e.g. by adding a weight array in parallel to the transition array and replacing the array for finality bits with an array of final weights.

2.3 The Random Access Format

In the random access format the transitions of a state are stored in an array having one entry for every input symbol in the input alphabet of the transducer. If the state has transitions with a given symbol, its entry contains a pointer to a set of transitions with that input symbol, otherwise the entry is $\emptyset$. The first entry in the input symbol array codes finality.

The transitions with an input symbol a in a state are found by indexing the input symbol array of that state with symbol a. Lookup of transitions is therefore $\mathrm{O}(1)$ w.r.t. the number of transitions in a state. The downside of constant time transition access is the large size of the transducer. For a transducer with input alphabet Σ and state set Q, the input symbol array occupies $|\Sigma| \times |Q| \times p$ bytes, where p is the size of the pointer to the transition array. E.g. a lexical transducer with 150 000 states and 100 input symbols, where the pointers occupy 4 bytes, needs approximately 57 MB in order to store the input symbol arrays (1 MB is 1024 × 1024 bytes). In addition it also needs to store the actual transitions.

Both Liang and Kiraz note that linguistic transducers have far less transitions than they could have in theory. Accordingly most space in the random access transducers is taken by ∅ pointers. The transducers can be compacted considerably, using an algorithm by Liang, whilst maintaining random access. The compactification makes the transducer very hard to modify, so the compressed random access format is mainly suited as an end storage format for transducers which need not be modified (e.g. lexical transducers).

3 The HFST Runtime Format

The HFST Runtime Format combines the constant time requirement for finding transitions, in a random access transducer, with a compaction technique, which reduces the size of the transducer almost to the same level as a linked list transducer. We first factor the transducer into a deterministic finite state acceptor and a transition array. The finite state acceptor is then coded as a compacted random access automaton.

The factoring of a state p' in the original transducer is illustrated in Table 1. Every state p' in the original transducer corresponds directly to a state p in the acceptor. The acceptor has one transition for each input symbol of state p'. A transition with input symbol a in the acceptor corresponds to a set of transitions in the transition array (the transitions at indices q and $q + 1$ in Table 1). All the transitions with input symbol a in the state p' of the original transducer are stored contiguously in the transition array beginning at index q.

Table 1. The state p' in the original acceptor (above), the state p in the input acceptor (middle) and the corresponding transition array (below)

p'		
$a{:}a$	$a{:}b$	$c{:}c$

p			
$a{:}\ \ q$	...	$c{:}\ \ q+2$	...

	q	$q+1$	$q+2$	
...	$a{:}a\ \ s_1$	$a{:}b\ \ s_2$	$c{:}c\ \ s_2$	...

E.g. the state p' corresponding to the acceptor state p in Table 1 has transitions with input symbols a and c. There are 2 transitions with input symbol a in state p corresponding to a state p' in the original transducer. The transitions with symbol a can be found at index q in the transition array and the transitions with input symbol c can be found at index $q + 2$ in the transition array.

3.1 The Transition Index Array

For input symbols with transitions, the corresponding entry in the acceptor is an index of the transition array. The acceptor in the random access format has entries $\emptyset$ for input symbols without corresponding transitions. This is illustrated in in Table 2. We interchangeably call the acceptor a *transition index array.*

Table 2. The state p in the input acceptor, i.e. transition index array, (above) and the corresponding transition array (below)

p:			
a: q	b: $\emptyset$	c: $q+2$	d: $\emptyset$

	q	$q+1$	$q+2$	
...	a:a s_1	a:b s_2	c:c s_2	...

If we follow the example in Table 2, and store the input symbol together with the pointer to the relevant transition set, in each state and for each input symbol, we may utilize the entries with pointer $\emptyset$ in the transition index array for overlaying symbol pointer pairs of other states. Doing this, we can compact the transition index array considerably.

We observe that indexing with a in state p in Table 2 gives the symbol a and the index q. Indexing with b gives the symbol b and the empty index $\emptyset$, which signifies that there are no transitions for the input symbol b. We also observe that another way to signify the lack of transitions is to mark the symbol b as something else, e.g. a or c. These two observations permit the overlaying of states in the transition index array, which is the compacted random access format suggested by Liang.

Table 3. The state v in the input acceptor

v:			
a : w	b: $\emptyset$	c: $\emptyset$	d: $w+s$

The state v in Table 3 only has transitions with input symbols a and d. The states p and v in the input acceptor can be compacted without losing information about their transitions, as shown in Tables 4 and 5, by letting the states partially overlap.

After compaction, indexing with symbols a or c in state p still gives the input symbols back and the relevant transition indices. Indexing with b gives input symbol a, which signifies that the index obtained w does not belong to

Table 4. The states p and v in the input acceptor

p:				
a: q	b: $\emptyset$	c: $q+r$	d: $\emptyset$	
	v:			
	a : w	b: $\emptyset$	c: $\emptyset$	d: $w+s$

Table 5. The partially overlapping states p and v still contain all necessary information to find the transitions in the transition index array.

p:				
	v:			
a: q	a : w	c: $q+r$	c: $\emptyset$	d: $w+s$

state p. Indexing with d gives the input symbols c. Again we know that state p has no transitions with input symbol d.

Similarly we see that the transitions for the state v are preserved in the compaction. In fact, it is easy to see that transitions are never lost or misinterpreted provided that two states do not begin at the same index and two positions in the transition index array corresponding to original transitions are not superimposed.

With compaction, the ability to change the transition sets of states is severely limited, hence the compacted random access format is not suited for transducers, whose states need to be modified. The HFST runtime format is intended to be used as the end format of a morphological analyzer, but it can also be used for other lexical lookup applications like stemmers and translation dictionaries.

3.2 The Transition Array

Compacted random access transducers and ordered array transducers have almost identical transitions arrays. The difference is that in the ordered array format, there is no additional array indicating the boundaries between states in the compacted random access format. Hence the transitions of two adjacent states need to be separated, e.g. by a $\emptyset$ transition.

To see why the separating transition is needed, consider the fragment of a transition array in Table 6. The transitions at indices $q-2$ and $q-1$ belong to one state and the transitions at $q+1$ belong to another. If there were no $\emptyset$ transitions between the transitions beginning at indices $q-2$ and $q+1$, the transitions beginning at $q+1$ would be mistaken for a continuation of the transitions beginning at $q-2$, since the input symbol a for the transitions at $q-2$ and $q-1$ is the same as the input symbol for the first transitions at $q+1$. A $\emptyset$ transition always delimits the search for further transitions of a state.

As an optimization, states with a single input symbol may be omitted in the transition index array. Transitions to such states point directly to an index in

Table 6. Two adjacent states in a transition array with the same input symbol in neighboring transitions need to be separated by a $\emptyset$ transition.

	$q-2$:		$q-1$:		q		$q+1$:		
...	a:a	p_{n-1}	a:b	p_n	$\emptyset$:$\emptyset$	$\emptyset$	a:a	r_0	...

the transition array instead of an index in the transition index array like other transitions. The states which are omitted need to code for finality vs. non-finality in the transition array itself. The $\emptyset$ transition preceding the state is used for this purpose. The input and output symbols are still $\emptyset$, but the transition index is 1 if the state is final and $\emptyset$ otherwise.

By reserving some extra space, weights may be associated with transitions.

3.3 An Example of a Compacted Transducer in HFST format

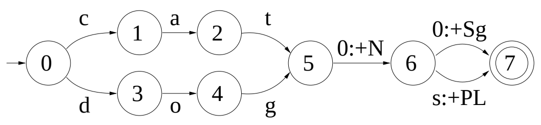

Fig. 1. A very small morphological analyzer T with input alphabet $\Sigma = \{0, a, c, d, g, o, s, t\}$

We give the transition index array and transition array (in Table 7) for transducer T in Figure 1. States 0 and 6 have entries both in the index array and the transition array, since they have transitions with more than one input symbol. We use roman numerals to index the transition index array and Arabic numerals to index the transition array.

4 Test Data

Our test material consists of two morphological analyzers, one for French and the other for Northern Sámi together with text-corpora to be analyzed. The morphological analyzer for French was converted to a Xerox LexC lexicon description format from an existing XML lexicon *Morphalou*[3] and compiled using HFST-LexC[4]. It recognizes some 550 000 word-forms. There is no productive compounding mechanism. The resulting transducer is acyclic. The morphological analyzer for Sámi was compiled from a Xerox LexC lexicon description and TwolC two-level morphology grammar description using HFST-LexC

[3] http://www.cnrtl.fr/lexiques/morphalou/
[4] https://kitwiki.csc.fi/twiki/bin/view/KitWiki/HfstLexC

Table 7. The first two rows show the index arrays of states 0 and 6 of T. Below them are the compacted index array and transition array for the transducer in Figure 1.

∅ ∅	0: ∅	a: ∅	c: 0	d: 1	g: ∅	o: ∅	s: ∅	t: ∅				
				∅ ∅	0: 13	a: ∅	c: ∅	d: ∅	g: ∅	o: ∅	s: 14	t: ∅

The transition index array of T:

∅ ∅	0: ∅	a: ∅	c: 0	d: 1	0: 13	a: ∅	c: ∅	d: ∅	g: ∅	o: ∅	s: 14	t: ∅
I	II	III	IV	V	VI	VII	VIII	IX	X	XI	XII	XIII

The transition array of T:

c:c 2	d:d 6	∅ ∅	a:a 4	∅ ∅	t:t 10	∅ ∅	o:o 8	∅ ∅
0	1	2	3	4	5	6	7	8
g:g 10	∅ ∅	0:+N V	∅ ∅	0:+Sg 15	s:+Pl 15	∅ 1		
9	10	11	12	13	14	15		

and HFST-TwolC[5]. The lexicon and grammar were developed by the *Divvun project*[6]. The LexC lexicon contains some 105 000 entries. The analyzer also has a productive compounding mechanism, which means that the resulting transducer is cyclic. Regarded as automata with pair-labels in transitions, both analyzers are deterministic, but they are not subsequential transducers.

In order to evaluate the effects of using transducers encoded in the HFST runtime format, we also produced transducer binaries for the morphological analyzers in the HFST standard format and the SFST compact format. The sizes of the binaries are compared in Table 8. Both the HFST standard format and the SFST compact format are dense transducer formats as explained above. Hence both of them require less space than the HFST runtime format.

Table 8. Size of the morphological analyzers in the different transducer formats.

Language	HFST Standard	SFST Compact	HFST Runtime
Northern Sámi	2.9 MB	1.5 MB	4.8 MB
French	1.7 MB	0.8 MB	2.9 MB

The difference in storage space requirements for transducers in the SFST compact format and the HFST standard format stems from the fact that the compact SFST format uses a number of different compression mechanisms, among others a bit-level compression of transitions on the hard-drive. The compression is undone when the transducer is read into memory.

[5] https://kitwiki.csc.fi/twiki/bin/view/KitWiki/HfstTwolC
[6] http://www.divvun.no

The storage space requirement for the HFST runtime format is the highest, as the compaction of the transition index array is never perfect. In addition, the runtime format does not currently use bit-level compression of transitions to reduce the size on hard-drive. To make a fair evaluation of the effectiveness of the compaction of the HFST runtime format, one should compare the size of a compacted runtime format transducer with the size of the uncompacted one. The uncompacted runtime binary for the French analyzer is 26 MB and the uncompacted binary for the Northern Sámi analyzer is 61 MB. This means that the sizes of the compacted analyzers for French and Northern Sámi are only about 11 % and 8 % of the original sizes.

We tested the analyzers using running text. We used the *Europarl parallel corpus* for French and English [6] to test the analyzer for French. The Europarl corpus for French contains some 45 000 000 words. Of the words 94.26 % received an analysis and on average there were 1.68 analyses per word. Of all words, 2.72 % contained symbols, which were not known to the morphological analyzer. The remaining 3.02 % of the words were unknown. We used a corpus containing some 2 200 000 words of running text to test the analyzer for Sámi. Of the 2 200 000 words 86.11 % received at least one analysis and there were 2.62 analyses per word on average. Of all words 7.64 % could not be tokenized. The rest of the words, i.e. 6.25 % could be tokenized, but were still rejected as unknown.

We evaluate the formats using running text, since a morphological analyzer usually forms a part of a wider application (e.g. tagger or disambiguator). Therefore it is important to show how the analyzer functions as a part of a system dealing with running text.

5 Lookup Performance

The results of out tests are shown in Table 9. All tests were conducted on an Intel computer with a Xeon E5450 64 bit 3.00 GHz CPU and 64 GB of memory. In order to monitor the memory consumption, we used the GNU `top` command. The times were extracted using the GNU `time` command in a command line Unix script.

Table 9. Above: Analysis rate, in input words per second, of the morphological analyzers in the different transducer formats above. Below: Memory consumption of the analysis algorithms for the morphological analyzers in the different transducer formats.

Language	HFST Standard	SFST Compact	HFST Runtime
Northern Sami	55 000 w/s	97 000 w/s	119 000 w/s
French	146 000 w/s	270 000 w/s	408 000 w/s

Language	HFST Standard	SFST Compact	HFST Runtime
Northern Sami	21 MB	5 MB	31 MB
French	14 MB	3 MB	18 MB

6 Discussion and Further Research

Further speed would be gained by all the formats by making sure that the input languages of the lexicons are as minimal and deterministic as possible, by subsequentializing the lexical transducers [2]. If the transducers were p-subsequential, the division into the transition index array and the transition array would not be necessary, since each input symbol would correspond to at most one transition in a state. This could have a profound impact on the speed of lookup, since lookup would never need to backtrack.

All transducers cannot be subsequentialized. Roche and Schabes give an example in text annotation [9] (p. 48), but morphological phenomena, e.g. compounding, can also give rise to non-subsequentializable transducers. Hence the input side of the transducer cannot always be deterministic and both the transition index array and the transition array are needed[7]. However, there are special cases (e.g. acyclic transducers) where a subsequential transducer suffices, which allows for a great optimization. Subsequentialization is not likely to change the relative speeds of the three transducer formats, since each format benefits from the deterministic input side.

If minimal disk space is crucial, the storage requirement for the HFST runtime format could probably be considerably reduced using similar compression techniques as SFST utilizes and Kiraz [4] proposes. We believe this could nearly halve the size of the binaries, though it wouldn't effect memory consumption in runtime. If some speed may be sacrificed but main memory storage is crucial, as may still be the case in portable or integrated devices, another idea that might be worth while investigating is lazy decompression of states. States in the transducer could be more or less decompressed depending on whether they have been visited during the lookup process. It is likely that the great majority of states never get visited during any given lookup.

The current implementation codes input symbols using two byte integers. In some applications, specifically syntax, there might be more than 2^16 input symbols. Hence input symbols would have to be coded using larger data types. An alternative would be to use methods like stretching [3] to reduce the number of input symbols in the transducer, although this would probably slow down lookup.

To evaluate the HFST runtime format we compared it with two other open source transducer formats. A more thorough evaluation would also include proprietary formats like the runtime format for Xerox finite state tools.

7 Conclusions

Lexical transducers form part of most language-aware applications, which means that less time spent on lexical lookup will have wide-ranging effects. The effi-

[7] Random access in the current implementation requires that all entries in the transition index array are equally long. Hence many transitions with the same input symbol cannot easily be coded in the transition index array.

ciency of a transducer format depends greatly on the way its transition sets are represented. We considered three principal ways to represent transition sets, i.e. the linked list model, the ordered array model and the random access model. As test material for evaluating the speed allowed by the formats, we used two full-fledged morphological descriptions. Both have sufficient coverage to be used in real applications. While some further compression is possible without loss of speed and some additional optimizations, we were able to achieve the fastest throughput with the random access model implemented using Liang compaction. The speed was 119 000-408 000 w/s on running text with a moderate memory consumption.

Acknowledgments

This research was funded by the Finnish Academy and the Finnish Ministry of Education. We would like to thank Kimmo Koskenniemi who brought our attention to the Liang compaction and the anonymous referees for their comments on the paper. We would also like to thank our colleagues Jan Guillen and Petri Uusitalo for interesting discussions and insightful ideas on various aspects of optimization of runtime transducers.

References

1. Alfred Aho and Jeffrey Ullman. 1977. Principles of Compiler Design (sect. 3.8 and 6.8).
2. Cyril Allauzen and Mehryar Mohri. 2003. p-Subsequentiable Transducers. In *Seventh International Conference, CIAA 2002*. Eds. Jean-Marc Champarnaud and Denis Maurel, 24–34.
3. Noud de Beijer, Bruce Watson and Derrick Kourie. 2003. Streching and Jamming of Automata. In *Proceedings of the 2003 annual conference of the South African institute of computer scientists and information technologists on Enablement through technology*, 198–207.
4. George Anton Kiraz. 1999. Compressed Storage of Sparse Finite-State Transducers. In *Workshop on Implementing Automata WIA99 - Pre-Proceedings*, 109–121.
5. Donald E. Knuth. 1968. *The Art of Computer Programming - Fundamental Algorithms, Vol 1.* Addison-Wesley Publishing Company.
6. Philipp Koehn. Europarl: A Parallel Corpus for Statistical Machine Translation. 2005. In *MT Summit 2005.*
7. Franklin Mark Liang. 1983. Word Hy-phen-a-tion by Com-put-er. PhD Thesis. Department of Computer Science, Stanford University.
8. Mehryar Mohri 1997. Finite-State Transducers in Language and Speech Processing. In *ACL*
9. Emmanuel Roche and Yves Schabes. 1997. Introduction. *In Finite State Language Processing* (eds. Emmanuel Roche and Yves Schabes).
10. Helmut Schmidt. 2005. A Programming Language for Finite State Transducers. In *Proceedings of the 5th International Workshop on Finite State Methods in Natural Language Processing, FSMNLP 2005.*
11. Robert Tarjan and Andrew Yao. 1979. Storing a Sparse Table. In *Communications of the ACM.*