A General Weighted Grammar Library

Cyril Allauzen, Mehryar Mohri, and Brian Roark

AT&T Labs – Research, Shannon Laboratory
180 Park Avenue, Florham Park, NJ 07932-0971
{allauzen, mohri, roark}@research.att.com
http://www.research.att.com/~{allauzen, mohri, roark}

Abstract. We present a general weighted grammar software library, the *GRM Library*, that can be used in a variety of applications in text, speech, and biosequence processing. The underlying algorithms were designed to support a wide variety of semirings and the representation and use of very large grammars and automata of several hundred million rules or transitions. We describe several algorithms and utilities of this library and point out in each case their application to several text and speech processing tasks.

1 Introduction

The statistical methods used in text and speech processing [18] or in bioinformatics [9] require the representation and use of models that are given either directly as weighted automata or as a result of the approximation and compilation of more powerful grammars such as probabilistic context-free grammars. In all cases, the weights play a crucial role in their definition and use, in particular because they can be used to rank alternative sequences.

This constituted our original motivation for the creation of a general *weighted grammar* library and the design of essential algorithms for creating, modifying, compiling, and approximating large weighted statistical or rule-based grammars. The algorithms of our software library, *GRM Library*, were designed to support a wide variety of semirings, thus weight sets. While keeping a high degree of generality, the algorithms were also designed to be very efficient to support the representation and use of grammars and automata of several hundred million rules or transitions. The representations and functions of a general weighted-transducer library (the FSM library [17]), served as the basis for the design of the GRM library.

Another motivation for the design of the GRM library was the need for general text and automata processing algorithms, which, in many cases, constitute the first step of the creation of a statistical grammar. An example is the requirement to compute from the input, the counts of some fixed sequences to create statistical language models. When the input is not just text, but a collection of weighted automata output by a speech recognizer or an information extraction system, novel algorithms and utilities are needed.

In the following, we describe several algorithms and utilities of the GRM library and point out in each case their application to several text and speech processing tasks. Some of the algorithms and utilities of an older version of this library, e.g., the algorithms and utilities for the compilation of weighted context-dependent rules, were presented elsewhere [15]. Here we describe three categories of algorithms and utilities of the library: local grammar and text processing utilities, context-free grammar compilation and approximation, and statistical language modeling algorithms and tools.

2 Design

The core foundation of the GRM library is the FSM library [17]. Both libraries are implemented in C and share the same data representations, the same binary file format and the same command-line interface style. In the FSM library, the memory representation of a weighted automaton or transducer is determined by the use of an FSM class that defines methods for accessing and modifying it. The efficient implementation of several algorithms required the definition of new FSM classes in the GRM library: the *edit*, *replace* and *failure* classes. The latter will be described in this article, the reader can refer to the documentation for the other classes.

3 Local Grammars and Text Processing

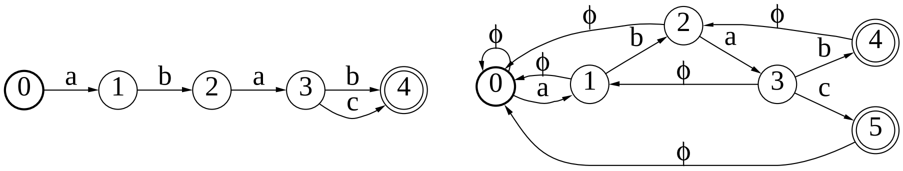
The GRM library includes several utilities for text processing. This section briefly reviews the relevant utilities.

3.1 Failure transitions

There exists a general technique for representing the transitions of automata in an implicit manner, which can lead to substantial savings in space. The method is based on the use of *failure transitions.* A failure transition is a specific type of transitions with the semantic of 'otherwise': it is taken when no regular transition with the desired input label is found. Failure transitions were popularized by [1] and are used to represent local grammars (see section 3.2) and backoff language models (see section 5.3).

The use of failure transitions is made possible in the GRM library through a dedicated FSM class, the *failure class.* The utility `grmfailure` can convert a regular FSM representation to a failure class representation by interpreting transitions labeled with the symbol `phi` specified by the option `-p` as failure transitions:

```
grmfailure -p phi A.fsm > A.failure.fsm
```



(a) (b)

Fig. 1. (a) A deterministic finite automaton A and (b) a deterministic automaton recognizing $\Sigma^* A$ where transitions labeled with ϕ are failure transitions.

3.2 Local Grammars

Algorithm. Let A be a deterministic finite automaton and let $L(A)$ be the regular language accepted by A. An algorithm constructing a compact representation of the deterministic automaton representing $\Sigma^* L(A)$ using failure transitions was proposed in [14]. This algorithm can be seen as an extension to the case of an arbitrary deterministic automaton A of the classical algorithms of [12, 1] which were designed for a string or a tree. When A is a tree, its complexity coincides with that of [1]: it is linear in the sum of the lengths of the strings accepted by A.

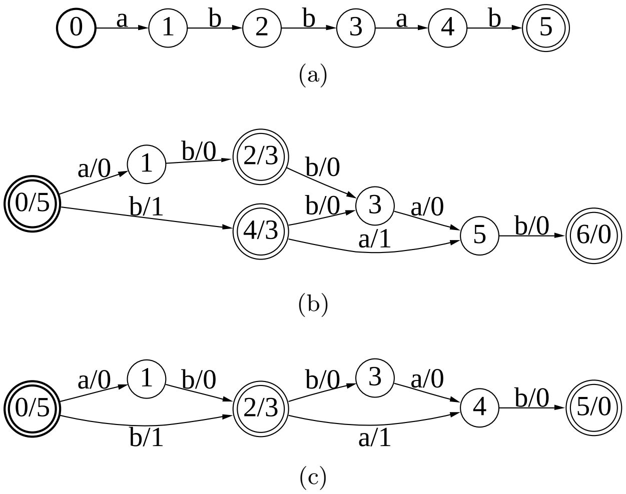
Utility. The algorithm of [14] was implemented in the GRM Library. The utility `grmlocalgrammar` takes as input a deterministic finite automaton A and returns a deterministic finite automaton recognizing $\Sigma^* A$ represented with failure transitions. The symbol used to label the failure transitions can be specified through the option `-p`:

```
grmlocalgrammar -p phi A.fsm > sigma-star.A.fsm
```

Examples and Applications. A deterministic finite automaton A is given Figure 1(a) and the corresponding automaton recognizing $\Sigma^* A$ is given Figure 1(b), the failure transitions being labeled with ϕ. The main applications of local grammars are string-matching [1, 14] and disambiguation as a first step before part-of-speech tagging or parsing [13].

3.3 Weighted Suffix Automata

Algorithms. The *suffix automaton* of a string u is the minimal deterministic finite automaton recognizing the set of suffixes of u [5, 7]. Its size is linear in the length of u: it has between n and $2n - 1$ states and between $n + 1$ and $3n - 2$ transitions. This automaton can be obtained by minimizing the suffix trie of u. A crucial advantage of suffix automata is that, unlike suffix trees, they do



(a)

(b)

(c)

Fig. 2. (a) A string u represented by a finite automaton. (b) The weighted suffix automaton of u. (c) The weighted suffix oracle of u.

not require the use of ’compact’ transitions (transitions labeled with strings) for the size to be linear in $|u|$. In [7], the notion of *weighted suffix automaton* was introduced. It is defined over the tropical semiring and has the same topology as the suffix automaton. The weight associated to a suffix x by the weighted suffix automaton $\mathrm{SA}(u)$ of a string u is the position of the first occurrence of x in u. A string x is a factor of u iff it is the label of a path π in $\mathrm{SA}(u)$ starting from the initial state. The weight of π gives the position of the first occurrence of x in u. A weighted suffix automaton can be built by an on-line algorithm deriving $\mathrm{SA}(u\sigma)$ from $\mathrm{SA}(u)$ for $\sigma \in \Sigma$. This algorithm is based on the definition of failure transitions similar to the suffix links defined in a suffix tree. The complexity of the on-line construction algorithm is $O(\log(|\Sigma|)|u|)$ in time and $O(|u|)$ in space.

The *weighted suffix oracle* $\mathrm{SO}(u)$ of a string u is an approximation of the suffix automaton recognizing a superset of the set of suffixes of u [2]. It has exactly $|u| + 1$ states and at most $2|u| - 1$ transitions. The weight associated by $\mathrm{SO}(u)$ to a string x is the position of the first occurrence of x in u when x a suffix of u. The construction algorithm is a simplified version of the on-line construction algorithm of the suffix automaton, its complexity is $O(\log(|\Sigma|)|u|)$ in time and $O(|u|)$ in space.

Utilities. The on-line construction algorithms of the weighted suffix automaton and oracle have been implemented in the GRM library and can be invoked through the `grmsuffix` command-line utility:

```
grmsuffix A.fsm > suffix.fsm
grmsuffix -o A.fsm > oracle_suffix.fsm
```

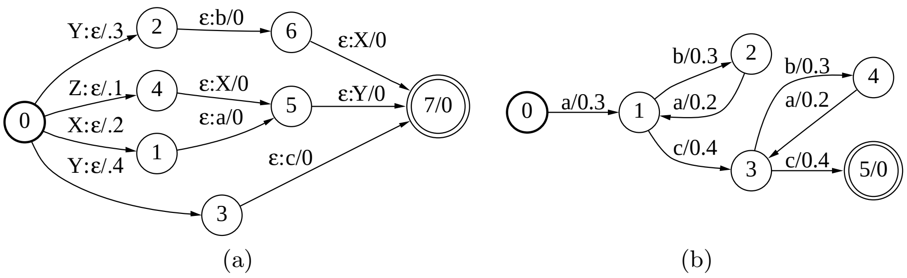


Fig. 3. (a) Binary representation of the context-free grammar G. (b) Compilation of G into a weighted automaton.

This utility takes as input a string represented by a finite automaton and returns the weighted suffix automaton of that string. When the `-o` option is used, the weighted suffix oracle is returned instead.

Examples and Applications. The weighted suffix automaton SA($abbab$) of the string $abbab$ is given Figure 2(b). The weight associated by SA($abbab$) to ab is 3, which is the position in $abbab$ where ab occurs as a suffix, and the weight of the path starting from the initial state and labeled with ab is 0, which is indeed the position of the first occurrence of ab in $abbab$. The weighted suffix oracle SO($abbab$) of $abbab$ is given Figure 2(c). Note that the string $abab$ is recognized by SO($abbab$) although it is not a suffix of $abbab$.

The (weighted) suffix automaton can be used for indexing [6, 7], string-matching [8, 4] and compression [7]. The main application of the suffix oracle is string-matching [2].

4 Context-Free Grammars

The GRM library includes several utilities for reading, compiling, and approximating context-free grammars (CFGs) into finite automata. This section briefly reviews the relevant utilities of the GRM library.

4.1 Textual and Binary Representations

A textual representation of a weighted context-free grammar can be used directly as input to the GRM utilities. The following illustrates that representation in the case of a simple CFG.

CFG rules	`cfg.txt`
Z .1 $\rightarrow XY$	`Z .1 X Y`
X .2 $\rightarrow aY$	`X .2 a Y`
Y .3 $\rightarrow bX$ | .4 c	`Y .3 b X | .4 c`

The textual representation is a straightforward translation of the classical way a CFG is written. Since by definition the first symbol of each rule is a non-terminal, there is no need to keep the arrow symbol for indicating the rule derivation. The second symbol of each line is the weight associated to the rule (in the case of weighted CFGs). The weights can be elements of an arbitrary semiring.

For efficiency purposes, this textual representation can be turned into a binary format using the utility `grmread`. The following is a command line sequence that generates the binary representation `cfg.bin` of the CFG `cfg.txt` where the file `labels` is a user-defined association between the symbols (terminal and non-terminal) and some numbers associated with them.

```
grmread -i labels -w cfg.txt >cfg.bin
```

The flag `-w` indicates that the input CFG is weighted. In the GRM library, the current binary representation is in fact that of a weighted transducer, see Figure 3(a). There are several reasons that motivated that choice. First, this representation makes it natural to apply grammar operations such as union or concatenation directly at the binary level. Secondly, and perhaps more importantly, the use of general determinization and minimization algorithms with this representation increase the sharing (*factoring*) among grammar rules that start or end the same way, which improves dramatically the time and space needed for the grammar compilation.

4.2 Compilation and Regular Approximation

When the input weighted context-free grammar is *strongly regular*, it can be compiled by the GRM library into an equivalent weighted automaton using the utility `grmcfcompile`. A CFG is strongly regular when the rules of each set M of mutually recursive nonterminals are either all right-linear or all left-linear (nonterminals that do not belong to M are considered as terminals for deciding if a rule of M is right- or left-linear). The following illustrates the use of the GRM utility `grmcfcompile` for compiling a CFG given by the binary representation `cfg.bin`.

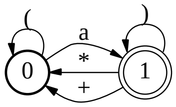
```
grmcfcompile -i labels -s Z cfg.bin >cfg.fsm
```

Figure 3(b) shows the result of the compilation of that grammar. The CFG compilation of the GRM library produces an FSM that can be expanded on-demand. The FSM returned by `grmcfcompile` is a delayed acceptor, thus, its states and transitions are expanded as required by the FSM operation that is applied to it.

Not all weighted CFGs are strongly regular and thus can be compiled into weighted automata using `grmcfcompile`. We have designed an efficient context-free approximation algorithm that transforms any context-free grammar into one that is strongly regular [16]. The algorithm is based on a simple transformation that applies to any context-free grammar. The resulting grammar contains at most one new nonterminal for any nonterminal symbol of the input grammar. The result thus remains readable and if necessary modifiable. A mapping from an arbitrary CFG generating a regular language into a corresponding finite automaton cannot be realized by any algorithm [23]. Thus, in general, our approximation cannot guarantee that the language is preserved when the grammar already generates a regular language (neither can any other approximation). However, this is guaranteed when the grammar is strongly regular.

The GRM utility `grmcfapproximate` takes as input the binary representation of a CFG and produces the textual representation of a strongly regular grammar approximating the input. The approximation creates new non-terminal symbols. The option `-o olab` specifies a new symbols file to be created, `olab`, containing the original and the new symbols. The following illustrates the use of `grmcfapproximate`.

```
grmcfapproximate -i lab -o nlab cfg.bin >ncfg.txt
grmread -i nlab ncfg.txt | grmcfcompile -i nlab -s E >cfg.fsm
```

cfg.txt	ncfg.txt		cfg.fsm
	E' eps	T T	
E E + T	T' eps	T' * F	
E T	F' eps	F' T'	
T T * F	E E	T F	
T F	E' + T	F' T'	
F (E)	T' E'	F (E	
F a	E T	E') F'	
	T' E'	F a F'	

The grammar `cfg.txt` above represents a simple grammar of arithmetic expressions. When applied to `cfg.txt`, `grmcfapproximate` returns the strongly regular grammar `ncfg.txt` that can be compiled into the automaton `cfg.fsm` represented by the figure.

5 Statistical language models

The GRM library includes utilities for counting n-gram occurrences in corpora of text or speech, and for estimating and representing n-gram language models based upon these counts. The use of weighted finite-state transducers allows for an efficient algorithm for computing the expected value of n-gram sequences given a weighted automaton. Failure transitions provide a natural automata encoding of stochastic language models in the tropical semiring. Some of the algorithmic details related to these utilities are presented in [3]. Here we give a brief tutorial on their use.

5.1 Corpora

For counting purposes, a corpus is a collection (or archive) of weighted automata in the log semiring. A corpus of strings can be compiled into such an archive

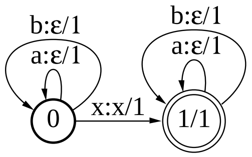


Fig. 4. Counting transducer for sequence x.

corp	wl	
hello	ϵ	0
bye	hello	1
hello	bye	2
bye bye	<s>	3
	</s>	4

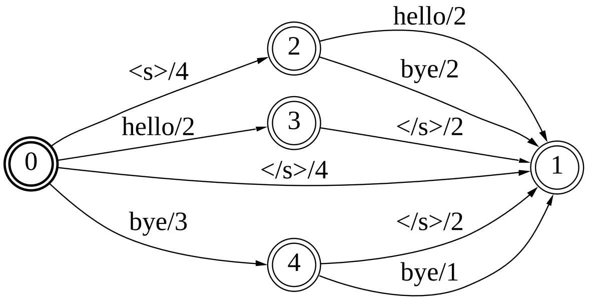


Fig. 5. Example corpus and count automata resulting from the command: `farcompilestrings -i wl corp | grmcount -n2 -s"<s>" -f"</s>" -i wl>bg.fsm`

with the FSM library utility `farcompilestrings`. A collection of word lattices (acyclic weighted graphs of alternative word strings, e.g. output from a speech recognizer) can be simply concatenated together to form an archive. For posterior counts from word lattices, weights should be pushed toward the initial state and the total cost should be removed, using `fsmpush`.

5.2 Counting

We define the *expected count* (the *count* for short) $c(x)$ of the sequence x in A as: $c(x) = \sum_{u \in \Sigma^*} |u|_x \, \llbracket A \rrbracket(u)$, where $|u|_x$ denotes the number of occurrences of x in the string u, and $\llbracket A \rrbracket(u)$ the weight associated to u by A. The transducer in figure 4 can be used to provide the count of x in A through composition with A, projection onto output labels, and epsilon-removal. While we have been mentioning just acyclic automata (e.g., strings and lattices), the algorithm can count from cyclic weighted automata, provided that cycle weights are less than one, a requirement for A to represent a distribution.

The utility `grmcount` takes an archive of weighted automata and produces a count automaton as shown in figure 5. Optional arguments include the n-gram order, start and final symbols, which will be automatically appended to the beginning and end of each automaton to be counted. In addition to `grmcount`, the utility `grmmerge` is provided, which takes k count files of the format produced by `grmcount`, and combines the counts into a single file of the same format. This allows counting to be parallelized, and the results combined. These counting utilities are used as follows:

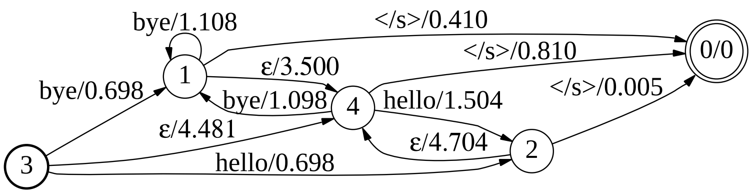


Fig. 6. Bigram language model with ϵ backoff arcs.

```
grmcount -n2 -s3 -f4 foo.far > foo.2g.counts.fsm
grmmerge foo.counts.fsm bar.counts.fsm > foobar.counts.fsm
```

5.3 Making a backoff model from counts

The counts described in the previous section can be used in a variety of ways, e.g., to calculate expected counts and gradients for machine learning algorithms. They can also be used to produce backoff language models, commonly used for automatic speech recognition, and also for speech synthesis, information retrieval, and machine translation. The utility `grmmake` takes counts in the format produced by `grmcount` and produces a backoff model in the tropical semiring:

```
grmmake foo.2g.counts.fsm > foo.2g.lm.fsm
```

Figure 6 shows the bigram language model in the tropical semiring that results from the counts in figure 5. The smoothing technique that is used by default is Katz backoff [10], but the utility also provides for alternative estimation methods, such as absolute discounting [19] and Kneser-Ney smoothing [11]. Backoff transitions are naturally represented as failure transitions, but the `grmmake` utility produces them with ϵ-transitions, a convenient off-line approximation of the failure-function representation.

The utility `grmshrink` takes a model output from `grmmake` and removes transitions when their absence results in a change to the model of magnitude less than some threshold. Two methods are provided, the weighted difference method [20] and the relative entropy method [21]. The utility `grmconvert` converts a model output from `grmmake` or `grmshrink` to a failure class model or an interpolated model. Also, an exact off-line model can be produced from `grmconvert`, using ϵ-transitions instead of failure transitions, as detailed in [3]. These utilities are used as follows:

```
grmshrink -c 4 foo.2g.lm.fsm > foo.2g.s4.lm.fsm
grmconvert -t failure foo.2g.lm.fsm >foo.fail.2g.lm.fsm
```

5.4 Comparison with other utilities

The statistical language modeling utilities in the GRM library are similar in many ways to those in the SRILM toolkit [22], but there are some key differences. All the essential functionalities of SRILM are covered by the combination of the FSM and GRM libraries, some specific functionalities of SRILM being a special case of a more general algorithm provided by the FSM library. In addition, the GRM library is more general by providing, for instance, the ability to deal with arbitrary weight sets, or the ability to use `grmcount` to build a language model from a weighted automata which is essential for a number of natural language processing task: adaptation, bootstrapping and speech mining. Finally, in addition to the statistical language modeling tools, the GRM library also provides the text and grammar processing utilities presented in the previous sections.

6 Conclusion

We presented a general weighted grammar library and illustrated its use in several text and speech processing applications. The GRM library is currently used in a variety of applications, services, and products of AT&T. The binary executables of the library are available for download from the following URL:

`http://www.research.att.com/sw/tools/grm/`

The GRM algorithms and utilities can be used in a similar way in many computational biology applications.

Acknowledgements

We thank our colleagues Donald Hindle, Mark-Jan Nederhof, Fernando Pereira, Michael Riley, and Richard Sproat for their help and contributions to various aspects of the design of GRM library.

References

1. A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.
2. C. Allauzen, M. Crochemore, and M. Raffinot. Efficient experimental string matching by weak factor recognition. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM'2001)*, volume 2089 of *Lecture Notes in Computer Science*, pages 51–72, 2001.
3. C. Allauzen, M. Mohri, and B. Roark. Generalized algorithms for constructing language models. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (ACL 2003)*, pages 40–47, 2003.
4. C. Allauzen and M. Raffinot. Simple optimal string matching. *Journal of Algorithms*, 36(1):102–116, 2000.

5. A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, and J. I. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40(1):31–55, 1985.
6. A. Blumer, J. Blumer, D. Haussler, R. M. McConnel, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987.
7. M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45(1):63–86, 1986.
8. M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12(4/5):247–267, 1994.
9. R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, Cambridge UK, 1998.
10. S. M. Katz. Estimation of probabilities from sparse data for the language model component of a speech recogniser. *IEEE Transactions on Acoustic, Speech, and Signal Processing*, 35(3):400–401, 1987.
11. R. Kneser and H. Ney. Improved backing-off for m-gram language modeling. In *Proceedings of ICASSP*, volume 1, pages 181–184, 1995.
12. D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.
13. M. Mohri. Syntactic analysis by local grammars automata: an efficient algorithm. In *Proceedings of the International Conference on Computational Lexicography (COMPLEX 94)*. Linguistic Institute, Hungarian Academy of Science, 1994.
14. M. Mohri. String-matching with automata. *Nordic Journal of Computing*, 2(2):217–231, 1997.
15. M. Mohri. *Robustness in Language and Speech Technology*, chapter Weighted Grammar Tools: the GRM Library, pages 165–186. Kluwer, 2001.
16. M. Mohri and M.-J. Nederhof. *Robustness in Language and Speech Technology*, chapter Regular Approximation of Context-Free Grammars through Transformation, pages 153–163. Kluwer, 2001.
17. M. Mohri, F. C. N. Pereira, and M. Riley. The design principles of a weighted finite-state transducer library. *Theoretical Computer Science*, 231:17–32, January 2000. http://www.research.att.com/sw/tools/fsm.
18. M. Mohri, F. C. N. Pereira, and M. Riley. Weighted Finite-State Transducers in Speech Recognition. *Computer Speech and Language*, 16(1):69–88, 2002.
19. H. Ney, U. Essen, and R. Kneser. On structuring probabilistic dependences in stochastic language modeling. *Computer Speech and Language*, 8(1):1–38, 1994.
20. K. Seymore and R. Rosenfeld. Scalable backoff language models. In *Proceedings of ICSLP*, volume 1, pages 232–235, Philadelphia, Pennsylvania, 1996.
21. A. Stolcke. Entropy-based pruning of backoff language models. In *Proc. DARPA Broadcast News Transcription and Understanding Workshop*, pages 270–274, 1998.
22. A. Stolcke. SRILM – an extensible language modeling toolkit. In *Proc. Intl. Conf. on Spoken Language Processing (ICSLP'2002)*, volume 2, pages 901–904, 2002.
23. J. Ullian. Partial algorithm problems for context free languages. *Information and Control*, 11:80–101, 1967.