The Design Principles and Algorithms of a Weighted Grammar Library

CYRIL ALLAUZEN

allauzen@research.att.com
AT&T Labs – Research
180 Park Avenue
Florham Park, NJ 07932, USA

and

MEHRYAR MOHRI

mohri@cs.nyu.edu
Courant Institute of Mathematical Sciences
New York University
719 Broadway, 12th Floor
New York, NY 10003, USA

and

BRIAN ROARK

roark@cslu.ogi.edu
Center for Spoken Language Understanding
OGI School of Science & Engineering
Oregon Health & Science University
20000 NW Walker Road
Beaverton, Oregon 97006, USA

ABSTRACT

We present the software design principles, algorithms, and utilities of a general weighted grammar library, the *GRM Library*, that can be used in a variety of applications in text, speech, and biosequence processing. Several of the algorithms and utilities of this library are described, including in some cases their pseudocodes and pointers to their use in applications. The algorithms and the utilities were designed to support a wide variety of semirings and the representation and use of large grammars and automata of several hundred million rules or transitions.

1. Introduction

Most modern grammars used in text and speech processing [24] or bioinformatics [11] are statistical models derived from large datasets. They may be probabilistic context-free grammars, or more generally weighted context-free grammars. In all cases, the weights play a crucial role in their definition since 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 fundamental algorithms for creating, modifying, compiling, and

approximating large weighted statistical or rule-based grammars. Three essential principles guided our design of this grammar library (the *GRM Library*):

Generality. To generalize the applicability of our algorithms, as much as possible, we defined *generic* algorithms, i.e. algorithms that work with as general a weight set or algebra as possible. This approach followed a general *principle of separation of algorithms and algebras* [21, p. 347] that can be viewed as a mathematical analogue of the classical software engineering principle of separation of algorithms and data structures. Many of the algorithms we devised and implemented support a wide variety of weight sets (or semirings). In some cases, we extended existing algorithms to work with different semirings. An interesting example of this extension is our generalized algorithm for constructing weighted suffix automata.

Another motivation for the design of a grammar library was the need for more 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 sequence of weighted automata output by a speech recognizer or an information extraction system, novel algorithms and utilities are needed.

Efficiency. While keeping a high degree of generality, the algorithms were also designed to be very efficient so as 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 [23]), served as the basis for the design of the GRM library. The utilities of our library were used for real-time applications such as the dynamic modification of large weighted grammars in the context of spoken-dialog applications, or for rapid creation of statistical grammars from a very large set of several million sentences or weighted automata. The principle of efficiency also led us to select algorithms that could scale. An example is our choice of an algorithm for regular approximation of context-free grammars. Our experiments showed that several existing approximation algorithms worked only with grammars of a few hundred rules [22]. This motivated the introduction and implementation of faster approximation algorithms leading to minimal deterministic automata practical for applications such as speech recognition [22]. We will report empirical results to illustrate the efficiency of our algorithms and their implementations.

Modularity. The principle of generality helped us avoid redundancy and create a modular core of algorithms. For this, we also benefited from the modularity and generality of our weighted transducer library (FSM Library) which served as the basis for many of our implementations. 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 classes in the GRM library: the *edit*,

2

replace, and *failure* classes. The failure class is described in this article; other classes are defined in detail in the documentation of the library.

This paper gives a general description of several algorithms and utilities of the GRM library, including in some cases their pseudocode, and points out their application to text and speech processing tasks. It also provides an overview of the command-line utilities of the library and a flavor of its C-level programs through a number of examples and code samples. Three main categories of algorithms and utilities of the library are described: statistical language modeling algorithms and tools, local grammar and text processing utilities, and context-free grammar compilation and approximation.[a]

2. 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.

2.1. Notation

The weighted automata considered in this paper are defined over the tropical or the log semirings. A weighted automaton A is defined as a 6-tuple $(Q, \Sigma, E, i, F, \rho)$, where Q is the set of states, Σ the alphabet, $E \subseteq Q \times \Sigma \cup \{\epsilon\} \times \mathbb{R} \times Q$ the set of transitions, $i \in Q$ the initial state, $F \subseteq Q$ the set of final states, and $\rho : F \to \mathbb{R}$ the final weight function. We denote by $p[e]$ the origin of a transition $e \in E$, $l[e]$ its label, $w[e]$ its weight, and $n[e]$ its destination state. The set of outgoing transitions of a state q is denoted by $E[q]$. A weighted automaton is deterministic if no two transitions leaving the same state share the same label. We also denote by $\delta(q, \sigma)$ the set of states reached by transitions leaving q and labeled with σ.

2.2. Corpora

For counting purposes, a corpus is a sequence (or archive) of weighted automata in the log semiring. A corpus of strings such as that of Figure 1(a) can be compiled into such an archive with the utility of the FSM library `farcompilestrings`, where the prefix `FAR` stands for FSM ARchive. The binary representation of a sequence of word lattices (acyclic weighted automata of alternative word strings, e.g., output from a speech recognizer) can be simply concatenated together to form an archive, a `FAR` file. For posterior counts from word lattices, weights should be pushed toward the initial state and the total cost should be removed, using `fsmpush`.

---

[a]Some of the algorithms and utilities of the original version of this library, e.g., the algorithms and utilities for the compilation of weighted context-dependent rules, were presented elsewhere [20].

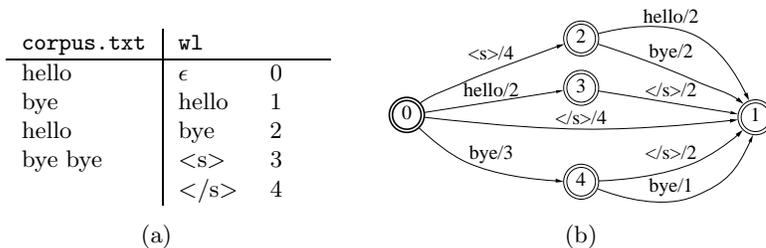| corpus.txt	wl	
hello	ϵ	0
bye	hello	1
hello	bye	2
bye bye	<s>	3
	</s>	4

(a)

(b)

Fig. 1. (a) Example of a small corpus (`corpus.txt`) and its corresponding word list (`wl`). (b) Count automaton output by the commands: `farcompilestrings -i wl corpus.txt | grmcount -n2 -s"<s>" -f"</s>" -i wl > bg.fsm`.

2.3. Counting

We define the *expected count* (the *count* for short) $c(x)$ of the sequence x in A as: $c(x) = \sum_{u \in \Sigma^*} |u|_x \, [\![A]\!](u)$, where $|u|_x$ denotes the number of occurrences of x in the string u, and $[\![A]\!](u)$ the weight associated to u by A. The transducer of Figure 2 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. There exists a general algorithm for computing efficiently higher order moments of the distributions of the counts of a sequence x in a weighted automaton A [8].

The utility `grmcount` takes an archive of weighted automata and produces a count automaton as shown in Figure 1(a)-(b). Optional arguments include the n-gram order, and the start and final symbols, which are represented by <s> and </s> respectively in the examples of this section. These symbols are automatically appended by `grmcount` 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:

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

The following source code shows how the `grmcount` utility just described can be implemented with a simple call to the library function `GRMCountNgrams`.

```
Fsm ifsm, cfsm = NULL;
int start = 3, stop = 4, order = 2;
FILE *fp = fopen("foo.far", "rb");
for(nfsm = 1; ifsm = FSMArchiveRead(fp, "foo.far", nfsm); nfsm++)
    cfsm = GRMCountNgrams(ifsm, cfsm, order, start, stop, FSMDestruct);
FSMDump("foo.2g.counts.fsm", cfsm);
fclose(fp);
```
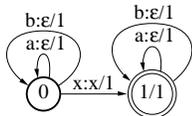
4

Fig. 2. Counting transducer for sequence x.

2.4. Creating a back-off model from counts

The counts described in the previous section can be used in a variety of applications, e.g., to compute expected counts and gradients for machine learning algorithms. They can also be used to produce n-gram back-off language models, commonly used in many natural language processing applications, e.g., automatic speech recognition, speech synthesis, information retrieval, or machine translation.

An n-gram model is based on the Markovian assumption that the probability of the occurrence of a word only depends on the $n-1$ preceding words. Thus,

$$\mathbf{P}(w_1 \ldots w_k) = \prod_{i=1}^{k} \mathbf{P}(w_i \mid h_i) \tag{1}$$

where the conditioning history h_i has length at most $n-1$: $|h_i| \leq n-1$. Let $c(hw)$ denote the count of n-gram hw and let $\widehat{\mathbf{P}}(w \mid h)$ be the maximum likelihood probability of w given h, estimated from counts. $\widehat{\mathbf{P}}$ is often adjusted to reserve some probability mass for unseen n-gram sequences. Denote by $\widetilde{\mathbf{P}}(w \mid h)$ the adjusted conditional probability. For all n-grams $h = wh'$ where $h \in \Sigma^k$ for some $k \geq 1$, we refer to h' as the Katz back-off n-gram of h [12]. Conditional probabilities in a back-off model are of the form:

$$\mathbf{P}(w \mid h) \quad = \quad \begin{cases} \widetilde{\mathbf{P}}(w \mid h) & \text{if } c(hw) > 0 \\ \alpha_h \mathbf{P}(w \mid h') & \text{otherwise} \end{cases} \tag{2}$$

where α_h is a factor that ensures a normalized model. In practice, for numerical stability, negative log probabilities are used.

Back-off language models admit a natural representation by weighted automata using failure transitions. A failure transition is a special transition that is taken when no standard transition with the desired input label is found (see section 3.1). The set of states of the weighted automaton representing a back-off model is defined by associating a state q_h to each n-gram h of order strictly less than n found in the corpus, $i.e.$ $Q = \{q_h : |h| < n \text{ and } c(h) > 0\}$. Its transition set E consists of failure transitions labeled by ϕ and of regular transitions:

$$E = \{(q_{wh'}, \phi, \alpha_h, q_{h'}) : q_{wh'} \in Q\} \cup \{(q_h, w, \tilde{\mathbf{P}}(w|h)), n_{hw}) : c(hw) > 0\}$$

where n_{hw} is defined as q_{hw} if $|hw| < n$ and as $q_{h'w}$ if $|hw| = n$.

Figure 3 gives the pseudocode of the algorithm to change the topology of a weighted automaton A encoding counts of n-grams of order at least n to the topology of a back-off model of order n. The discounting, smoothing and normalization steps need to be applied subsequently. For each call of AddBackoff except from the initial one ($q = q' = i$), q' is the back-off of q, thus a failure transition from q to q' is added (line 1). For every outgoing transition e, the back-off state of $n[e]$ is identified as q'' lines 3-4. If the order of the n-gram

5

AddBackoff(A, q, q', k, n)
   1   **if** $q \neq q'$ **then** $E \leftarrow E \cup \{(q, \phi, 0, q')\}$
   2   **for** $e \in E[q]$ **do**
   3   **if** $q = q'$ **then** $q'' \leftarrow q'$
   4   **else** $q'' \leftarrow \delta(q', l[e])$
   5   **if** $k + 1 < n$ **then** AddBackoff$(A, n[e], q'', k + 1, n)$
   6   **else** $n[e] \leftarrow q''$

MakeModel(A, n)
   7   AddBackoff$(A, i, i, 0, n)$
   8   **if** $n > 1$
   9   **then** let $e \in E[i]$ such that $l[e] =$<s>
 10        $i \leftarrow n[e]$; $E \leftarrow E - \{e\}$
 11   create a new state f
 12   **for** $e \in E$ **do** **if** $l[e] =$</s> **then** $n[e] \leftarrow f$
 13        $F \leftarrow \{f\}$
 14   remove non accessible and non coaccessible states

Fig. 3. Pseudocode of the algorithm to change the topology of a weighted automaton A encoding counts to the topology of a back-off model of order n.
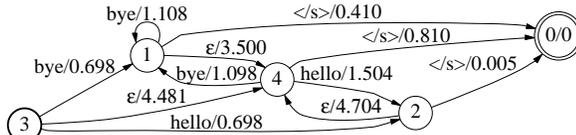

Fig. 4. Bigram language model with ϵ back-off arcs.

corresponding to e is strictly less than n, AddBackoff is called to set the back-off state of $n[e]$ to q''. Otherwise, $n[e]$ is set to q''. This corresponds to the case where n_{hw} is $q_{h'w}$ and not q_{hw}. If the order n is at least 2, the initial state is set to the state representing the n-gram <s> (lines 8-10) and every transition labeled by </s> is redirected to the unique final state of the model, representing the n-gram **</s>** (lines 11-14).

The utility `grmmake` takes counts in the format produced by `grmcount` and produces a back-off model in the tropical semiring:

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

Figure 4 shows the bigram language model in the tropical semiring that results from the counts in Figure 1. The smoothing technique that is used by default is Katz back-off [12], but the utility also provides for alternative estimation methods, such as absolute discounting [27] and Kneser-Ney smoothing [13]. As previously mentioned, back-off transitions are naturally represented as failure transitions, but the `grmmake` utility produces them with ϵ-transitions, a convenient off-line approximation. These ϵ-transitions can be changed to failure transitions using `grmconvert`.

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 [29] and the relative

entropy method [31]. 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
```

2.5. Applications and benchmarks

The GRM library statistical language modeling utilities can be used in the manner presented here to produce language models for use in, e.g., automatic speech recognition (ASR) or machine translation (MT). One benefit for ASR of encoding such models as weighted finite automata lies in the ability to compose the models offline with other speech transducers, including pronunciation dictionaries, context-dependent phone transducers, and hidden Markov models, and optimizing the resulting transducers for efficient decoding [26, 25, 24].

The models built for these applications are often very large, so the utilities must scale up to handle automata of tens or hundreds of millions of states and transitions. To demonstrate the scalability of the utilities presented here, table 1 shows model sizes and training times for three training corpora, of size one, ten and a hundred million words. All of the training corpora are from the LDC North American Business (NAB) corpus, and training was performed on an Intel Pentium 4 3.2GHz CPU with 2GB of RAM.

In addition to its use for generative n-gram language models, such as those produced by `grmmake`, the counts produced by `grmcount` can also feed other parameter estimation techniques from the machine learning literature. For example, conditional log-linear models, such as conditional random fields [15], have been used for a range of sequence learning tasks, including shallow parsing [30] and named entity extraction [17]. Recently the counting utility from the GRM Library was used to efficiently calculate feature gradients for a related discriminative n-gram language modeling approach [28]. Briefly, let x_j be utterance j of n training examples $\mathcal{D}$; let y_j be the true transcription of x_j; and let Z_j be the set of candidate transcriptions of x_j (i.e. a weighted word lattice). Then, in this approach, the derivative (or gradient) of the conditional log likelihood $L(\mathcal{D})$ with respect to a particular parameter α_i is

$$\frac{\partial L(\mathcal{D})}{\partial \alpha_i} \quad = \quad \sum_{j=1}^{n} \left[f_i(y_j) - \sum_{z \in Z_j} \frac{p(z)}{\sum_{z' \in Z_j} p(z')} f_i(z) \right] \tag{3}$$

where $f_i(z)$ is the count of feature i in transcript z. In the case of n-gram models, f_i would be a particular n-gram feature count. The utility `grmcount` was used in [28] for counting both terms of the sum above: $f_i(y_j)$ and $\sum_{z \in Z_j} p(z|Z_j) f_i(z)$. This latter sum is simply the expected count of the n-gram feature f_i in the weighted word lattice Z_j with the costs pushed. This efficient counting approach scaled effectively to millions of parameters and hundreds of thousands of training word lattices. In addition, the failure-class was used to encode the resulting model efficiently for intersection with word lattices. See [28] for more details on the approach.

7

Table 1. Counts of n-grams, states and transitions and run times of the GRM
utilities used to create a trigram model from a corpus (including I/O's) on an
Intel Pentium 4 3.2GHz CPU with 2GB of RAM.

Counts of	Corpus size (million words)		
	1	10	100
unigrams	29,830	72,858	138,103
bigrams	312,216	1,649,338	8,027,537
trigrams	626,460	4,562,590	31,020,778
states	342,047	1,722,197	8,165,641
transitions	1,310,550	8,006,980	47,352,056

time (s)	Corpus size (million words)		
	1	10	100
farcompilestrings -iwl -u<unk> Corpus >far	5.1	48.4	480.9
grmcount -n3 -iwl -s<s> -f</s> far >c.fsm	28.5	324.6	5736.3
grmmake c.fsm >m.fsm	7.7	51.9	373.7
grmshrink m.fsm > m.s.fsm	6.9	66.7	591.4
grmconvert -t fail m.s.fsm >m.fs.fsm	2.6	17.3	110.7
total	50.8	508.9	7293.0

2.6. Comparison with other utilities

The statistical language modeling utilities of the GRM library are similar in many
ways to those of the SRI Language Modeling Toolkit (SRILM toolkit) [32], but there are
some key differences. The SRILM toolkit provides a large variety of scripts and utilities
for not only counting and creating language models, but also for the use and manipulation
of these models. Since the models produced by the GRM library are in the format used by
the FSM library, they can be readily used and manipulated with existing FSM utilities.
Hence additional utilities are not part of the core GRM library.

For example, to score a string with a language model, the string must simply be encoded
as an automaton (*farcompilestrings*) and intersected with the model (*fsmintersect*). Many
of the same modeling options are provided by the utilities in both the SRILM toolkit
and the GRM library, as well as count merging and model pruning capabilities. Class-
based modeling is included explicitly in the SRILM toolkit, but, as shown in [3], general
class-based models can be straightforwardly represented with the GRM library, without
requiring additional utilities, through the use of weighted transducers [7]. With such an
approach, classes can be (weighted) regular languages, rather than just a finite set of words
or a finite list of sequences of words.

The GRM library provides some features that are not covered by the SRILM Toolkit.
It allows for counting from weighted automata, e.g., word lattices, which is crucial in a
number of text and speech processing applications. Also, the use of failure transitions for
the representation of language models and its off-line approximation based on ϵ-transitions
provide efficient and useful encodings for intersection and composition with other finite
automata and finite-state transducers. Finally the GRM's tight coupling with the FSM
library allows one to benefit from the wide range of utilities of that library. In reverse,
some of the features provided by the SRILM Toolkit, e.g., different discounting methods
such as that of Witten-Bell are not provided by the current release of the GRM library

```
typedef struct fsm_rec {          typedef struct failure_rec {
  void *data;                       Fsm fsm;
  FSMTypes type;                    int phi;
  FSMClass clas;                    unsigned char side;
  Smr smr;                          DSTPool match_pool;
  SYSHandle hdl;                  } *Failure;
  int refcnt;
} *Fsm;
```

Fig. 5. Definition of the C types `Fsm` and `Failure`

but are likely to be available in future versions. The SRILM Toolkit also provides a utility for converting its models to and from that of the FSM library.

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 [1, 19]. 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 are used in the GRM library to represent local grammars (Section 3.2) and back-off language models (Section 2.4).

The use of failure transitions is made possible in the GRM library through a dedicated FSM class, the *failure class*. In the FSM library, a weighted automaton or transducer is represented by an object of type `Fsm` defined figure 5. The basic idea is that the member `data` points to the actual memory representation and `clas` specifies methods to access this memory representation. For instance, `clas->start` is a pointer to a function that takes as argument an `Fsm fsm` and returns the index of the initial state of the weighted automata or transducer represented by `fsm`.

The failure class is defined as an object `FSMFailureClass` of type `FSMClass`. In an `fsm` of that class, the `data` pointers is of the type `Failure` defined figure 5. The member `fsm` is the representation of the underlying automata or transducer without failure transitions, `phi` specifies the label of failure transitions, `side` specifies, if `fsm` is a transducer, whether the failure transitions are defined on the input or output side. `FSMFailureClass->start` is then a pointer to the function `failure_start` defined by:

```
static int failure_start(Fsm fsm){
  Failure failure = (Failure) FSMData(fsm);
  return FSMStart(failure->fsm);}
```

where `FSMData(fsm)` is a macro defined as `fsm->data` and `FSMStart(fsm)` a macro defined as `fsm->clas->start(fsm)`. This function simply returns the initial state of the failure class representation which is the same as that of the underlying representation.

The utility `grmfailure` can convert a regular FSM representation to a failure class

9

representation by interpreting transitions labeled with the symbol `phi` specified by the option `-p` as failure transitions:

<div align="center">

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

</div>

The conversion can be done at the C library level as follows:

```
Fsm fsm = FSMLoad("A.fsm");
int phi = 18, side = 0;
fsm = GRMFailure(fsm, phi, side);
FSMDump("A.failure.fsm", fsm);
```

3.2. Local Grammars

3.2.1. 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 given by [19]. The following is the pseudocode of that algorithm in the case where A is acyclic.

LocalGrammar(A)
1   $E \leftarrow E \cup \{(i, \phi, i)\}$
2   ENQUEUE(S, i)
3   **while** $S \neq \emptyset$ **do**
4          $p \leftarrow$ DEQUEUE(S)
5   **for** $e \in E[p]$ **do**
6             $q \leftarrow \delta(p, \phi)$
7   **while** $q \neq i$ **and** $\delta(q, l[e]) =$ UNDEFINED **do** $q \leftarrow \delta(p, \phi)$
8   **if** $p \neq i$ **and** $\delta(q, l[e]) \neq$ UNDEFINED
9   **then** $q \leftarrow \delta(q, l[e])$
10   **if** $\delta(n[e], \phi) =$ UNDEFINED
11   **then** $\delta(n[e], \phi) \leftarrow q$
12   **if** $q \in F$ **then** $F \leftarrow F \cup \{n[e]\}$
13                    $L[n[e]] = L[n[e]] \cup \{n[e]\}$
14   ENQUEUE($S, n[e]$)
15   **else if** there exists $r \in L[o[n[e]]]$ such that $(r, \phi, q) \in E$
16   **then** $n[e] \leftarrow r$
17   **else if** $o[q] \neq n[e]$
18   **then** create new state r
19   **for** $e' \in E[n[e]]$ such that $l[e'] \neq \phi$ **do**
20                        $E \leftarrow E \cup \{(r, l[e'], o[n[e']])\}$
21                    $E \leftarrow E \cup (r, \phi, q)$
22                    $o[r] \leftarrow o[n[e]]$
23   **if** $o[n[e]] \in F$ **then** $F \leftarrow F \cup \{r\}$
24                    $L[o[n[e]]] = L[o[n[e]]] \cup \{r\}$
25                    $n[e] \leftarrow r$
26   ENQUEUE(S, r)
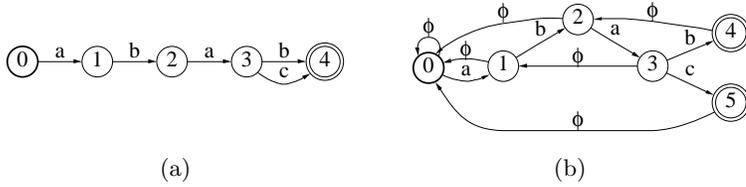27   **else** $n[e] \leftarrow q$

Fig. 6. (a) A deterministic finite automaton A and (b) a deterministic automaton recognizing $\Sigma^* L(A)$ where transitions labeled with ϕ are failure transitions.

The general algorithm of [19] can be seen as a generalization to the case of an arbitrary deterministic automaton A of the classical algorithms of [14] and [1] which were designed for strings or trees. When A is a tree, the complexity of the algorithm of [19] coincides with that of [1]: it is linear in the sum of the lengths of the strings accepted by A. The following describes the algorithm and the pseudocode.

The algorithm takes as input a deterministic unweighted automaton A and modifies it to represent the corresponding local grammar. States of A are visited in the order of a breadth-first search using a FIFO queue S. In the description that follows, we will say for short that x is recognized in q when there is a path from i to q labeled by x.

Each state q admits a failure transition labeled by ϕ. The destination state of that transitions is the *failure state* of q, that is the state where the longest proper suffix of the strings recognized in q prefix of $L(A)$ is recognized. Two distinct paths reaching q may correspond to two distinct failure states for q. In that case, q must be duplicated. Thus, the algorithm maintains the two following attributes: $o[q]$, the original state from which q was copied and, if q was originally in A (*i.e.* $o[q] = q$), $L[q]$ the set of states obtained by copying q.

For each state p extracted from the queue (line 4), each of the outgoing transition e is examined. The (candidate) failure state q of $n[e]$ is determined (lines 6-10) as the first state on the failure path of p that has an outgoing transition labeled by $l[e]$. If $n[e]$ does not already have a failure state, its failure state is set to q and $n[e]$ is added to the queue (lines 10-14). If there exists a state r that has the same original state as $n[e]$ and has q as a failure state, then the destination of e is changed to r (lines 15-16). If q is not a copy of $n[e]$, then a new state r is created by copying $n[e]$, the failure state of r is set to q, the destination state of e is changed to r and r is added to the queue (lines 17-26). Otherwise, the destination state of e is changed to q (line 27).

When A is not acyclic, the condition of the test of line 17 needs to be generalized as described in detail in [19].

3.2.2. Utility.

The algorithm of [19] was implemented in the GRM Library. The library utility **grmlocalgrammar** takes as input a deterministic finite automaton A and returns a deterministic finite automaton recognizing $\Sigma^* L(A)$ 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
```

3.2.3. Examples and Applications.

A deterministic finite automaton A is given by Figure 6(a) and the corresponding automaton recognizing $\Sigma^* L(A)$ is given by Figure 6(b), the failure transitions being labeled with ϕ. The main applications of local grammars are string-matching [1, 19] and disambiguation as a first step before part-of-speech tagging or parsing [18].

3.3. Weighted Suffix Automata

3.3.1. Algorithms.

The *suffix automaton* of a string u is the minimal deterministic finite automaton recognizing the set of suffixes of u [5, 9]. Its size is linear in the length of u. More precisely, its number of states is between $|u|$ and $2|u| - 1$ and its number of transitions between $|u| + 1$ and $3|u| - 2$. 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 not require the use of 'compact' transitions (transitions labeled with strings) for the size to be linear in $|u|$. In [9], the notion of *weighted suffix automaton* was introduced. It is defined over the tropical semiring and has the same topology as the suffix automaton. Let $SA(u)$ be the weighted suffix automaton of a string u and let x be a suffix of u. The weight associated by $SA(u)$ to x is the position of the suffix x in u. A string x is a factor of u iff it is the label of a path π in $SA(u)$ 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 $SA(u\sigma)$ from $SA(u)$ 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, which is the same as the complexity of the best algorithms for constructing suffix trees.

The *weighted suffix oracle* $SO(u)$ 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 $SO(u)$ to a string x is the position in u where x would occurs if x was 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. The pseudocode of the algorithm for the construction of the weighted suffix automaton (case where **oracle** = FALSE) and oracle (**oracle** = TRUE) of a string u is given below.

SuffixAutomaton(u, **oracle**)

```
1       create automaton A with initial state i
2       d[i] ← 0; p[i] ← 0
3       E ← E ∪ {(i, φ, 0, i)}; F ← {i}
4       p ← i
5       for  k ∈ [0, |u| − 1] do
6            create new state q
7            d[q] ← d[p] + 1; p[q] ← p[p] + 1
8            while  p ≠ i  and δ(p, u_k) = UNDEFINED do
9                   E ← E ∪ {(p, u_k, p[q] − p[p] − 1, q)}
10                  p ← δ(p, φ)
```

```
11              if  δ(p, u_k) = UNDEFINED
12                 then  E ← E ∪ {(i, u_k, p[q] − 1, q), (q, φ, 0, i)}
13              else  if  oracle = TRUE or d[p] + 1 = d[δ(p, u_k)]
14                     then  E ← E ∪ {(q, φ, 0, δ(p, u_k))}
15              else  create new state r
16                    for  e ∈ E[δ(p, u_k)] do
17                          E ← E ∪ {(r, l[e], w[e], n[e])}
18                          if  l[e] = φ then  n[e] ← r
19                    d[r] ← d[p] + 1; p[r] ← p[δ(p, u_k)]
20                    E ← E ∪ {(q, φ, 0, r)}
21                    while  d[δ(p, u_k)] ≥ d[r] do
22                          there exists e ∈ E[p] such that l[e] = u_k
23                          n[e] ← r; w[e] ← p[p] − p[r] − 1
24                          p ← δ(p, φ)
25          p ← q
26     while  p ≠ i do
27           F ← F ∪ {p}
28           ρ(p) ← p[q] − p[p]
29           p ← δ(p, φ)
30     return A
```

We give a brief overview of the algorithm (see [9] for more details and the proofs). We will say for short that x is recognized in state q when there is a path from i to q labeled by x. For every state q, the algorithm maintains two attributes: $d[q]$, the length of the longest path from i to q, and $p[q]$, the length of the longest suffix than can be read from q. Every state q has a failure transition labeled with ϕ, its destination state, the *failure state* of q, being the state where the longest proper suffix of the strings recognized in q is recognized. Note that, when building the suffix automaton, the additional condition that the longest string recognized in the failure state of q is a suffix of all the strings recognized in q must be verified.

Starting from line 6, the existing suffix automaton (or oracle) of $u_0 \ldots u_{k-1}$ is extended to construct the suffix automaton (or oracle) of $u_0 \ldots u_k$. p is the state where $u_0 \ldots u_{k-1}$ is recognized. Lines 6-7 create a state q where $u_0 \ldots u_k$ is recognized. While p is not the initial state and p does not have a transition labeled by u_k, a new transition to q labeled with u_k is added and p is set to its failure state (lines 8-10). This loop identifies p as the potential failure state for q. If $p = i$, then there was a transition from i to q labeled by u_k and i is set to be the failure state of q (lines 11-12). If the oracle is being constructed, or if the outgoing transition in p labeled by u_k belongs to the longest path to $\delta(p, u_k)$ from i, then p is the failure state of q (lines 13-14).

Otherwise, a copy r of $\delta(p, u_k)$ must be created, the failure state of $\delta(p, u_k)$ and q must be set to r (lines 15-20), and the transitions labeled with u_k to $\delta(p, u_k)$ from p and states along the failure path of p must be redirected to r (lines 21-24). p is then set to q for the next iteration of the for loop (line 25). Finally, the final states of the automaton are identified as the states belonging to the failure path from q, the state where u is recognized (lines 26-29). Note that the automaton created at line 30 contains failure transitions and is then the local grammar for the suffix automaton or oracle. To obtained the actual suffix automaton or oracle, the failure transitions need to be removed.
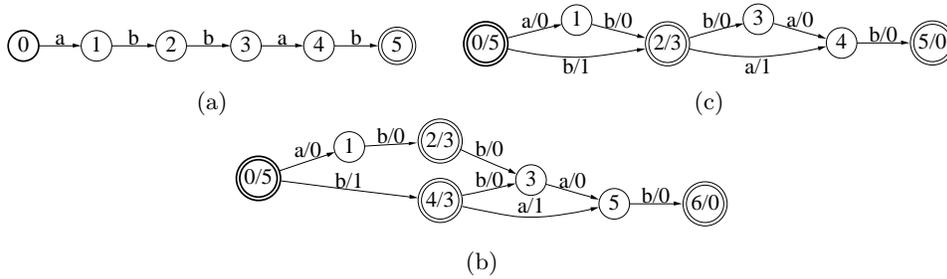
Fig. 7. (a) A string u represented by a finite automaton. (b) The weighted suffix automaton of u. (c) The weighted suffix oracle of u.

Table 2. Running time of `grmsuffix` on an Intel Pentium 4 3.20GHz CPU with 2GB of RAM, including I/O's, and size of the suffix automata produced.

Input size		Suffix automata	
(no. of words)	time (s)	states	transitions
1,000,456	15.1	1,297,378	2,131,255
10,000,873	208.4	13,128,749	21,211,487

3.3.2. 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
```

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.

3.3.3. Examples and Applications.

The weighted suffix automaton SA(*abbab*) of the string *abbab* is given by Figure 7(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 7(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, 9], string-matching [10, 4] and compression [9]. The main application of the suffix oracle is string-matching [2]. Table 2 gives the runtime benchmarks for the construction of suffix automata from long inputs of about one million or ten million words, using `grmsuffix` on an Intel Pentium 4 3.2GHz CPU with 2GB of RAM demonstrating the efficiency of the algorithm and the utility.
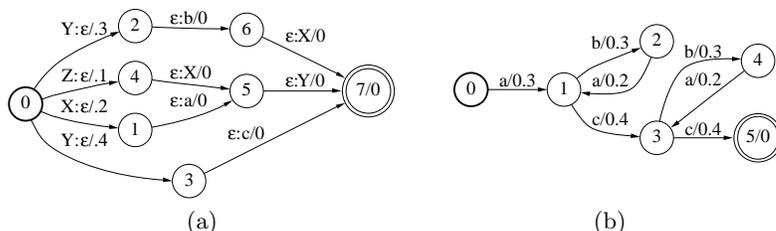
Fig. 8. (a) Binary representation of the context-free grammar G. (b) Compilation of G into a weighted automaton.

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 8(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

15

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 8(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 [22]. 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 [33]. 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
```

The grammar `cfg.txt` below 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.

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'	

We induced a weighted CFG from sections 2-21 of the Penn Wall St. Treebank [16], including all unary part-of-speech to lexical terminal productions, resulting in 62 459 rules. This is a very commonly used broad coverage CFG of English. The grammar is not strongly regular, hence context-free approximation had to occur before compiling the grammar. The entire grammar compilation process – reading the grammar with `grmread`, approximating it with `grmcfapproximate`, reading the approximated grammar with `grmread` and finally compiling the grammar with `grmcfcompile` – took a total of 3.25 seconds on an Intel Pentium 4 3.20GHz CPU with 2GB of RAM.

5. Conclusion

We presented a general weighted grammar library and emphasized its use in several text and speech processing applications. The binary executables of the library are available for download from the following URL:

<div align="center">

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

</div>

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

Acknowledgments

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 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 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. P. F. Brown, V. J. D. Pietra, P. V. deSouza, J. C. Lai, and R. L. Mercer. Class-based n-gram models of natural language. *Computational Linguistics*, 18(4):467–479, 1992.

8. C. Cortes and M. Mohri. Distribution Kernels Based on Moments of Counts. In *Proceedings of the Twenty-First International Conference on Machine Learning (ICML 2004)*, Banff, Alberta, Canada, July 2004.

9. M. Crochemore. Transducers and Repetitions. *Theoretical Computer Science*, 45(1):63–86, 1986.

10. 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.

11. 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.

12. 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.

13. R. Kneser and H. Ney. Improved Backing-off for M-gram Language Modeling. In *Proceedings of ICASSP*, volume 1, pages 181–184, 1995.

14. 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.

15. J. Lafferty, A. McCallum, and F. Pereira. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proceedings of the 18th International Conference on Machine Learning*, pages 282–289, 2001.

16. M. P. Marcus, B. Santorini, and M. A. Marcinkiewicz. Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330, 1993.

17. A. McCallum and W. Li. Early Results for Named Entity Recognition with Conditional Random Fields, Feature Induction and Web-Enhanced Lexicons. In *Seventh Conference on Natural Language Learning (CoNLL)*, 2003.

18. 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.

19. M. Mohri. String-Matching with Automata. *Nordic Journal of Computing*, 2(2):217–231, 1997.

20. M. Mohri. *Robustness in Language and Speech Technology*, chapter Weighted Grammar Tools: the GRM Library, pages 165–186. Kluwer, 2001.

21. M. Mohri. Semiring Frameworks and Algorithms for Shortest-Distance Problems. *Journal of Automata, Languages and Combinatorics*, 7(3):321–350, 2002.

22. 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.

23. 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.

24. 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.

25. M. Mohri and M. Riley. Integrated Context-Dependent Networks in Very Large Vocabulary Speech Recognition. In *Proceedings of the 6th European Conference on Speech Communication and Technology (Eurospeech '99)*, Budapest, Hungary, 1999.

26. M. Mohri and M. Riley. Network Optimizations for Large Vocabulary Speech Recognition. *Speech Communication*, 28(1):1–12, 1999.

27. H. Ney, U. Essen, and R. Kneser. On Structuring Probabilistic Dependences in Stochastic Language Modeling. *Computer Speech and Language*, 8(1):1–38, 1994.

28. B. Roark, M. Saraclar, M. Collins, and M. Johnson. Discriminative Language Modeling with Conditional Random Fields and the Perceptron Algorithm. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics*, pages 47–54, 2004.

29. K. Seymore and R. Rosenfeld. Scalable backoff language models. In *Proceedings of ICSLP*, volume 1, pages 232–235, Philadelphia, Pennsylvania, 1996.

30. F. Sha and F. Pereira. Shallow Parsing with Conditional Random Fields. In *Proceedings of HLT-NAACL*, Edmonton, Canada, 2003.

31. A. Stolcke. Entropy-Based Pruning of Backoff Language Models. In *Proc. DARPA Broadcast News Transcription and Understanding Workshop*, pages 270–274, 1998.

32. A. Stolcke. SRILM – An Extensible Language Modeling Toolkit. In *Proc. Intl. Conf. on Spoken Language Processing (ICSLP'2002)*, volume 2, pages 901–904, 2002.

33. J. Ullian. Partial Algorithm Problems for Context Free Languages. *Information and Control*, 11:80–101, 1967.